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

GRADUAL RETRIEVAL, RANKED ENUMERATION, AND LAZY EVALUATION OF GRAPH DATABASE JOINS

TESIS PARA OPTAR AL GRADO DE
MAGÍSTER EN COMPUTACIÓN

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERA CIVIL EN COMPUTACIÓN

ASUNCIÓN CAROLINA GÓMEZ COLOMER

PROFESOR GUÍA:
GONZALO NAVARRO BADINO

MIEMBROS DE LA COMISIÓN:
DIEGO ARROYUELO
AIDAN HOGAN
CRISTÓBAL NAVARRO

Este trabajo ha sido parcialmente financiado por Instituto Milenio Fundamentos de los Datos y Proyecto FONDECYT 1-230755

SANTIAGO DE CHILE
2025

Resumen

En bases de datos de grafos, el *join* es de las operaciones más costosas y, al mismo tiempo, más frecuentes. Los sistemas tradicionales usan algoritmos de *join* de a pares, lo cual se ha demostrado que es subóptimo.

Nuevos enfoques consisten en hacer el *join* de múltiples tablas a la vez, lo cual puede reducir el tiempo y espacio, y ha permitido diseñar algoritmos *worst-case optimal*, es decir, óptimos en el peor caso.

El uso de estructuras de datos compactas ha permitido ir más allá: no solo crear algoritmos *worst-case optimal*, sino también reducir considerablemente el espacio requerido, sin afectar la eficiencia en las operaciones. Arroyuelo et al. (2018) diseñaron un nuevo algoritmo de join basado en el uso de *qdags*, una versión compacta de los árboles quadtree. En su algoritmo buscan optimizar el tiempo de cómputo total del *join* así como el espacio en memoria.

Si bien este algoritmo es competitivo e incluso más eficiente que otros algoritmos usados en los actuales sistemas de bases de datos de grafos para calcular todo el *join*, en muchos casos nos basta con encontrar unos pocos resultados.

El objetivo de esta tesis es diseñar e implementar estrategias para disminuir el tiempo de obtención de algunas tuplas del *join*. La primera parte de este trabajo, *"Gradual retrieval"*, busca entregar resultados lo antes posible. La segunda parte, *"Ranked enumeration"* busca obtener aquellos resultados más importantes ordenados de acuerdo a un peso o prioridad. La tercera parte, *"Lazy qdags"*, tiene como objetivo implementar la versión perezosa de los *qdags*, la cual permite evaluar toda el álgebra relacional (no solo el *join*) sobre bases de datos y además, puede disminuir el tiempo de cómputo de los primeros resultados en algunos casos.

El código del proyecto se encuentra público en www.github.com/asugomez/qdags.

Abstract

In graph databases the join is one of the most expensive and, at the same time, most frequent operations. Traditional systems use pairwise join algorithms, which are suboptimal.

New approaches build on multi-way join algorithms, that is, joining multiple tables at once, which can reduce time and space. These approaches have enabled the design of worst-case optimal algorithms.

The use of compact data structures has allowed the research to go further, not only creating worst-case optimal algorithms but also reducing the required space considerably without affecting the efficiency of the operations. Arroyuelo et al. (2018) designed a new join algorithm using *qdags*, a compact version of quadtrees. Their algorithm seeks to optimize the total computation time of the join and the memory space.

Although this algorithm is proficient in computing the full join and is more efficient than other algorithms used in current graph database systems, applications often only need to find a few results.

This thesis aims to design and implement a new strategy to reduce the time needed to obtain some results of the join. The first part of this work, *"Gradual retrieval"*, aims to output results as soon as possible. The second part, *"Ranked enumeration"*, aims to obtain the most important results according to weight or priority. The third part, *"Lazy qdags"*, aims to implement the lazy version of the qdags, which allows the evaluatiion of the full relational algebra (not just the join) on databases and can also reduce the computation time of the first results in some cases.

The project code is available at github.com/asugomez/qdags.

Agradecimientos

No hubiese sido mi paso por la universidad tan grato sin todas las amistades que pude hacer. Agradezco a mis amigos que hice de mechona, a quienes conocí en Proyecto Reinserción, a mis amigos de computación, y a mis compañeras de fútbol. Con cada uno de ellos viví experiencias que espero recordar siempre. No puedo dejar de nombrar a mis amigos y amigas del colegio, quienes, a pesar del tiempo, fueron un apoyo constante durante todos estos años.

Agradezco al Instituto Milenio Fundamento de los Datos por darme un ambiente de trabajo tan cálido y humano. Allí no solo pude participar de workshops y clubs de lecturas, también encontré un espacio acogedor para hacer investigación con un enfoque social.

Gracias a Gonzalo, quien me motivó a participar en charlas y seminarios incluso antes de ingresar al magíster. Su pasión por lo que hace es contagiante, y sin duda fue una de las mayores motivaciones para realizar este trabajo y continuar en la investigación.

Por último, a mi familia: gracias por estar siempre presente, apoyándome en cada proyecto que decido hacer. A mis padres, por darme no solo una gran educación, sino también infinitas oportunidades y todo su cariño. A mi hermano, por ser una fuente de distracción y alegría. Y a mi hermana, quien además de mi editora y consejera en el mundo académico, siempre está ahí para escucharme y acompañarme.

Table of Content

Chapter 1

Introduction

Over the years, the need to store information has dramatically increased, but external memory has become slower in comparison with the CPU. In many fields, like Machine Learning, we have to deal with massive amounts of data originating from biology, internet routing, multimedia storage, etc. For this reason, it has become interesting to use less space to store data and still be able to perform operations on it in compact form.

Compact data structures [41] significantly reduce space and support fast operations. The main objective in a compact data structure is to use space close to the data entropy while retaining, if possible, the classic time complexity in the operations.

In this thesis, we are interested in representing large graph databases in a compact form. Graph databases represent binary relations as directed labeled edges and capture relationships and interactions between entities, allowing us to navigate and query complex, interconnected data efficiently. This type of database is used in many applications, such as social networks, transportation networks, semantic web, and knowledge bases. It gives a more intuitive representation of the data and is very useful when the topology of the data is as significant as its content [2, 3, 10].

Graph databases tend to feature multi-joins among many relations [4]. The core of the queries is Basic Graph Patterns (BGP), which translate into multiple joins [52]. Therefore, many works [4, 6, 28, 43, 54] focus on optimizing the join operation, which is usually the most expensive one. Recent works have shown that the classic approach to solving multi-joins via pairwise joins is suboptimal [43]. New multi-join algorithms that are worst-case optimal (wco), like LeapFrog Trie Join (LTJ) [51] or Tetris [36], are a promising advance, but they need a heavy index data structure that has to be stored on disk [6], requiring much space.

Arroyuelo et al. [6] proposed the first worst-case optimal multi-join algorithm focused **on both time and space**. It uses a version of compressed quadtrees called qdags, based on k^d-trees [13, 15], that supports join, union, intersection, and negation in wco time. Qdags could also be extended to all relational algebra operations, yet not guaranteeing worst-case optimal time.

The k^2-tree [15] is a data structure commonly used to represent grids and web graphs or to

store geographic information, among others, where k^2 refers to the arity of the tree. While the classic version needs $\mathcal{O}(1)$ **pointers** per node, the compact representation [41, Chapter 10] uses only $\mathcal{O}(k^2)$ **bits** per internal node and supports the basic navigation operations in constant time. With a k^d-tree, an extension of k^2-trees, we can represent a relation R of d attributes as a d-dimensional grid, where each tuple is represented as a point. With qdags, we can reduce the space and computational time of a full join query $J = R_1 \bowtie R_2 \bowtie ... \bowtie R_n$ with d attributes in total. We first (virtually) extend each of the relations into qdags of a higher dimension d and then intersect them all to get the join result. The main goal is to achieve worst case optimal time and low space complexity for computing the full join.

Most research in this area focuses on maximizing the throughput or, equivalently, minimizing the query completion time. In many database systems, however, the full join can be significantly large, and the user may only be interested in some of the results. Thus, we can be more concerned about the time it takes to show the first k results or find the best or most interesting results. This problem is close to ranked retrieval [50], where we must obtain the most relevant results first. Here, the goal is to return the top k results as soon as possible.

The first part of this thesis focuses on this area, using qdags to compute (i) partial results in less time and (ii) the top k results given a priority. We resort to prioritized retrieval techniques known in similarity search [28]. In this way, we study and implement a version of the qdags that we expect to return partial results in less time and a version that allows the output of ranked results.

For this first part, we study and compare two data structures to represent trees in a compact form and efficiently perform the join algorithm: the Level-Order Unary Degree Sequence (LOUDS) and the Depth-First Unary Degree Sequence (DFUDS), which let us navigate the tree in BFS and DFS order, respectively. We also study two types of techniques for each data structure: **optimal order**, which maintains a priority queue with all the nodes of the output that can be visited next, and **prioritized backtracking**, where we still backtrack to generate the output tree. We show that we can still achieve worst-case optimality to generate all the results with each algorithm. A summary of the implemented algorithms is presented in Figure 1.1.

Another way to report results soon is to use **lazy evaluation**. For some operations in a qdag, we do not need to evaluate all the nodes of the data structure. For example, for computing the `AND` between an empty quadtree and another quadtree, we do not need to compute the second one to return the result (an empty quadtree). Then, we are interested in studying a lazy version of the qdags, called *lqags*, that represents the output quadtree as a formula using a syntax tree and which computes only the needed nodes. In this part, we will not use a compact form to represent the formula; instead, we use pointers to represent it and to represent the output (a traditional quadtree).

Lqdags also permit extending qdags to all relational algebra operations. We implement and evaluate this data structure in **the second part of this thesis**. While lqdags could worsen throughput, as discussed in Chapter 4, they improve the time to obtain the first results and extend qdags to the more general relational algebra. Still, it is possible to achieve worst-case optimality for the join operation, but it is not guaranteed for all relational algebra [6].

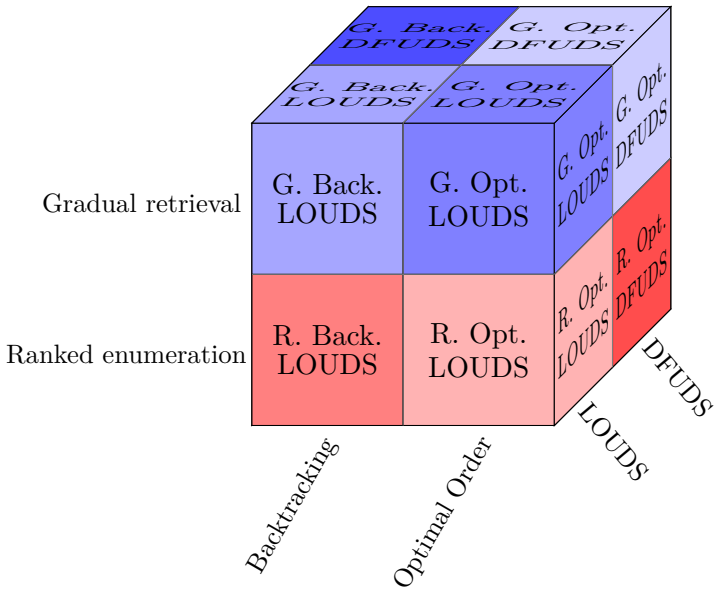


Figure 1.1: Summary of all the implemented algorithms for gradual retrieval and ranked enumeration.

The main goal of this thesis is to study and develop new versions of a compact data structure (qdags) that extends the basic multi-way join algorithm towards gradual retrieval and ranked enumeration, as well as to the full relational algebra. In general, we extend the use of qdags and show that it is possible to obtain partial and ranked results traversing fewer nodes than the original algorithm. We also show that it is possible to extend qdags to evaluate all the relational algebra.

Organization of the thesis. In Chapter 2, we will define the basic concepts and data structures used in this work, such as the qdags, LOUDS, DFUDS, and worst-case optimality. We will explain the algorithms to obtain partial and ranked results and the different techniques used to achieve them in Chapter 3. In Chapter 4, we will show how lazy qdags work, their implementation, and how to obtain full relational algebra formulas. We will discuss and compare the results of each algorithm in Chapter 5, to finally conclude in Chapter 6. The experiments and results are shown at the end of each chapter. The code with all the implementations will remain in a public GitHub repository (www.github.com/asugomez/qdags).

Chapter 2

Preliminaries

We introduce some data structures and concepts to better understand the problem and algorithms of this thesis.

2.1 Morton code

The Morton or Z-ordering [38] is a method to traverse the quadrants of a grid in a specific order: bottom-left, and bottom-right, top-left, top-right. This method is useful for binary encoding the path from the root to a node in a quadtree, where the first child corresponds to `00`, the second to `01`, and so on. The Morton code is a bit-string that represents the coordinates of a node in a quadtree, created by interleaving the bits of the node's y and x coordinates [18].

Figure 2.1 illustrates how to obtain the coordinates of a node using the Morton code.

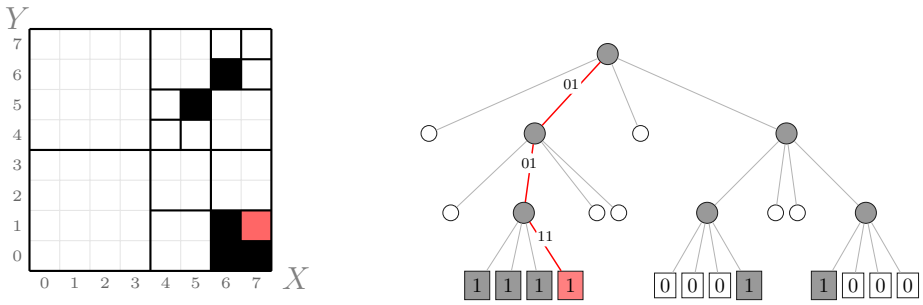


Figure 2.1: An illustration of the coordinates of a leaf using Morton code. *Concatenating the Morton code of each node of the path from the root to the red leaf, we can build the bit-string '010111'. This concatenation encodes the coordinates of* (x, y)*. For* y*, we have to look for the odd positions of the aforementioned bit-string; for* x*, we must look for the even positions. Therefore,* $y = 001$ *and* $x = 111$*, that is* $(x, y) = (7, 1)$.

2.2 K^2-tree

A bitvector is an array of bits $B[1,n]$ that supports the operations ACCESS(B,i) (the i-th bit of B), RANK$_v$(B,i) (the number of occurrences of bit v in $B[1,i]$) and SELECT$_v$(B,i) (the position of the i-th v in B). This data structure is the core of many compact data structures, such as the k^2-tree.

A succint representation encodes B in $n + o(n)$ bits, while supporting the operations in constant time [16, 34, 39] [41, Chapter 4]. We can achieve lower space for very sparse bitvectors [44] using $m \log \frac{n}{m} + O(m)$ bits (with m the number of 1s and $m \ll n$) and supporting SELECT in constant time and ACCESS and RANK in $\mathcal{O}(\min(\log m, \log \frac{n}{m}))$ time. Therefore, the space is proportional to m, the number of 1s, and not to the bitvector size.

A k^2-tree [15] is a compact quadtree representation (in the case $k = 2$). It saves the data of a binary matrix $M_{l \times l}$ by dividing it recursively into k^2 quadrants of the same area. The nonempty quadrants are recursively subdivided. If the matrix side is not a power of k, the matrix is completed with 0s. The height of a k^2-tree is $h = \lceil \log_k l \rceil$.

The k^2-tree is represented as a compact cardinal tree [15]. We compare two data structures to represent trees in a compact form: the Level-Order Unary Degree Sequence (LOUDS) and the Depth-First Unary Degree Sequence (DFUDS). LOUDS [34] represents each level of the cardinal tree using only a bitvector: we traverse the tree level by level, and each node is represented by its children using k^2 bits, where a 0 indicates an empty quadrant and a 1 means that points exist in that quadrant. These bits are then concatenated for all nodes across all levels, creating a compact representation.

Example 1 (LOUDS)**.** In a quadtree ($k = 2$), a leaf will be represented with four zeros 0000, a node with four children will be 1111, a node with only its first child will be 1000, and so on. A zero child means that there are no points within this quadrant. Thus, the empty areas will be captured in one node with only a 0, and the last level of the tree, that is full of leaves, is represented with the number of leaves times 0000. We can see an example of a matrix in Figure 2.2, represented by the quadtree in Figure 2.3. The LOUDS representation of this tree is shown in Figure 2.4. We can notice that the last level of the tree is represented with only 0s.

A LOUDS representation for cardinal trees with n nodes will use $k^2 n + o(n)$ bits and support RANK and SELECT in constant time [41, Chapter 8]. With these two essential operations, this representation supports in constant time the operations of Table 2.1. We can traverse the tree using the basic operations in bitvectors: ACCESS, RANK, and SELECT. For example, to find the t-th child of the node v, we use the algorithm in Figure 1.

A particularity of the k^2-tree representation is that, because we already know the height of the tree ($h = \lceil \log_k l \rceil$), we do not need to store the leaves at the last level (only 0s) because we already know there are only leaves at this depth. The k^2-tree representation using LOUDS (see Figure 2.3) is the same as in Figure 2.4, but without the last level of 0s. Therefore, we only need $k^2 n + o(n)$ bits, n being the number of **internal** nodes.

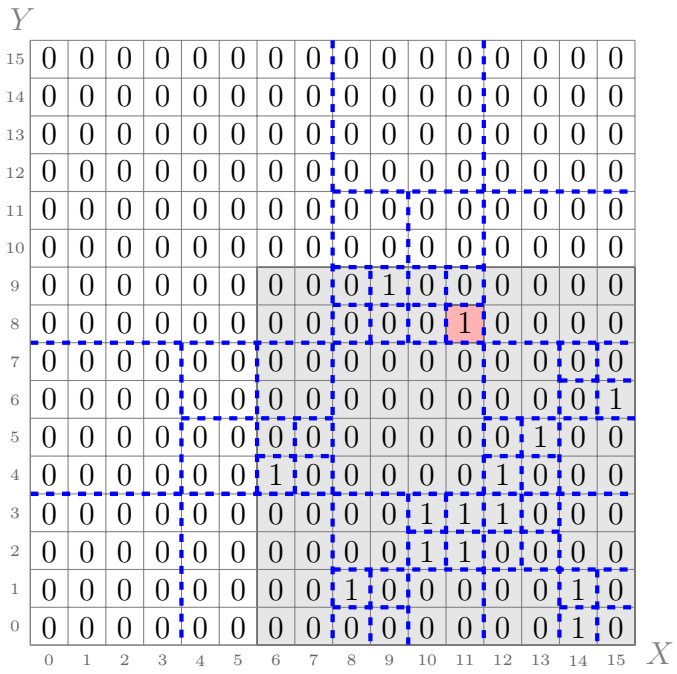


Figure 2.2: Example of a 16×16 matrix, with 14 points. *The gray area is where the matrix points are located. In red, the point (11,8).*

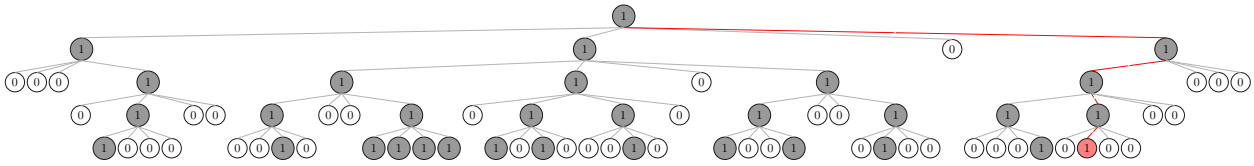


Figure 2.3: Quadtree of the matrix of Figure 2.2. *We traverse the quadrants in Morton order. In red, the path to the point (11,8).*

B = 1101
0001 1101 1000
0100 1001 0110 0110 1001 1100
1000 0010 1111 1010 0010 1001 0100 0001 0100
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

Figure 2.4: LOUDS representation of the cardinal tree of Figure 2.3. *In red, the description of the nodes that represents the path toward the point (11,8). Notice that in the example we separate the bits for each level in LOUDS, but in the actual representation, all levels are concatenated into a single bitvector.*

Using the compressed representation of very sparse bitvectors mentioned before, it is possible to reach the worst-case entropy of k^2-trees [41, Chapter 10]. The entropy is a lower bound of the number of bits needed to represent an element. From information theory, we know that we need at least $\log_2 |U|$ bits to distinguish one element from another in a universe U. This is called the wort-case entropy of U, H_{wc}.

When analyzing the worst-case entropy of k^2-trees with n nodes, their number is equivalent to the number of ways to place n points within an $l \times l$ matrix, which can be represented by

Algorithm 1 Computing an ordinal tree operation using LOUDS.

Require: Ordinal tree T (seen as LOUDS bitvector B), node v (a position in B), and indexes i and t.

1: **procedure** CHILD(T, v, t)
2: **return** $\text{select}_0(B, \text{rank}_1(B, v - 1 + t)) + 1$

Operation	Meaning
CHILDREN(v)	number of children of the node v
PARENT(v)	parent of the node v
FIRSTCHILD(v)	the first child of the node v
LASTCHILD(v)	the last child of the node v
TCHILD(v)	the t-th child of the node v
NEXTSIBLING(v)	the next sibling of v
PREVIOUSSIBLING(v)	the previous sibling of v
LEAFNUM(v)	the number of leaves in the subtree of node v
LEAFRANGE(v)	the positions of the first and last leaf of the subtree of node v

Table 2.1: Summary of some of the operations supported in LOUDS.

the binomial coefficient [41, Chapter 9] :

$$|T_n^{k^2}| = \binom{l^2}{n}$$

Thus, the worst-case entropy is:

$$H_{wc}(T_n^{k^2}) = \log_2 \binom{l^2}{n} = n \log_2 \frac{l^2}{n} + \mathcal{O}(n).$$

If we use the plain form to represent the k^2-tree, we need:

$$n \left(\frac{k^2}{k^2 - 1} + k^2 \left(1 + \log_{k^2} \frac{l^2}{n} \right) \right)$$

bits. And using very sparse bitvectors [41, Chapter 4], we need:

$$\mathcal{O}\left(n \log \frac{l^2}{n} + n \log k \right)$$

bits, where $n \log k$ is redundant regarding the worst-case entropy. The DFUDS [12] is a more complex data structure, which is defined in the following subsection.

2.3 Parentheses

A balanced sequence of parentheses [40] is represented as a bitvector $B[1, n]$ where `1` represents an open parenthesis '(' and `0` a closed parenthesis ')'. There are $n/2$ matching pairs of parentheses, so each '(' can be associated with a ')' and vice versa. This structure supports operations such as CLOSE(B, i), which returns the position of the matching closing parenthesis of $B[i]$ ='(' ; OPEN(B, i), which returns the position of the opening parenthesis corresponding to $B[i]$ =')' ; and ENCLOSE(B, i), that returns the rightmost position $k < i$ such that $[k, \text{CLOSE}(B, k)]$ contains i. We can use this data structure to represent a tree: we visit the tree in depth-first search order, and the first time we visit a node, we put a '(' , and once we leave the subtree, we put a ')' [41, Chapter 8].

DFUDS [12] is another way to represent a k^2-tree using only $k^2 + 2 + o(1)$ bits per node. The structure is based on parentheses (so it supports its operations) and consists of two bitvectors, $B[1, 2n + 2]$ and $S[1, k^2 n]$, with n the number of nodes. We traverse the tree in preorder and save in B the description of each node: $\texttt{1}^c\texttt{0}$, where c is the number of children. The second bitvector S contains k^2 bits per node. It stores information about whether each child of the node exists or not, similarly to the representation used in LOUDS, but in preorder [41, Chapter 8]. We support RANK, SELECT, and all the parentheses operations in bitvector B. Figure 2.5 shows an example of this data structure.

B =	110	1110	10	10	10	0
			1110	110	10	0
					11110	0000
				110	110	00
					10	0
				110	110	00
					10	0
			10	110	10	0
					10	0

S =	1101	0001	0100	1000	0000			
		1101	1001	0010	0000			
				1111	0000	0000	0000	0000
			0110	1010	0000	0000		
				0010	0000			
			1001	1001	0000	0000		
				0100	0000			
		1000	1100	0001	0000			
				0100	0000			

Figure 2.5: DFUDS representation of the cardinal tree of Figure 2.3. *In red, the description of the nodes that represents the path toward the point (11,8). Note that in bitvector B (parentheses) we add `110` at the beginning to handle some border cases [41, Chapter 8].*

We are interested in this data structure because we can count the leaves descending from a node in constant time [41, Chapter 8], as we will see in Section 3.2.

S =	1101	0001	0100	1000
		1101	1001	0010
				1111
			0110	1010
				0010
			1001	1001
				0100
		1000	1100	0001
				0100

Figure 2.6: New version of bitvector S from the DFUDS representation of the cardinal tree of Figure 2.3.

Contribution: new version of DFUDS As we can see in Figure 2.5, to properly map from B to S, we have to store in S the description of each leaf (`0000`). We can save space by omitting the description of those leaves. To compute the node in the bitvector S that corresponds to $B[i]$, we now use $\text{RANK}_{10}(i)$, which counts the occurrences of `10` in $B[1, i]$ (i.e, counts the internal nodes up to i). Then, S will use $k^2 n$ bits, with n the number of **internal** nodes, just like the k^2-tree representation using LOUDS. In Figure 2.6 we can see the new version of S. In Algorithm 2 we show how to compute the t-th child of a node v using this new DFUDS representation.

Algorithm 2 Computing the child DFUDS operation on ordinal trees.

Require: Ordinal tree T (seen as DFUDS bitvector B), node v (a position in B), and index t.

1: **procedure** CHILD(T, v, t)
2: **return** close(B, $\text{succ}_0(B, v) - t$) $+1$

2.3.1 Range Maximum Query

The Range Maximum Query (rMq) problem is defined as follows: given an array P of n totally ordered elements, build a data structure on P that efficiently returns the position of the highest value within any given range $[i, j]$ of P, where $1 \leq i \leq j \leq n$ [21] [22].

Let P store the importance of the leaves of a cardinal tree, and assume we represent the tree with DFUDS. In DFUDS, given a node, we can determine the range in P corresponding to its descendant leaves, and the rMq on P allows us to find the position of the most important descendant leaf in constant time [41, Chapter 8].

2.4 Graph patterns

In graph databases, we encode the relations as triples (s, p, o), where s (subject) and o (object) are nodes connected through the edge labeled p (predicate). A **basic graph pattern** (BGP) is an expression that consists of a set of triple patterns: two nodes linked by an edge label, where nodes or labels can be identified as variables. The goal is to determine if there is an assignment of values to variables such that the BGP matches in the graph database. Each triple pattern is matched to an edge, but since triple patterns share variables, solving a BGP is equivalent to solving a conjunctive query composed of multiple triple patterns, each triple pattern standing for an atomic query [1, 9, 52].

Figure 2.7 is an example of a graph database inspired by Arroyuelo et al. [4]. We can see an example of a BGP in Figure 2.8.

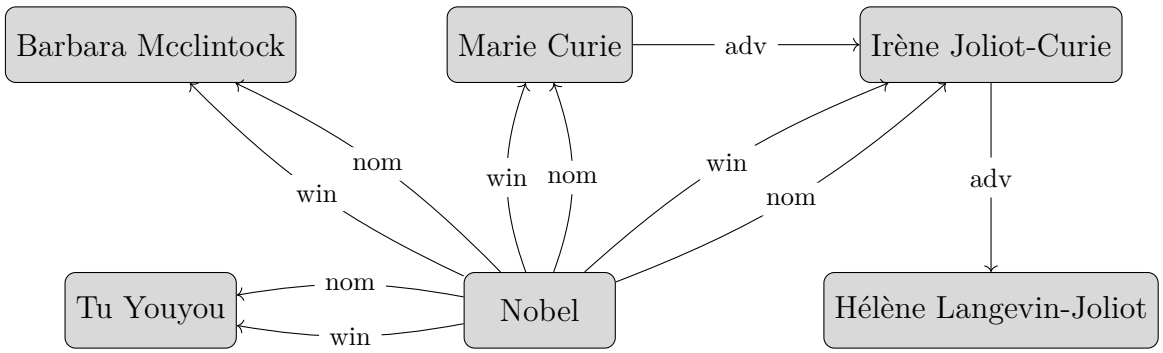
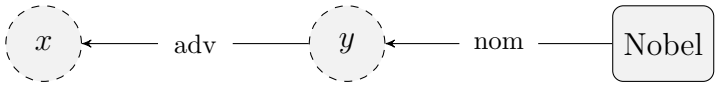


Figure 2.7: Graph of Nobel winners, nominees and advisors.



(a)

x	y
Irène Joliot-Curie	Marie Curie
Hélène Langevin-Joliot	Irène Joliot-Curie

(b)

Figure 2.8: In (a) a basic graph pattern (BGP) $\{(\text{Nobel}, \text{nom}, y), (y, \text{adv}, x)\}$. In (b) the results of the evaluation of the BGP over the graph of Figure 2.7. *This evaluation returns all the people who were Nobel nominees and their students.*

A hypergraph $H = (V; E)$ is a generalization of graphs, where V is a set of vertices and E is a set of hyperedges. A hyperedge can connect more than two vertices, unlike edges in standard graphs. If there is a path P from vertex u to vertex w, denoted as, $P_{\{u,w\}} = (u = v_1, e_1, v_2, ..., e_t, v_{t+1} = w)$, where $\{v_i, v_{i+1}\} \subseteq e_i \; \forall i$, we say that P *connects* u and w. If $u = w$, the path forms a hypercycle [14]. In databases, a join query can be represented as a

hypergraph, where each attribute corresponds to a vertex and the relationship corresponds to an edge.

Example 2 (Hypergraph)**.** The join query $R(A, B) \bowtie S(B, C, D) \bowtie T(A, D)$ can be visualized as the hypergraph depicted in Figure 2.9.

A query is cyclic if its hypergraph, defined by the attributes and relations of the query, contains at least one hypercycle. Otherwise, the query is **acyclic** [24]. In Figure 2.9, we can see an example of a cyclic hypergraph of a join query.

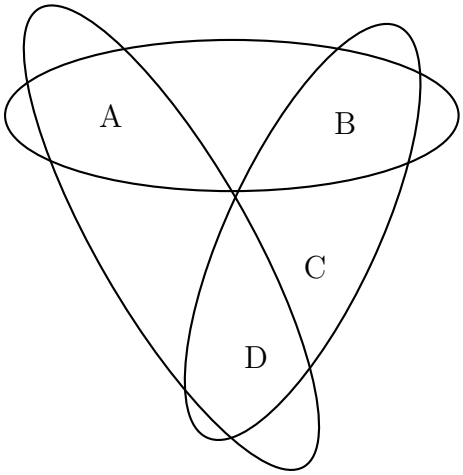


Figure 2.9: Hypergraph of query $R(A, B) \bowtie S(B, C, D) \bowtie T(A, D)$.

2.5 Worst Case Optimal Joins

The join operation is the basis of many queries, especially on graph databases, because graph queries tend to feature many joins [4]. Joins form the core of basic graph pattern queries, as we saw before. Therefore, improving time and space in join operations directly impacts many database queries.

Traditional database engines perform the join using a **pairwise join** algorithm [43], whose cost will depend on the plan to compute which pairs we join first. For example, we can perform the join between the relations $R(A, B)$, $S(B, C)$, $T(C, A)$ as $(R \bowtie S) \bowtie T$, or $R \bowtie (S \bowtie T)$, or also $(R \bowtie T) \bowtie S$, because of its commutative property. Following a pairwise strategy to perform the join is, indeed, *suboptimal* [43]: if the size of S, R, T is N, our query done by a pairwise join algorithm could take $\Omega(N^2)$ time, while its maximum possible output size is $\mathcal{O}(N^{3/2})$. Still, database management systems like Postgres use these sub-optimal algorithms to perform the join.

A recent approach to perform a join is a **multi-way join** algorithm, meaning to join several tables simultaneously. This could reduce intermediate results and, therefore, time and space. Worst-case optimal multi-way join algorithms have demonstrated high performance

on complex queries [35]. In particular, they can solve the given join ($R \bowtie S \bowtie T$) in the $\mathcal{O}(N^{3/2})$ optimal time.

We are interested in this type of algorithm due to its performance, especially on cyclic queries. If the query is acyclic, Yanakakis' algorithm introduces semi-joins to eliminate unnecessary tuples, and it produces the output in time linear in the size of the input and the output [54]. However, we need other algorithms for cyclic queries such as Leapfrog Trie Join [51] or Tetris [36].

A **worst-case optimal algorithm** (wco) means that the algorithm is optimal in the worst case: there exists an instance of the problem, with the same input size, that generates a result whose size is proportional to the time the algorithm takes. In other words, "(...) the algorithm's runtime is bounded by the worst-case cardinality of the query result." [32]. We can prove an algorithm is wco if it satisfies the **AGM bound** [8], which takes into account the structure (graph) and the size of tables. That is, it considers both types of information [43], and defines a bound according to these variables. The worst-case optimality concept was proposed for cyclic queries because we cannot design an instance optimal algorithm, i.e., we cannot compute its join in $\mathcal{O}(\text{IN} + \text{OUT})$ time, where IN is the input size, and OUT corresponds to the output size [33]. Worst-case optimal joins are a promising advance, but they can require a lot of space [36, 51].

2.6 Compressed quadtrees

Some multi-way join algorithms, such as LeapFrog Trie Join [51], or Tetris [36], are indeed worst-case optimal, but they need a heavy index data structure to perform the join. Arroyuelo et al. [6] proposed the first wco multi-way join algorithm using a compact data structure instead of the heavy index data structures used by the other algorithms of the time. It was the first algorithm that focused on time and space.

A tuple of the relation $R(\mathcal{A})$ with d attributes $A_1, A_2, ..., A_d$ over domain $[1...l]$ can be represented as a point in a d-dimensional grid of size l^d. We use a d-dimensional k^2-tree called a k^d-tree to represent this grid of l^d cells, where each child represents a subgrid of size $(l/k)^d$. Moreover, we use $k = 2$, so each node will be described with 2^d bits. From now on, we refer to this data structure as a quadtree.

To perform a join between multiple relations, we must do two operations: EXTEND and AND. The first one "(...) lifts the quadtree representation of a grid to a higher-dimensional grid" [6], and the second one computes the intersection of the *qdags*. The wco join algorithm is based on this extended version of a quadtree (k^d-tree), called *qdag*. In this representation, the k^d-tree is stored in a compact form using LOUDS. The quadtree is extended to a qdag by lifting its dimensionality from d' to a higher dimension d to represent a relation of d attributes, incorporating all the attributes needed for the join. This extension involves creating new values for nodes corresponding to the additional $d - d'$ dimensions to maintain consistency across the joined relations. We can see an illustration of this in Figure 2.10.

Example 3. Let $R'(A, B)$, $S'(B, C)$, and $T'(A, C)$ be relations, and let $R(A, B, C)$, $S(B, C, A)$, and $T(A, C, B)$ be the qdags obtained by extending the relations R', S', and T' to the

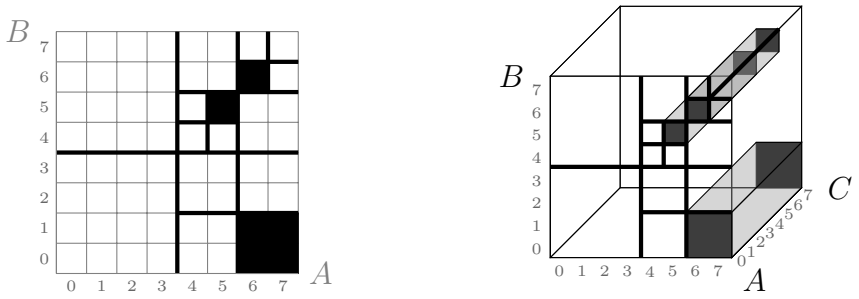


Figure 2.10: Extend operation of $R'(A, B)$ to $R(A, B, C)$. *We can see the grid of a relation $R'(A, B)$ and a 3-dimensional grid, which results from extending the relation $R'(A, B)$ to the attribute C. These grids are represented by a k^d-tree as shown in Figure 2.14.*

attribute C, A and B, respectively. To perform a join between these relations, we first extend the relations as shown in Figures 2.10 and 2.11. Then, we intersect the qdags R, S, and T using the AND operation, as shown in Figure 2.12. The result of the join is a quadtree represented in Figure 2.13.

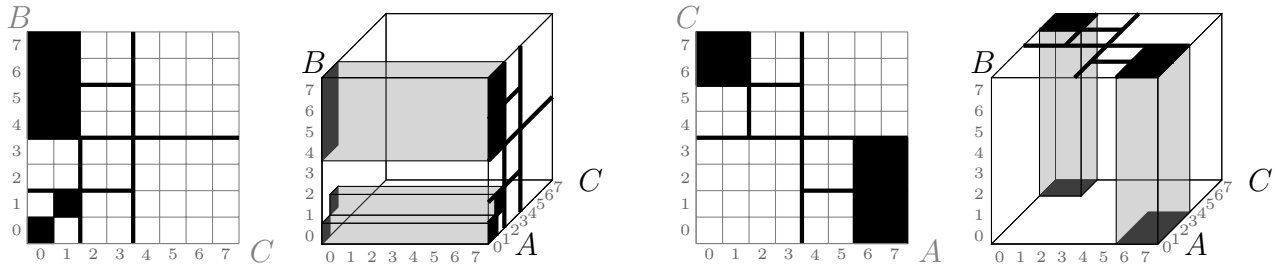


Figure 2.11: Extending relations $S'(B, C)$ and $T'(A, C)$ to $S(B, C, A)$ and $T(A, C, B)$.

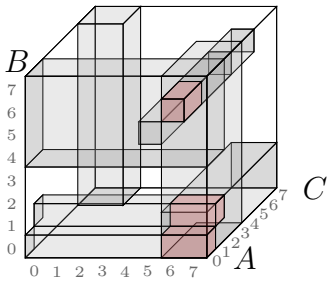


Figure 2.12: Intersection of relations $R(A, B, C)$, $S(B, C, A)$ and $T(A, C, B)$.

Definition 1 (EXTEND)**.** Consider $R(\mathcal{A}')$, a relation defined over the attributes $\mathcal{A}'$, and let $\mathbf{Q}_R = (Q', M')$ be a qdag that represents $R(\mathcal{A}')$. Operation EXTEND$(\mathbf{Q}_R, \mathcal{A})$ generates a qdag $\mathbf{Q}^*_R$ that represents the relation $R(\mathcal{A}')\times\text{All}(\mathcal{A} \setminus \mathcal{A}')$, where $\mathcal{A}' \subseteq \mathcal{A}$ and $\text{All}(\mathcal{A} \setminus \mathcal{A}')$ is a relation with the attributes $\mathcal{A} \setminus \mathcal{A}'$ storing all the possible elements. It is supported in $\mathcal{O}(2^d)$ time, and its output takes $\mathcal{O}(2^d)$ words of space.

Although the qdag has a higher dimension than the quadtree, many nodes share subtrees. Instead of representing the higher-dimensional quadtree explicitely, qdags use a mapping

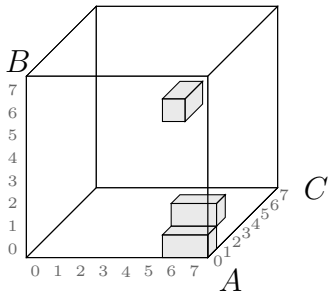


Figure 2.13: Final output of $R'(A,B) \bowtie S'(B,C) \bowtie T'(A,C)$, that is, the points remaining in the intersection.

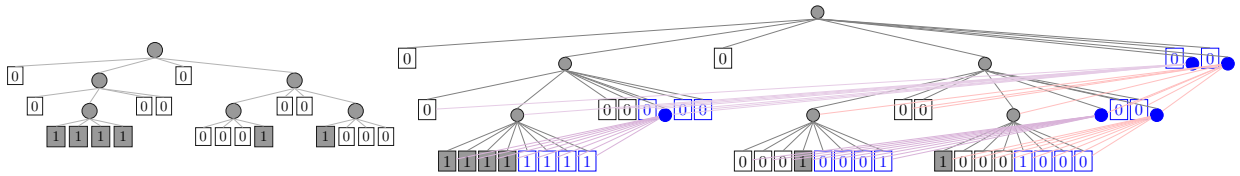
Algorithm 3 EXTEND $(\mathbf{Q}, \mathcal{A})$

Require: A qdag $\mathbf{Q} = (Q', M')$ representing a relation $R(\mathcal{A}')$, and a set $\mathcal{A}$ such that $\mathcal{A}' \subseteq \mathcal{A}$.
Ensure: A qdag $\mathbf{Q} = (Q', M)$ whose materialization 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']
return (Q', M)
```

function $M : [0, 2^d - 1] \to [0, 2^{d'} - 1]$ to map the nodes from the extended quadtree to the original one.

Formally, a d-dimensional qdag $\mathbf{Q}$ is a pair $(Q',\ M)$, where Q' is a d'-dimensional quadtree (with $d' \leq d$) and M is a mapping function. The i-th child of a node in Q corresponds to the $M[i]$-th child of the corresponding node in Q'. Thus, the qdag simulates a d-dimensional quadtree, and requires only $2^{d'}n + o(n)$ bits, where n is the number of internal nodes of the quadtree. In Figure 2.14, we can see an example of a quadtree and its extension to a qdag.



$M' = [0, 1, 2, 3]$ $\qquad$ $M = [0, 1, 2, 3, 0, 1, 2, 3]$

Figure 2.14: Qdag of $R(A, B, C)$. *On the left, we can see a quadtree of a relation $R'(A, B)$, and on the right, the virtual qdag. The grey nodes represent `1`s, and the white nodes represent `0`s. We order the nodes using the Morton [38] partitioning. We can also see many nodes sharing subtrees on the second tree, so we only need to store the first quadtree and a mapping function to simulate the second k^2-tree. The mapping function is shown at the bottom of the k^2-tree, and the nodes in blue are not materialized.*

Algorithms 4 and 5 let us navigate through the qdag and simulate the implicit representation of the d-dimensional quadtree Q. The VALUE operation returns 0 if the subquadrant that is

represented by this root is empty; 1 if the grid is a full single cell, and 1/2 otherwise (when it is an internal node). Operation CHILD accesses the i-th child of the node through the mapping function.

Algorithm 4 VALUE $(\mathbf{Q})$

Require: A qdag $\mathbf{Q} = (Q', M)$ 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.
1: **if** $l = 1$ **then return** the integer Q'
2: **if** Q' is a leaf **then return** 0
3: **return** $\frac{1}{2}$

Algorithm 5 CHILD $(\mathbf{Q}, i)$

Require: A qdag $\mathbf{Q} = (Q', M)$ 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 $\mathbf{Q}_i = (Q'', M)$ corresponding to the i-th child of $\mathbf{Q}$
1: **return** $(Q[M(i)], M)$

In Algorithm 3, we show how to build the qdag for each relation. To efficiently implement line 5, we build a table that depends on the number of attributes $|\mathcal{A}| = d$ and tells us which child of the d'-dimensional quadtree to access to navigate through the qdag. This table requires $\mathcal{O}(2^{2d})$ space and computes $m_{d'}$ in constant time. In Figure 2.10 we can see the representation of the k^d-trees of Figure 2.14 in a d'- and d-dimensional grid.

Once we extend all the relations to have the same attributes, we perform the AND operation, which computes the intersection of the qdags. To compute the AND of all qdags, we backtrack on the qdags recursively until we find an empty quadrant in one of them or a leaf. The result is another compressed quadtree, and the points within the structure result from the join [6]. The pseudo-code is shown in Algorithm 6.

Definition 2 (AND)**.** Let $\mathbf{Q}_1^*, ..., \mathbf{Q}_n^*$ be qdags representing the relations $R_1(\mathcal{A}_1), ..., R_n(\mathcal{A}_n)$. The operation AND$(\mathbf{Q}_1^*, ..., \mathbf{Q}_n^*)$ computes the intersection of the qdags, that is, a quadtree that represents the relation $\cap_{i=1}^n R_i(\mathcal{A}_i)$.

Algorithm 6 AND$(\mathbf{Q}_1, ..., \mathbf{Q}_n)$

Require: n qdags $\mathbf{Q}_1, ..., \mathbf{Q}_n$ representing relations $R_1(\mathcal{A}), ..., R_n(\mathcal{A})$.
Ensure: A quadtree representing the relation $\cap_{i=1}^n R_i(\mathcal{A})$.
1: $m \leftarrow \min\{$VALUE$(\mathbf{Q}_1)$,...,VALUE$(\mathbf{Q}_n)\}$
2: **if** $l = 1$ **then return** the integer m
3: **if** $m = 0$ **then return** a leaf
4: **for** $i \leftarrow 0, ..., 2^d - 1$ **do**
5: $\quad C_i \leftarrow$ AND(CHILD$(\mathbf{Q}_1, i)$,..., CHILD$(\mathbf{Q}_n, i)$)
6: **if** $\max\{$VALUE(C_1),...,VALUE(C_{2^d-1}) $\} = 0$ **then return** a leaf
7: **return** a quadtree with children $C_0, ..., C_{2^d-1}$

Algorithm 7 MULTIJOIN $(R_1, ..., R_n)$

Require: Relations $R_1, ..., R_n$, stored as quadtrees $Q_1, ..., Q_n$; each relation R_i is over attributes $\mathcal{A}_i$ and $\mathcal{A} = \bigcup \mathcal{A}_i$.
Ensure: A quadtree representing the output $J = R_1 \bowtie ... \bowtie R_n$.
1: **for** $i \leftarrow 0, ..., n$ **do**
2: $\quad$ Let $\mathbf{Q}_i$ be the qdag $(Q_i, \mathrm{Id}(\mathcal{A}_i))$
3: $\quad \mathbf{Q}_i^* \leftarrow$ EXTEND$(\mathcal{Q}_i, \mathcal{A})$
4: **return** AND$(\mathbf{Q}_1^*, ..., \mathbf{Q}_n^*)$

To analyze the cost of the AND operation, let n be the number of qdags we want to intersect, and h the height of the materializations of $\mathbf{Q}_1, ..., \mathbf{Q}_n$.

Definition 3 (Materialization of a qdag)**.** Let $\mathbf{Q} = (Q', M)$ be a d-dimensional qdag, where Q' is a d'-dimensional quadtree. The materialization $\mathbf{Q}^*$ of $\mathbf{Q}$ is the explicit representation of the d-dimensional quadtree that $\mathbf{Q}$ simulates.

In Algorithm 6, the cost of AND is bounded by the number of CHILD operations we need to call. As a result, the cost is $\mathcal{O}(2^d \cdot (||\mathbf{Q}_1|| + ... + ||\mathbf{Q}_n||)) = \mathcal{O}(2^d n \cdot |Q^+|)$, where $||\mathbf{Q}_i||$ is the number of internal nodes in the materialization of $\mathbf{Q}_i$; and Q^+ is the non-pruned version of Q (the resulting quadtree of Algorithm 6 if we remove the pruning step of line 6). This can be bounded by $\mathcal{O}(m \cdot 2^d n \cdot h)$, where m is the maximum number of internal nodes at any level of the final quadtree.

Worst-case optimality As we can see, the cost of the join (see Algorithm 7) is dominated by the cost of the AND operation. Let $J = R_1 \bowtie ... \bowtie R_n$ be a full join query and use $2^{\rho^*(J,D)}$ to define the AGM bound of the join J over the database D with d different attributes over the domain $[0, l-1]$. Let us define N as the total number of tuples in the database and S as the total number of tuple components. The output of the query can be computed in

$$\mathcal{O}(2^{\rho^*(J,D)} \cdot 2^d n \log \min(l, S)) = \tilde{\mathrm{O}}(2^{\rho^*(J,D)})$$

time, because $m = \mathcal{O}(2^{\rho^*(J,D)})$, and using only $S \log l + 2N \log l + o(S \log l) + \mathcal{O}(n \log d)$ bits.[1]

Furthermore, the concept of qdags can be extended to handle not only joins and intersections but also unions and negations in worst-case optimal time. We can achieve this with a lazy version of qdags, which generalizes the principle of processing the arguments only as needed to construct the output. While this approach allows for performing all relational algebra operations, it does not guarantee worst-case optimal times. We will discuss this in more detail in Chapter 4.

2.7 Graph Patterns and experimental results

For each of our algorithms, we are going to study a set of 17 query patterns (see Figure 2.15) from the Wikidata Graph Pattern Benchmark, proposed by Hogan et al. [32]. This is the same set that the original algorithm used to experiment on its performance [6]. This benchmark provides different query patterns to test acyclic and cyclic queries of different widths and shapes.

We will make 50 iterations for each query pattern, and for each iteration, we will use the same set of predicates used in [6] from the WikiData, presented as a binary relation. We will test all our algorithms with the same set of queries and predicates. We will also focus on tests that produced more than 1000 results. Finally, we will compare the results in terms of time and space.

[1] For the full proof, see the original paper [6].

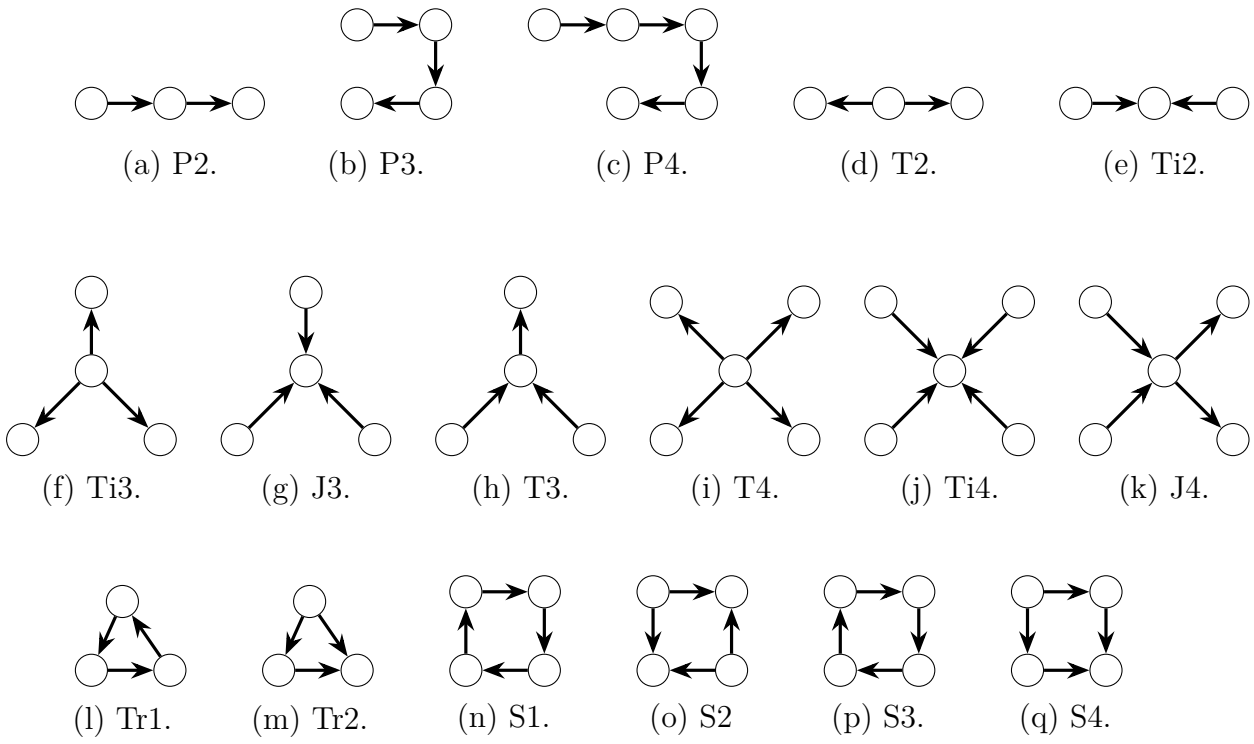


Figure 2.15: Query patterns for the Wikidata Graph Pattern.

Chapter 3

Gradual retrieval and ranked enumeration

3.1 Motivation

In the previous chapter, we saw we have to compute the intersection between all *qdags* to perform a join. Then, we can output the join result once we finally compute all the leaves. This method works efficiently when we need all the results or a significant part of them. However, in some cases, we will be more interested in some of the results. In many cases, such as in user interfaces, finding some results quickly or gradually retrieving them is more important. Additionally, when we want specific results —such as when applying an operator to limit the number of results, like '`LIMIT`' in SQL, or when sorting results by a particular property— we are more interested in finding the most relevant results first [50].

If we use the previous algorithm for gradual retrieval, we will need to stop the computation once we have found the first k results. This approach is not optimal, so we will study new algorithms to perform it more efficiently. For ranked enumeration, using the previous algorithm, we would need to compute all the results first and then order them according to the property of interest, adding a complexity of $\Omega(N)$, where N is the size of the output. Computing the full join merely to retrieve a subset of the results is wasteful. Therefore, we are interested in exploring better methods for obtaining some results as fast as possible (gradual retrieval) and prioritizing the most important results (ranked enumeration) using qdags.

This section aims to modify the original multi-way join algorithm to traverse the k^2-trees in a specific order to compute either some results faster or the important results first. On the one hand, for **gradual retrieval**, our goal is to perform the join operation to obtain partial results quickly, optimizing the time to retrieve the initial results. If we are interested in the first k results, we aim to output any result as fast as possible.

On the other hand, the main objective for **ranked enumeration** is to return the top-k most important results for k given at query time. This means we want to output the most relevant results first, followed by the less relevant ones.

As with the original join, we aim to extend each relation (see Algorithm 3), and then modify the AND operation (see Algorithm 6) to perform it in a particular order. This variation must allow either quickly returning partial results or considering the node's priority to output ranked results while still ensuring worst-case optimality in traversing all the results. Unlike the original algorithm, we do not construct the quadtree but directly report the coordinates using their Morton code [38].

First, we implement two cardinal tree representations: LOUDS and DFUDS. We compare the time and space complexity of both implementations. For the LOUDS tree, we use the sdsl-lite library [23]. We also use this library to implement the base data structure, parentheses (see Section 2.3), and its operations to build the DFUDS tree.

Second, we implement two different algorithms to perform the join. The first algorithm uses a priority queue of non-fixed size to maintain an **optimal order** of the tuples of nodes that can be visited next. The second algorithm uses an array (for gradual retrieval) or a fixed-size priority queue (for ranked enumeration) and employs **backtracking** to store only the top k results seen so far. Once we have the top k results, we report the points.

We use the same dataset in which the original join algorithm was tested (see Section 2.7) and employ the same queries to evaluate the performance of the new algorithms.

3.2 LOUDS and DFUDS

As we saw in Chapter 2, the main difference between LOUDS and DFUDS for k^d-trees is that in the first one, we only have to store one bitvector with the node's description. In contrast, in the second one, we have to store two bitvectors: one with the number of children and the other with the node's description. Another difference is that in LOUDS, we traverse the tree in BFS, while in DFUDS, we store it in DFS. In both representations, we can access a node's child, parent, or sibling in constant time.

As we will see in Sections 3.4 and 3.5, we need to compute the number of leaves for gradual retrieval or a range of leaves for ranked enumeration in qdags. To compute the number of leaves of a node, using **LOUDS**, we descend recursively by the first node with a child until we find a leaf, and we do the same for the last node. Then, we count the `1`s within this range to obtain the number of leaves of the subtree. For this, we have to perform a `rank` on each level, which gives a total complexity of $\mathcal{O}(1) \cdot \log l = \mathcal{O}(\log l)$, where l is the side of the grid.

Therefore, it takes $\mathcal{O}(\log l)$ time to compute the leaves number in LOUDS, while it takes $\mathcal{O}(1)$ for DFUDS. The pseudo-code is shown in Algorithm 8. We do the same to retrieve the range of leaves, but instead of returning the number of leaves, we return the positions of the first and last child: we must modify line 10 of Algorithm 8 to return RANK(level,fLeaf) and RANK(level,lLeaf).

Although, in theory, we expect DFUDS to perform better, LOUDS could also be highly efficient because it only requires storing one bitvector. This becomes particularly advantageous when managing large amounts of data, where accessing different data structures decreases

locality of reference.

Operation	LOUDS	DFUDS
Children	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Parent	$\mathcal{O}(1)$	$\mathcal{O}(1)$
FirstChild	$\mathcal{O}(1)$	$\mathcal{O}(1)$
LastChild	$\mathcal{O}(1)$	$\mathcal{O}(1)$
TChild	$\mathcal{O}(1)$	$\mathcal{O}(1)$
NextSibling	$\mathcal{O}(1)$	$\mathcal{O}(1)$
PreviousSibling	$\mathcal{O}(1)$	$\mathcal{O}(1)$
leafNum	$\mathcal{O}(\log l)$	$\mathcal{O}(1)$
leafRange	$\mathcal{O}(\log l)$	$\mathcal{O}(1)$

Table 3.1: Time complexities for various operations in LOUDS and DFUDS.

Using **DFUDS** to represent the k^2-tree allows us to compute the number of leaves of a node in constant time using the operations close and rank_{00}[1], yet at the expense of requiring more space to save this data structure (see Section 2.3). We can retrieve in constant time the number of leaves up to a specific position i of the bitvector B ($\text{rank}_{00}(B, i)$). Then, we can compute the difference between the next sibling and the node to determine the number of leaves in $\mathcal{O}(1)$ time. The pseudo-code is shown in Algorithm 9. To compute the range of leaves, we have to return the positions of a node's first and last leaf. Similarly, we return the positions computed to calculate the number of leaves. In Algorithm 9, we have to modify line 4 to return leafNode and leafSibling.

Algorithm 8 leafNum ($\mathbf{Q}$, level, i) for LOUDS

Require: a qdag $\mathbf{Q}$, the level of the node, and the node i. Assumes the node exists and it is not an empty quadrant.

Ensure: the number of leaves in the quadrant that represents the node i.

1: **if** node i is a leaf **then return** the bit corresponding the node.
2: level $\leftarrow$ level + 1
3: fLeaf $\leftarrow$ firstChild(i)
4: lLeaf $\leftarrow$ lastChild(i)
5: **while** fLeaf is not a leaf **do**
6: level $\leftarrow$ level + 1
7: fLeaf $\leftarrow$ firstChild(fLeaf)
8: lLeaf $\leftarrow$ lastChild(lLeaf)
9: nChildren $\leftarrow$ rank(l,lLeaf) $-$ rank(l,fLeaf -1)
10: **return** nChildren

[1] rank_{00}(B,i) is the number of occurrences of 00 in $B[1, i]$

Algorithm 9 LEAFNUM $(\mathbf{Q}, i)$ for DFUDS

Require: a qdag $\mathbf{Q}$ and the node i.
Ensure: the number of leaves in the quadrant that represents the node i.
1: nSibling $\leftarrow$ NEXTSIBLING(i)
2: leafSibling $\leftarrow$ RANK$_{00}(B$,nSibling$)$
3: leafNode $\leftarrow$ RANK$_{00}(B, i)$
4: **return** leafSibling $-$ leafNode

3.3 Computing the join

To understand the different strategies for computing the join, we first need to define the concept of *tuple of qdags*.

Definition 4 (Tuple of qdags or nodes)**.** A tuple of qdags or a tuple of nodes refers to a set of nodes, one from each qdag, all of the same level, representing a potential combination of sub-quadrants across the different relations. For example, selecting the root node from each qdag forms a valid tuple.

Definition 5 (Result of the join)**.** We call a result of the join a tuple of nodes at the last level, that is, a tuple of leaves. We can transform this tuple into coordinates (a point) using the Morton code (see Section 2.1).

Whether we are interested in gradual retrieval or ranked enumeration, we need a strategy to compute the join in a particular order. We follow existing work on proximity search [28].

- Optimal order: We maintain a max priority queue with all the tuples of nodes of the output that can be visited next, sorted by their upper-bound estimations. At each step, we obtain the next node from the queue. If it is a leaf, we report it; otherwise, we insert its children in the queue.
- Prioritized backtracking: We still backtrack to generate the output tree, but their predictors give the order in which the nodes' children are visited. For gradual retrieval, we store the first k results in an array, and in ranked enumeration we use a min priority queue for top-k results seen so far.

A priority queue is a data structure that stores N objects, each associated with a specific weight, using $\mathcal{O}(N)$ space. It supports insertions and extractions in $\mathcal{O}(\log N)$ time and finds the maximum or minimum element in constant time. In the optimal order strategy, we use a priority queue to manage the tuples of qdags we need to visit. For the backtracking approach, we implement a priority queue – when performing ranked enumeration– to store the results (see Definition 5). The weight used in the priority queue differs between gradual retrieval and ranked enumeration. The priority of a node and a tuple of qdags are defined as follows:

Definition 6 (Gradual retrieval: priority of a node)**.** The priority of a node is defined either by:

- The number of leaves descending from the node.
- The density of the node: the number of leaves descending from the node divided by the size of its subgrid.

Definition 7 (Ranked enumeration: priority of a node)**.** Each point of a relation R has a weight or priority stored in an array $\mathbf{P} = [\mathbf{p}_1, \mathbf{p}_2, ..., \mathbf{p}_n]$. The priority of a node in a qdag representing the relation R is defined as the highest priority among the points within the quadrant represented by that node.

Definition 8 (Gradual retrieval: priority of a tuple of qdags)**.** The weight of a tuple of qdags is defined by the minimum weight of the nodes of that tuple: $w = \min\{\mathbf{p}_{i_{\mathbf{Q}_1}}, \mathbf{p}_{i_{\mathbf{Q}_2}}..., \mathbf{p}_{i_{\mathbf{Q}_n}}\}$.

Definition 9 (Ranked results: priority of a tuple of qdags)**.** The weight of the tuple of n qdags $\mathbf{Q}_1, ..., \mathbf{Q}_n$ is defined by:

1. The maximum weight of the tuple of nodes: $w = \max\{\mathbf{p}_{i_{\mathbf{Q}_1}}, \mathbf{p}_{i_{\mathbf{Q}_2}}..., \mathbf{p}_{i_{\mathbf{Q}_n}}\}$.
2. The sum of their weights: $w = \sum_{k=1}^{n} \mathbf{p}_{i_{\mathbf{Q}_k}}$.
3. Another monotone function of the weights.

For this thesis, we study the first two alternatives.

In **optimal order**, we first insert a tuple containing the root of all qdags into a priority queue. We then retrieve the tuple with the highest weight, which initially will be just the tuple of the roots. Next, we insert the tuples of its children into the queue, along with their corresponding weights or priorities. When we reach a leaf node, we can return the tuple of nodes as a result. This is a valid output—in gradual retrieval and ranked enumeration—because we traverse the intersection of the qdags in accordance with the upper bounds on the priorities of the tuples.

An advantage of this strategy is that it visits the minimal set of nodes needed to achieve the best results and identifies the output nodes directly in decreasing order of relevance. If a node does not behave as expected, such as having a lower priority than anticipated, the strategy skips traversing its children and instead explores its siblings or other nodes with higher priority. Moreover, when computing ranked results and reaching a leaf node, we recognize it has the highest priority and can output it immediately [2]. A notable disadvantage of this strategy is that it requires additional space to store the entire queue, resulting in increased operational costs when managing it.

Using the **prioritized backtracking** strategy for gradual retrieval is trivial. We use an array to store the results computed and once we have computed k results, we can return them. For ranked enumeration, we still have to check the priorities of the other tuples of qdags to ensure that we have computed the most important nodes. For that, we need to use a priority queue of a fixed size to store the top k results seen so far. When we arrive at a

[2]In gradual retrieval, we are only concerned in output the results as fast as possible, so all the results are valid.

tuple and the queue is complete, we check if the weight is higher than the lowest weight in the priority queue. If this is the case, we start navigating its children. Otherwise, we prune that branch, so we do not access a node with less priority than our k nodes computed so far. Therefore, we abandon a branch if a node's upper bound in the backtracking is no higher than the value of the current k-th result.

If we arrive at a leaf, we insert the tuple in the priority queue if its weight is higher than the stored tuple with lowest weight and remove that one. Once we have traversed all the qdags, we can return the tuples of the priority queue. Unlike in optimal order, the first k points in the queue do not necessarily mean the top k results, so we can not output them directly.

Although the second strategy uses less space than the first one, it could waste some time when we enter a promising subtree for the backtracking, and its leaves do not produce as many results (in gradual retrieval), or there were no nodes with high priorities as we thought (for ranked results).

We implement these two strategies described above for gradual and ranked enumeration. However, we pursue different techniques to compute the weight of a tuple to process the join in a particular order, as we saw at the begining of the section.

3.4 Gradual retrieval

When we want to obtain only a few results from a join operation, such as in user interfaces like Wikidata, gradual retrieval is beneficial. An interface is more user-friendly when results are retrieved gradually, with the initial results being returned as quickly as possible and subsequent results following with minimal delay. Instead of waiting for the entire join operation to complete, we can display the first results to the user promptly. This approach is also advantageous when working with large datasets, as we can stop the join operation once we have obtained a sufficient number of results.

The goal is to optimize the time required to obtain the first results. We will use an upper-bound estimator to prioritize descending through the child nodes that are estimated to yield the most results. This involves descending through the tuple of nodes representing the fullest quadrants of points, where the number of leaves is higher. To implement this strategy, we study two approaches

1. Estimate intersection with the minimum: we compute the number of leaves in the subtree of each child of the current node in each participant qdag. We value the children by the minimum over all the qdags. We traverse the children sorted from highest to lowest value.
2. Assume uniform distribution: this time, the value of a node is its density, given by its number of leaves and the size of its subgrid. We then compute the product of the densities of the children across all the qdags and process the denser nodes first.

Example 4 (Estimate intersection with the minimum)**.** For example, for the first strategy, in Figure 3.1, we will select the root and process its children in the following order:

1. *The tuple of the second child with priority* $\min(16, 13) = 13$.
2. *The tuple of the fourth child with priority* $\min(8, 16) = 8$.
3. *The tuple of the first child with priority* $\min(6, 12) = 6$.
4. *The tuple of the third child with priority* $\min(23, 5) = 5$.

Then, we will process the children of the tuple of the second child and insert them into the priority queue.

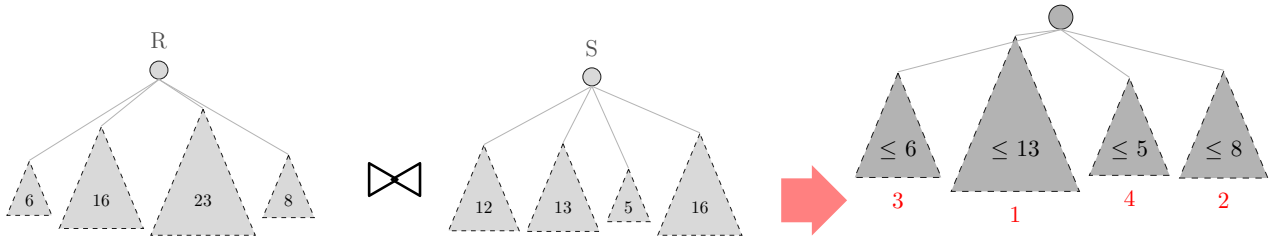


Figure 3.1: Order in partial results (number of leaves). Join between R and S. *We compute the intersection of the i-th child of relation R and the i-th child of relation S. The first child of the qdag of the relation R has six leaves (points), and the first child of the qdag of S has 12 leaves. Thus, the minimum of the tuple of the first children is* $min(6, 12) = 6$. *The tuple of qdags that has the maximum of the minima is the tuple formed by the second children, that is* $min(16, 13) = 13$. *This means that joining this quadrant will produce up to 13 results. We expect that the other quadrants produce fewer results so that the join will start with the tuple of the second children.*

Example 5 (Assume uniform distribution)**.** For example, for the second strategy, in Figure 3.2, we will select the root and process its children in the following order:

1. *The tuple of the fourth child with priority* $(0.16 \times 0.13) = 0.0208$.
2. *The tuple of the third child with priority* $(0.08 \times 0.16) = 0.0128$.
3. *The tuple of the first child with priority* $(0.23 \times 0.05) = 0.0115$.
4. *The tuple of the second child with priority* $(0.06 \times 0.12) = 0.0072$.

Then, we will process the children of the tuple of the forth child and insert them into the priority queue.

For the optimal order approach, we maintain a priority queue of the tuples of nodes to visit and a counter to track the results. We retrieve each tuple from the priority queue and explore its children if it is an internal node, or compute the intersection if it is a tuple of leaves. When we output a result, we update the counter. We stop the operation if we traverse all the internal nodes of the qdags or reach the maximum number of results. The pseudo-code for gradual retrieval using an optimal order strategy and the DFUDS representation is presented in Algorithms 10 and 11. In line 2 of Algorithm 11, $\mathrm{Id}(\mathcal{A})$ corresponds to the identity mapping that is $M : \left[0, 2^{|\mathcal{A}|} - 1\right] \rightarrow \left[0, 2^{|\mathcal{A}|} - 1\right]$.

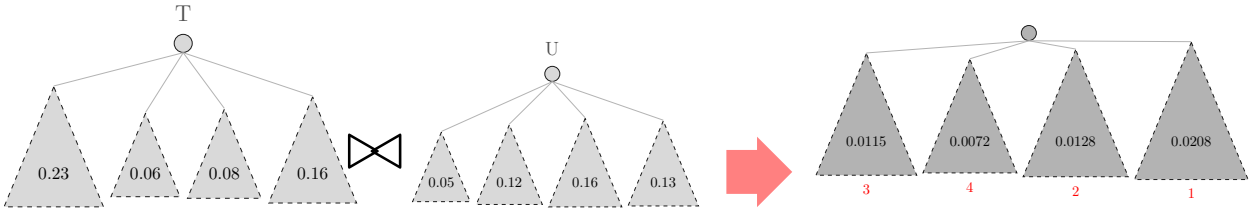


Figure 3.2: Order in partial results (density). Join between T and U. *We can see in red the order used to compute the join: the tuple of the last children is the one most likely to output more results.*

Algorithm 10 ANDGRADUALOPTORDER$(\mathbf{Q}_1, ..., \mathbf{Q}_n, \mathbf{q}, k)$ using DFUDS representation

Require: n qdags $\mathbf{Q}_1, ..., \mathbf{Q}_n$ representing relations $R_1(\mathcal{A}), ..., R_n(\mathcal{A})$, a max priority queue $\mathbf{q}$ with the tuples of nodes to visit in the future and k the maximum number of results we want to output.

Ensure: Up to k results representing points of the grid of the relation $\cap_{i=1}^{n} R_i(\mathcal{A})$.

1: $c \leftarrow 0$
2: $\mathbf{q}$.push({root($\mathbf{Q}_1$),...,root($\mathbf{Q}_n$)}, 0)
3: **while** $\mathbf{q}$ is not empty **and** $c < k$ **do**
4: tuple $\leftarrow \mathbf{q}$.pop()
5: $m \leftarrow \min\{$VALUE(tuple.$\mathbf{Q}_1$),...,VALUE(tuple.$\mathbf{Q}_n$)$\}$
6: **if** $m = 1$ **then**
7: **output** the tuple
8: c++
9: **else if** $m = 1/2$ **then**
10: **for** $i \leftarrow 0, ..., 2^d - 1$ **do**
11: **for** $j \leftarrow 1, ..., n$ **do**
12: TUPLE$_i[j]$ = CHILD$(\mathbf{Q}_j, i)$
13: $w \leftarrow \min\{$LEAFNUM$(\mathbf{Q}_1,$ CHILD$(\mathbf{Q}_1, i)), ...,$ LEAFNUM$(\mathbf{Q}_n,$ CHILD$(\mathbf{Q}_n, i))\}$
14: $\mathbf{q}$.push(TUPLE$_i$, w)

Using the LOUDS representation for the k^2-tree, we must modify line 14 according to Algorithm 8. Additionally, we must keep track of the level we are traversing. If we assume uniform distribution, we must divide each leaf number by the grid size. As said before in the motivation of the chapter, we only have to modify the AND algorithm (see Algorithm 6) to achieve gradual retrieval in the join operation.

Algorithm 11 MULTIJOINGRADUALOPTORDER $(R_1, ..., R_n, k)$

Require: Relations $R_1, ..., R_n$, stored as quadtrees $Q_1, ..., Q_n$; each relation R_i is over attributes $\mathcal{A}_i$ and $\mathcal{A} = \bigcup \mathcal{A}_i$. k is the maximum number of points we want to output.

Ensure: At most k results representing points of the grid of the output $J = R_1 \bowtie ... \bowtie R_n$.

1: **for** $i \leftarrow 0, ..., n$ **do**
2: Let $\mathbf{Q}_i$ be the qdag $(Q_i, \mathrm{Id}(\mathcal{A}_i))$
3: $\mathbf{Q}_i^* \leftarrow$ EXTEND$(\mathbf{Q}_i, \mathcal{A})$
4: $\mathbf{q} \leftarrow$ priQueue()
5: **return** ANDGRADUALOPTORDER$(\mathbf{Q}_1^*, ..., \mathbf{Q}_n^*, \mathbf{q}, k)$

The cost of computing ANDGRADUALOPTORDER$(\mathbf{Q}_1, ..., \mathbf{Q}_n, \mathbf{q}, k)$ is bounded by the cost of

retrieving the maximum tuple from the priority queue and the number of calls to LEAFNUM. The cost of line 4 does not depend on the data structure: it will be $\mathcal{O}(\log |Q^+|)$ for both LOUDS and DFUDS. The cost of line 14 is $\mathcal{O}(n)$ using DFUDS, due to its constant-time operation LEAFNUM, and for LOUDS is $\mathcal{O}(n \log l)$.

The cost analysis of the join is bounded by $|Q^+|$, the number of internal nodes that need to be visited to construct the resulting quadtree. In the worst case, we may need to examine each branch and node of Q^+ to compute the intersection. However, we expect the algorithm to perform better, providing results faster for a small k.

At worst, the cost of the whole operation using DFUDS is:

$$\mathcal{O}(|Q^+| \cdot (\log |Q^+| + 2^d n)).$$

The worst cost with LOUDS is higher because of its cost to compute the number of leaves:

$$\mathcal{O}(|Q^+| \cdot (\log |Q^+| + 2^d n + \log l)).$$

Comparing both algorithms, we added an extra time cost of $\mathcal{O}(\log l)$ using LOUDS instead of DFUDS. The trade-off is, of course, the memory needed to store the DFUDS representation, which is $2|\mathbf{Q}_i|$ extra bits for each qdag to store the input.

The backtracking method for gradual retrieval is straightforward: the first k results computed are output. Each result is stored in an array as the coordinates of the point in the grid representing the relation $\cap_{i=1}^{n} R_i(\mathcal{A})$. We explore branches in order from the most to the least promising and traverse their children to find results as quickly as possible. The pseudocode for this method is provided in Algorithms 12 and 13. However, ranked enumeration presents additional complexity, which is discussed in the following subsection.

Algorithm 12 ANDGRADUALBACKTRACKING($\mathbf{Q}_1, ..., \mathbf{Q}_n, \mathbf{res}, k$) using DFUDS representation

Require: n qdags $\mathbf{Q}_1, ..., \mathbf{Q}_n$ representing relations $R_1(\mathcal{A}), ..., R_n(\mathcal{A})$; an array **res** of size k with the results seen so far.

Ensure: Up to k results representing the points of the grid of the relation $\cap_{i=1}^{n} R_i(\mathcal{A})$.

1: $m \leftarrow \min\{$VALUE($\mathbf{Q}_1$),...,VALUE($\mathbf{Q}_n$)$\}$
2: **if** $m = 0$ **then return**
3: **if** $m = 1$ **then**
4: add Morton code of the leaf to **res**
5: **return**
6: orderToTraverse $\leftarrow$ priQueue()
7: **for** $i \leftarrow 0, ..., 2^d - 1$ **do**
8: **for** $j \leftarrow 1, ..., n$ **do**
9: TUPLE$_i[j]$ = root$\mathbf{Q}_j$
10: $w \leftarrow \min\{$LEAFNUM($\mathbf{Q}_1$, CHILD($\mathbf{Q}_1, i$)), ..., LEAFNUM($\mathbf{Q}_n$, CHILD($\mathbf{Q}_n, i$))$\}$
11: orderToTraverse.push(TUPLE$_i$, w)
12: **while** orderToTraverse is not empty **and** **res**.size $< k$ **do**
13: thisTuple $\leftarrow$ orderToTraverse.pop()
14: ANDGRADUALBACKTRACKING(thisTuple.tuple[1],...,thisTuple.tuple[n],**res**, k)

The cost of line 11 of Algorithm 12 (andGradualBacktracking) is the same as line 14 in Algorithm 10 (andGradualOptOrder) for LOUDS and DFUDS. On each call to andGradualBacktracking, we need to compute n times value, $\mathcal{O}(2^d \cdot n)$ times child, and $\mathcal{O}(2^d \cdot n)$ times leafNum. However, the cost of this last operation is charged to the child nodes as it is only computed if the child exists. Otherwise, the cost would be increased by an additional $\mathcal{O}(2^d n \cdot \log l)$ when using LOUDS. Then, the cost of the whole algorithm using DFUDS is:

$$\mathcal{O}(|Q^+| \cdot 2^d n),$$

and using LOUDS is:

$$\mathcal{O}(|Q^+| \cdot (2^d n + \log l))$$

An overview of the costs is provided in Table 3.2 later.

Algorithm 13 multiJoinGradualBacktracking $(R_1, ..., R_n, k)$

Require: Relations $R_1, ..., R_n$, stored as quadtrees $Q_1, ..., Q_n$; each relation R_i is over attributes $\mathcal{A}_i$ and $\mathcal{A} = \bigcup \mathcal{A}_i$.

Ensure: Points of the grid representing the output $J = R_1 \bowtie ... \bowtie R_n$.

1: **for** $i \leftarrow 0, ..., n$ **do**
2: Let $\mathbf{Q}_i$ be the qdag $(Q_i, \mathrm{Id}(\mathcal{A}_i))$
3: $\mathbf{Q}_i^* \leftarrow$ extend$(\mathbf{Q}_i, \mathcal{A})$
4: **res** $\leftarrow [\,]$
5: **return** andGradualBacktracking$(\mathbf{Q}_1^*, ..., \mathbf{Q}_n^*, \mathbf{res}, k)$

3.5 Ranked enumeration

Ranked enumeration queries aim to return the highest-ranked results one-at-a-time as quickly as possible, reducing the time to generate the first result and minimizing the delay between subsequent results. While this is similar to the top-k query evaluation problem [11], where k is specified in advance, and the goal is to retrieve the top-k results (not necessarily in order), our solutions maintain an ordered data structure. This structure keeps potential results sorted by a ranking function, allowing the system to output results one by one in ascending order of rank. This process evaluates tuples based on a predefined ranking criterion [20, 48, 50].

Enumerating query results in ranked order is crucial for many database management systems, as it reflects common real-world use cases. Many database languages support ranked enumerations–often through constructs such as `ORDER BY` and `LIMIT` clauses, typically through constructs such as 'ORDER BY' and 'LIMIT' clauses, which enable users to prioritize and limit result sets. Ranked retrieval is widely beneficial across domains such as web search and information retrieval [11] and has also been studied in the context of lexicographic orderings [20]. We can also extend this approach to multidimensional data, finding applications in areas like multimedia indexing and molecular biology [30].

Typically, database engines implement ranked enumeration by first computing the full join and then selecting the results, resulting in a time complexity of $\Omega(\mathrm{OUT})$, where OUT is the total number of results (the output size). However, when only the top-k results are needed,

we can reduce these costs by focusing on retrieving just those top k results. This approach is especially beneficial as it avoids computing unnecessary intermediate results [20, 49].

In this section, we explore the problem of ranked enumeration in the context of qdags. Our goal is to compute tuples with higher priority, focusing not only on improving the time to obtain the first results but also on returning the top-ranked results in order. This enhancement addresses a limitation of the original algorithm, which computes all results without considering any priority or weight, as discussed in Chapter 2 [6]. We follow the approach described in the literature [19, 20, 49], defining a ranking function that assigns a weight to each tuple. This ranking function introduces a total order $\succeq$ on the tuples in the relation and it is defined in Definition 9.

Each quadtree stores an array $P[1, p]$, where p is the number of points, and each element in P represents the weight or priority of a leaf. We precompute a Range Maximum Query (rMq) (see Section 2.3.1) on this array to compute the index of the range of priorities with the highest priority, and therefore know the maximum priority in a qdag subtree. We can still perform this operation after extending the quadtree, thanks to the mapping function. An example of this process is illustrated in Figure 3.3.

To compute the query results in order, we start by identifying the tuple with the highest priority. This involves evaluating the weight of a tuple by examining the maximum weight among its descendant leaves from each child, and then evaluating this across all qdags in the join. We refer to this algorithm as MAXPRI, which is quite similar to Algorithms 8 and 9. However, instead of returning the number of leaves, MAXPRI returns the maximum priority between the left-most and right-most children of a node. Consequently, the cost of MAXPRI using LOUDS is $\mathcal{O}(\log ||Q||)$, while using DFUDS is $\mathcal{O}(1)$ (see Section 3.2).

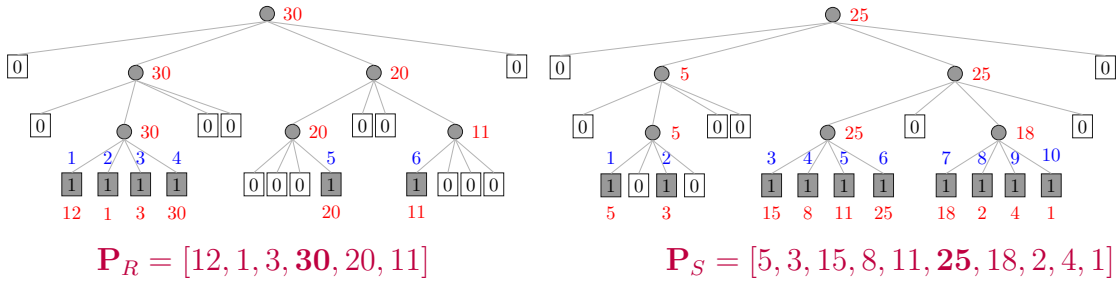


Figure 3.3: Order in ranked results. *We can see the priorities next to the node as a result of the range maximum query for each range. For example, first, we will compute the rMq of the roots, and we will have 4 and 6 as the positions of P with the highest weights, which are 30 and 25, respectively. We repeat the process for the children of these roots, and if we proceed by the* ***first measure*** *of Definition 9, we will first access the second child because the first tree has the highest priority leaf (30) on the second child. If we proceed by the* ***second measure*** *of Definition 9, we will first go down by the third child because the addition of the third nodes is* $20 + 25 = 45$*, which is bigger than the addition of the priorities of the second nodes (35).*

We use a priority queue for both strategies. For the optimal order strategy, we use it to manage the tuples we will visit next. When we reach a tuple of leaves, we output the result, which ensures that the results are output in decreasing order of priority. The pseudocode for this approach using DFUDS is presented in Algorithm 14. This algorithm is essentially

the same as Algorithm 10, but in line 14, instead of computing the number of leaves, we compute the maximum priority of the tuple. If we follow the second measure of Definition 9 to compute the weight of a tuple, we need to modify line 14 to sum the priorities of the tuple rather than finding the maximum. The cost of ANDRANKEDOPTORDER using DFUDS remains comparable to Algorithm 10, with an additional cost of $\mathcal{O}(2^d n)$ per call due to the rMq. Therefore, the final cost of ANDRANKEDOPTORDER using DFUDS is:

$$\mathcal{O}(|Q^+| \cdot (\log |Q^+| + 2^d n)),$$

and using LOUDS it is:

$$\mathcal{O}(|Q^+| \cdot (\log |Q^+| + 2^d n + \log l))$$

Algorithm 14 ANDRANKEDOPTORDER($\mathbf{Q}_1, ..., \mathbf{Q}_n, \mathbf{P}_1, ..., \mathbf{P}_n, \mathbf{q}, \mathbf{res}, k$)

Require: n qdags $\mathbf{Q}_1, ..., \mathbf{Q}_n$ representing relations $R_1(\mathcal{A}), ..., R_n(\mathcal{A})$; n vectors $\mathbf{P}_1, ..., \mathbf{P}_n$ with a priority for each point of its qdag; a max priority queue $\mathbf{q}$ with the tuples of nodes to visit in the future; an array **res** to store the results and the maximum number k of results we want to output.

Ensure: Up to top k points of the grid representing the relation $\cap_{i=1}^{n} R_i(\mathcal{A})$ ranked according to a priority.

1: $c \leftarrow 0$
2: $\mathbf{q}$.push({root($\mathbf{Q}_1$),...,root($\mathbf{Q}_n$)}, 0)
3: **while** $\mathbf{q}$ is not empty **and** $c < k$ **do**
4: tuple $\leftarrow \mathbf{q}$.pop()
5: $m \leftarrow \min\{$VALUE(tuple.$\mathbf{Q}_1$),...,VALUE(tuple.$\mathbf{Q}_n$)$\}$
6: **if** $m = 1$ **then**
7: **output** the tuple of qdags ($\mathbf{Q}_1$,...,$\mathbf{Q}_n$)
8: c++
9: **else if** $m = 1/2$ **then**
10: **for** $i \leftarrow 0, ..., 2^d - 1$ **do**
11: **for** $j \leftarrow 1, ..., n$ **do**
12: TUPLE$_i[j]$ = CHILD($\mathbf{Q}_j, i$)
13: $w \leftarrow \max\{$MAXPRI(TUPLE$_i$[1]),...,MAXPRI(TUPLE$_i[n]$)$\}$
14: $\mathbf{q}$.push(TUPLE$_i$, w)

Algorithm 15 MULTIJOINRANKEDOPTORDER ($R_1, ..., R_n, \mathbf{P}_1, ..., \mathbf{P}_n, k$)

Require: Relations $R_1, ..., R_n$, stored as quadtrees $Q_1, ..., Q_n$; each relation R_i is over attributes $\mathcal{A}_i$ and $\mathcal{A} = \bigcup \mathcal{A}_i$; n vectors $\mathbf{P}_1, ..., \mathbf{P}_n$ with a priority for each point of its qdag; and the maximum number k of results .

Ensure: Top-k points of the grid representing the output $J = R_1 \bowtie ... \bowtie R_n$.

1: **for** $i \leftarrow 0, ..., n$ **do**
2: Let $\mathbf{Q}_i$ be the qdag $(Q_i, \mathrm{Id}(\mathcal{A}_i))$
3: $\mathbf{Q}_i^* \leftarrow$ EXTEND($\mathbf{Q}_i, \mathcal{A}$)
4: $\mathbf{q} \leftarrow$ priQueue()
5: **res** $\leftarrow$ [ ]
6: **return** ANDRANKEDOPTORDER($\mathbf{Q}_1^*, ..., \mathbf{Q}_n^*, \mathbf{P}_1, ..., \mathbf{P}_n, \mathbf{q}, \mathbf{res}, k$)

For the backtracking approach, we use a priority queue to efficiently store the results and output them in increasing order. We modify Algorithms 14 and 15 to maintain a priority queue of the results. The pseudocode for this approach is provided in Algorithms 16 and 17. In line 3, we compute the maximum priority of a leaf ($l = 1$), as we are at the last level of the qdag, instead of computing the priority of a child, as done in line 16. In line 14, we map the corresponding i-th child of each qdag. From lines 18 to 24, we ensure that we do not explore a branch unless its priority is higher than the top priority in the queue. As shown in the previous algorithms, we only explore a promising branch if the upper bound of the tuple exceeds the priority of the k-th result. The priority queue stores the final results, and once all branches have been checked, the computation can be terminated.

Algorithm 16 ANDRANKEDBACKTRACKING($\mathbf{Q}_1, ..., \mathbf{Q}_n, \mathbf{P}_1, ..., \mathbf{P}_n, \mathbf{q}, k$)

Require: n qdags $\mathbf{Q}_1, ..., \mathbf{Q}_n$ representing relations $R_1(\mathcal{A}), ..., R_n(\mathcal{A})$; n vectors $\mathbf{P}_1, ..., \mathbf{P}_n$ with a priority for each point of its qdag; and a min priority queue $\mathbf{q}$ of size k with the top k results seen so far.

Ensure: Up to top-k points of the grid representing the relation $\cap_{i=1}^{n} R_i(\mathcal{A})$ ranked according to a priority.

```
1:  m ← min{VALUE(Q_1),...,VALUE(Q_n)}
2:  if m = 0 then return
3:  if m = 1 then
4:      w ← max{MAXPRI(Q_1),...,MAXPRI(Q_n)}
5:      if q is full then
6:          if q.top.w < w then
7:              q.pop()
8:              q.push({Q_1,...,Q_n}, w)
9:      else
10:         q.push({Q_1,...,Q_n}, w)
11: orderToTraverse ← priQueue()
12: for i ← 0, ..., 2^d − 1 do
13:     for j ← 1, ..., n do
14:         TUPLE_i[j] = CHILD(Q_j, i)
15:     w ← max{MAXPRI(TUPLE_i[1]),...,MAXPRI(TUPLE_i[n])}
16:     orderToTraverse.push(tuple_i, w)
17: while orderToTraverse is not empty do
18:     thisTuple ← orderToTraverse.pop()
19:     if q is not full or q.top.w < thisTuple.w then
20:         ANDRANKEDBACKTRACKING(thisTuple.tuple[1],...,thisTuple.tuple[n], P_1, ..., P_n, q, k)
```

The cost of performing *push* in the priority queue of results is $\mathcal{O}(\log k)$. Therefore, the cost of ANDRANKEDBACKTRACKING using DFUDS is:

$$\mathcal{O}((2^d n + \log k) \cdot |Q^+|)$$

Using LOUDS, the cost is:

$$\mathcal{O}((2^d n + \log l + \log k) \cdot |Q^+|)$$

Algorithm 17 multiJoinRankedBacktracking $(R_1, ..., R_n, \mathbf{P}_1, ..., \mathbf{P}_n, k)$

Require: Relations $R_1, ..., R_n$, stored as quadtrees $Q_1, ..., Q_n$; each relation R_i is over attributes $\mathcal{A}_i$ and $\mathcal{A} = \bigcup \mathcal{A}_i$; n vectors $\mathbf{P}_1, ..., \mathbf{P}_n$ with a priority for each point of its qdag.
Ensure: Top-k points of the grid representing the output $J = R_1 \bowtie ... \bowtie R_n$.

```
for i ← 0, ..., n do
    Let Q_i be the qdag (Q_i, Id(A_i))
    Q_i^* ← EXTEND(Q_i, A)
q ← priQueue()
return ANDRANKEDBACKTRACKING(Q_1^*, ..., Q_n^*, P_1, ..., P_n, q, k)
```

and Gradual retrieval		
Optimal order	LOUDS	$\mathcal{O}(\lvert Q^+\rvert \cdot (\log \lvert Q^+\rvert + 2^d n + \log l))$
	DFUDS	$\mathcal{O}(\lvert Q^+\rvert \cdot (\log \lvert Q^+\rvert + 2^d n))$
Backtracking	LOUDS	$\mathcal{O}(\lvert Q^+\rvert \cdot (2^d n + \log l))$
	DFUDS	$\mathcal{O}(\lvert Q^+\rvert \cdot 2^d n)$
and Ranked enumeration		
Optimal order	LOUDS	$\mathcal{O}(\lvert Q^+\rvert \cdot (\log \lvert Q^+\rvert + 2^d n + \log l))$
	DFUDS	$\mathcal{O}(\lvert Q^+\rvert \cdot (\log \lvert Q^+\rvert + 2^d n))$
Backtracking	LOUDS	$\mathcal{O}(\lvert Q^+\rvert \cdot (\log k + 2^d n + \log l))$
	DFUDS	$\mathcal{O}(\lvert Q^+\rvert \cdot (\log k + 2^d n))$
Original AND	LOUDS	$\mathcal{O}(\lvert Q^+\rvert \cdot 2^d n)$

Table 3.2: Time complexities for the AND operation.

Worst-case optimality As we showed, the multi-join algorithms are dominated by the AND operation (for the original algorithm, gradual retrieval, and also for ranked enumeration). In the original algorithm the AND operation costs $\mathcal{O}(2^d n|Q^+|) = \tilde{\mathrm{O}}(|Q^+|)$, which is worst-case optimal. As we can see in Table 3.2, all versions of the AND operation are also $\tilde{\mathrm{O}}(|Q^+|)$. Thus, we can conclude that our algorithms are worst-case optimal for the full join query. We expect to traverse much fewer nodes of Q^+ for a smaller k.

3.6 Experimental results

We implemented all our algorithms in C++17, utilizing the O3 optimization flag. In our **first experiment**, we conducted 50 different tests for each algorithm across various pattern queries (see Figure 2.15). The **second experiment** focused only on tests that produced more than 1000 results. In Table 3.3, we present the mean number of results produced for each pattern. All tests were performed using a single thread on an Intel Xeon E5-2609 computer equipped with 128GB of DDR3 1066 ECC RAM and three SATA3 hard drives (2

x 1TB and 1 x 2TB).

Some curves may exhibit irregular behavior; for instance, the execution time for certain patterns might be higher at $k = 10$ and lower at $k = 100$. This variability can be attributed to fluctuations in the shared server environment, which introduces some degree of noise into the measurements.

j3	j4	p2	p3	p4	s1	s2	s3	s4	t2	t3	t4	ti2	ti3	ti4	tr1	tr2
1131.18	292.78	28118.1	303.52	269.06	10.6	8.12	128.42	4.5	787.88	29.6	1860.83	5383.34	203545	2.24872e+06	11.62	14.9

Table 3.3: Average number of results per pattern.

We evaluated one approach for the upper-bound estimator for gradual retrieval (as detailed in Section 3.4) and another for ranked enumeration (as defined in Definition 9). For the gradual retrieval algorithms, we employed estimation intersection with the minimum, as it presented better results in terms of time (see Figure A.1 in Annex A). While for ranked enumeration, we used the maximum priority approach. Figure B.1 in Annex B illustrates the comparison between these two approaches. We compared our algorithms to the sequential version of the original algorithm, rather than the parallel version.

To compare the new algorithms with the original one, we will stop the latter after computing k results and build the corresponding compressed quadtree. We have adapted the original algorithm for this purpose. For the ranked results, we have created a priority file that assigns a random priority to each tuple in the dataset, ensuring that the same priority file is used consistently across all the ranked enumeration algorithms[3].

First scenario: all patterns

Gradual retrieval

In Figures 3.4 and 3.5, we present the execution time and the number of tuples visited by the gradual retrieval algorithms compared to the original algorithm used for retrieving up to k results, where $k = \{1, 10, 100, 1000\}$. The original algorithm shows a nearly constant execution time across different values of k in comparison with the gradual retrieval algorithms, making it the optimal choice regarding time efficiency. In contrast, the gradual retrieval algorithms exhibit a significant increase in execution time as k increases. For certain patterns, this increase is more evident between $k = 1$ and $k = 10$ (such as for J3, P3, P4, S1, S2, S3, S4, Ti4, Tr1 and Tr2). For other patterns, the increase is more pronounced between $k = 100$ and $k = 1000$ (as seen with J4, T3, Ti2 and Ti3).

[3]We also tested an alternative priority method, which was calculated as a permutation of the number of leaves, using the following exponential function:

$$\exp\left(\frac{1}{n}x\right),$$

where n represents the number of leaves and x corresponds to the value of the permutation. However, this biased scenario did not yield better results.

The LOUDS-Backtracking algorithm is ranked as the second fastest across nearly all tested patterns. It is followed by the DFUDS-Backtracking algorithm, and then by the LOUDS-Optimal Order and DFUDS-Optimal Order algorithms. In some cases, such as with the patterns T4, Ti3 and Ti4, the DFUDS-Backtracking algorithm is slightly faster than the LOUDS-Backtracking algorithm. Although there is no significant difference between the LOUDS and DFUDS data structures, we can observe a slight distinction between the two strategies employed: backtracking and optimal order.

In Figure 3.5, we plot the number of tuples visited to retrieve the first k results using gradual retrieval. As expected, the algorithms of the same approach traverse the same number of tuples: LOUDS-Backtracking and DFUDS-Backtracking, as well as LOUDS-Optimal Order and DFUDS-Optimal Order.

Generally, there is no significant difference between the backtracking algorithms and the original algorithms, as both visit almost the same number of tuples. The patterns where the optimal order algorithms visit fewer tuples than the backtracking algorithms are J4, S3 and T4. As in the previous figure, we see a notable increase in the number of tuples visited from $k = 1$ to $k = 10$ in the patterns J3, P3, P4, S1, S2, S3, S4, Ti4, Tr1 and Tr2, and between $k = 100$ and $k = 1000$ for the patterns J4, T3, Ti2 and Ti3.

In Table 3.4, we present the average number of tuples visited to retrieve k results for each algorithm across all patterns. As expected, algorithms that employ the same strategy visit the same number of tuples. All of the novel algorithms outperform the original algorithm in terms of the average number of tuples visited for $k = 1$ and $k = 1000$. Among the novel algorithms, the most efficient strategy for retrieving $k = 1$ and $k = 1000$ results is the backtracking approach, while the optimal order strategy yields fewer tuples on average for $k = 10$ and $k = 100$.

k	Original	G. Louds Back	G. Louds Opt	G. Dfuds Back	G. Dfuds Opt
1	5,451,446.50	5,156,158.10	5,315,398.40	5,156,158.10	5,315,398.40
10	10,972,050.60	12,244,376.30	11,825,519.70	12,244,376.30	11,825,519.70
100	14,740,835.10	15,069,112.90	14,772,250.20	15,069,112.90	14,772,250.20
1000	17,312,450.30	17,032,378.80	17,078,868.50	17,032,378.80	17,078,868.50

Table 3.4: Mean of the number of tuples visited to retrieve k results for each algorithm (gradual retrieval).

In Table 3.5, we present the percentage of tuples visited by the novel algorithms compared to the number of tuples visited by the original algorithm. All the gradual retrieval algorithms traverse nearly the same number of nodes as the original algorithm (approximately 98%). In fact, for values of $k = \{10, 100\}$, the novel algorithms visit more nodes than the original.

Ranked enumeration

In Figure 3.6, we display the execution times of ranked enumeration algorithms alongside the original algorithm for retrieving the top k results. The original algorithm must compute

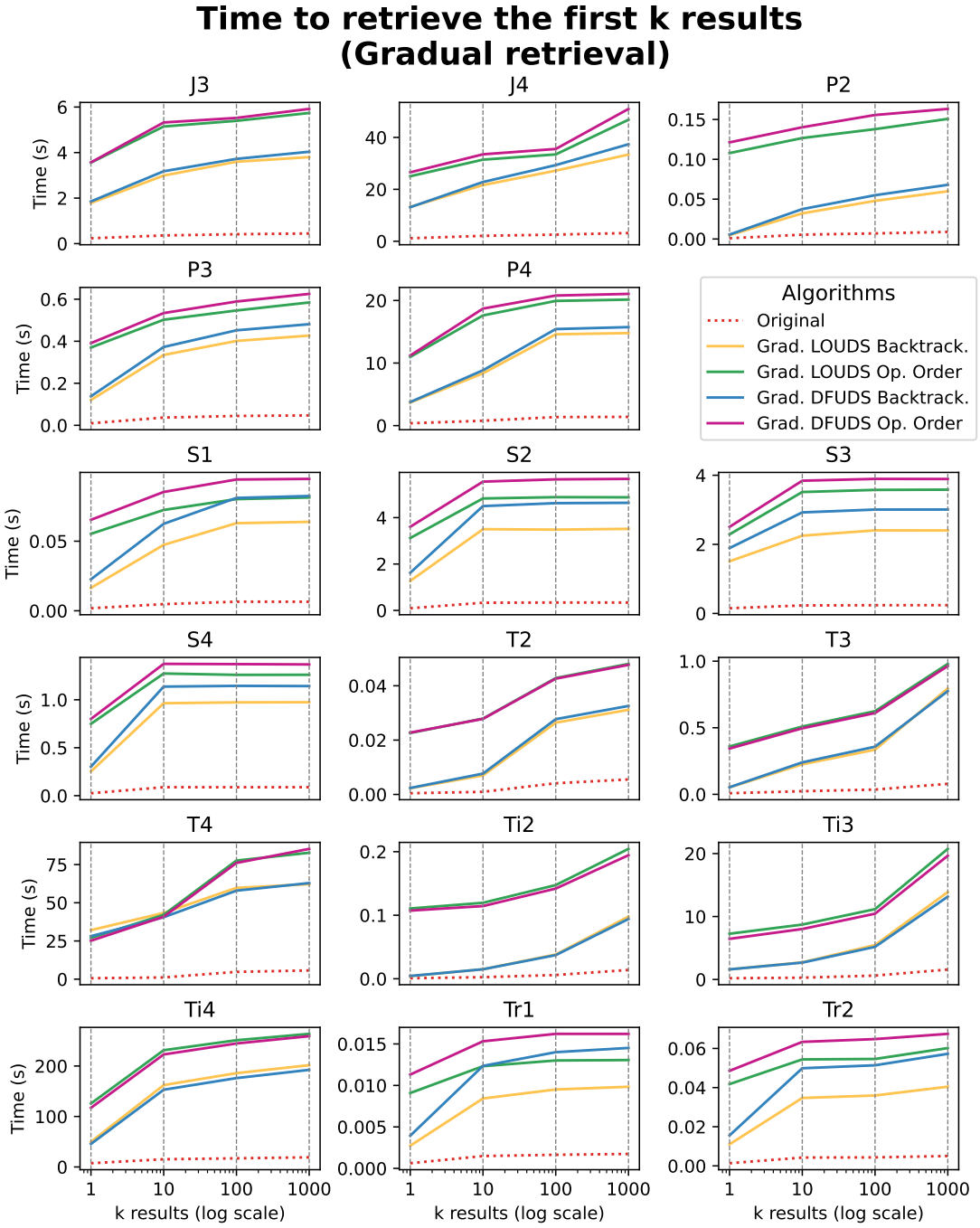


Figure 3.4: Query times (in seconds) to retrieve the first k results using the gradual retrieval algorithms.

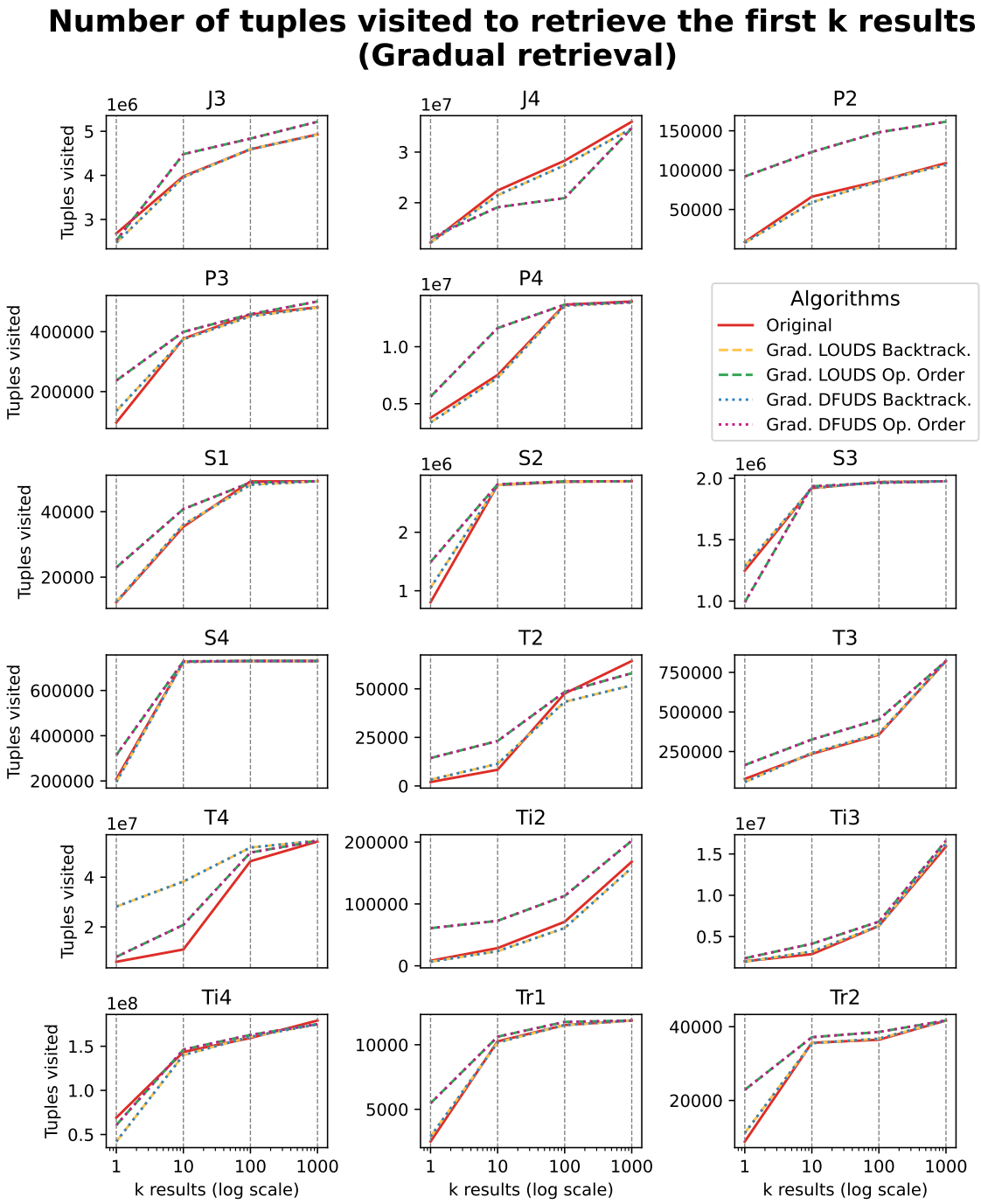


Figure 3.5: Number of tuples visited to retrieve the first k results using the gradual retrieval algorithms.

k	G. Louds Back	G. Louds Opt	G. Dfuds Back	G. Dfuds Opt
1	94.58%	97.50%	94.58%	97.50%
10	111.60%	107.78%	111.60%	107.78%
100	102.23%	100.21%	102.23%	100.21%
1000	98.38%	98.65%	98.38%	98.65%

Table 3.5: Percentage of tuples visited by gradual retrieval algorithms relative to the original algorithm.

all the results to obtain the top k entries, while the ranked enumeration algorithms retrieve only the top k results. This comparison highlights the performance difference, particularly in retrieving the top-1 result. Overall, the original algorithm maintains a faster execution time, except when retrieving the top-1 result in the pattern Ti2.

Among the ranked enumeration algorithms, the DFUDS-Backtracking and LOUDS-Backtracking methods are generally the most efficient, however, the differences among the four algorithms are minimal.

For patterns P2, T2, T3, Ti2, and Ti3, the execution times of the ranked enumeration algorithms continue to increase as k grows. In contrast, for the other patterns, the increase is less pronounced, and the execution time tends to stabilize after $k = 100$.

In Figure 3.7, we illustrate the number of tuples visited to retrieve the top k results for the ranked enumeration algorithms. The original algorithm needs to traverse significantly more tuples than the new algorithms. Across all patterns, the novel algorithms visit an equal or smaller number of tuples than the original algorithm for all values of k. This difference is particularly noticeable when retrieving the top-1 result. A clearer distinction is evident in patterns P2, T2, T3, Ti2, Ti3, and Ti4. As in the previous figure, the number of tuples visited tends to increase between $k = 1$ and $k = 100$ and then stabilizes.

Among the new algorithms, the differences in the number of tuples visited are not significant, except in patterns J4, S3 and S4, where the optimal order strategy outperforms.

In Table 3.6, we present the mean of the nodes visited to retrieve the top k results between all the patterns for each algorithm. As expected, the most efficient strategy in terms of mean nodes visited is the optimal order, followed by the backtracking approach, and finally the original algorithm.

k	Original	R. Louds Back	R. Louds Opt	R. Dfuds Back	R. Dfuds Opt
1	23,128,759.10	11,048,285.60	8,773,763.40	11,048,285.60	8,773,763.40
10	23,128,759.10	16,389,733.10	15,259,889.80	16,389,733.10	15,259,889.80
100	23,128,759.10	18,379,702.90	17,514,963.60	18,379,702.90	17,514,963.60
1000	23,128,759.10	19,103,629.90	18,890,963.40	19,103,629.90	18,890,963.40

Table 3.6: Mean of the number of tuples visited to retrieve the top k results for each algorithm (ranked enumeration).

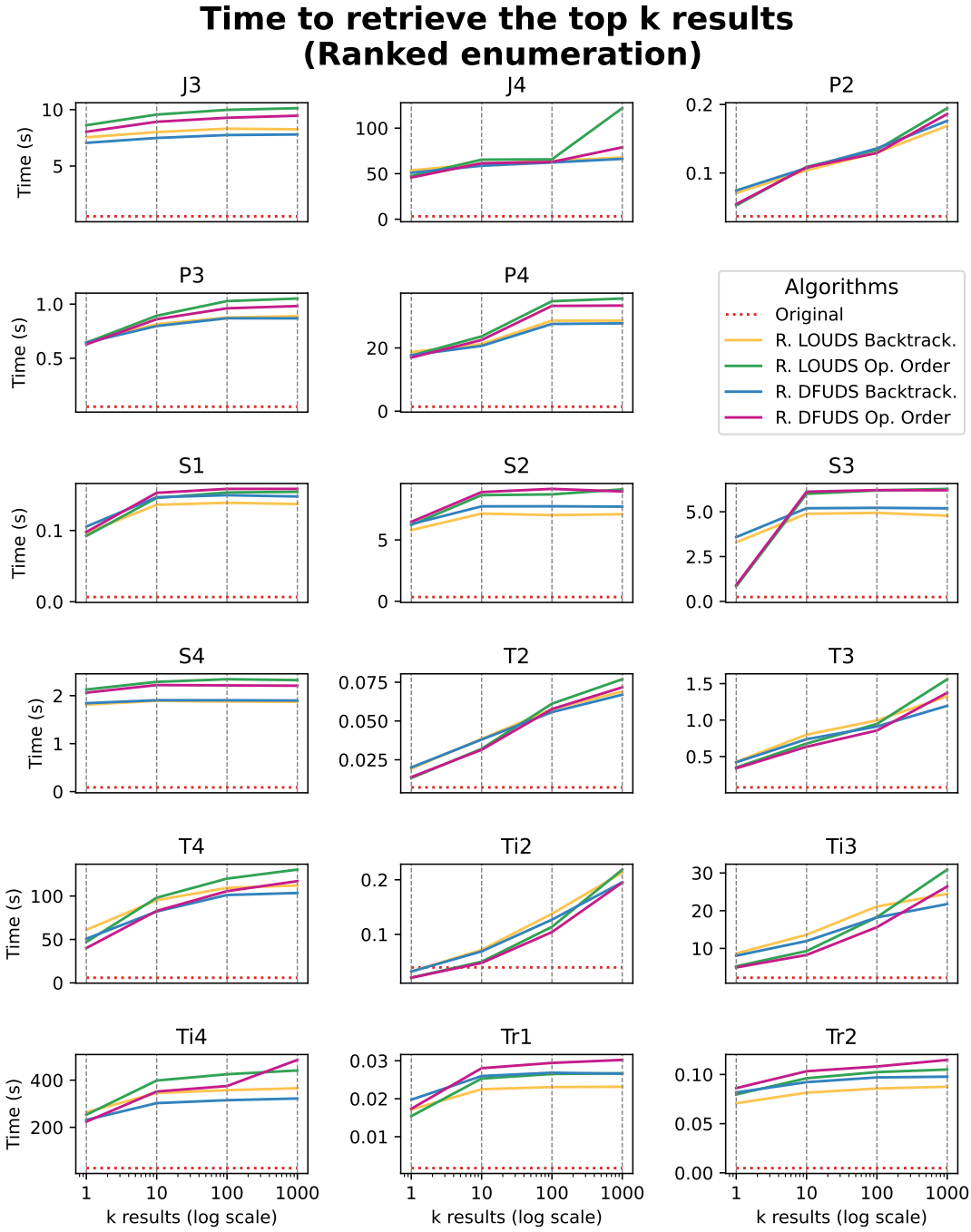


Figure 3.6: Query times (in seconds) to retrieve the top k results using the ranked enumeration algorithms.

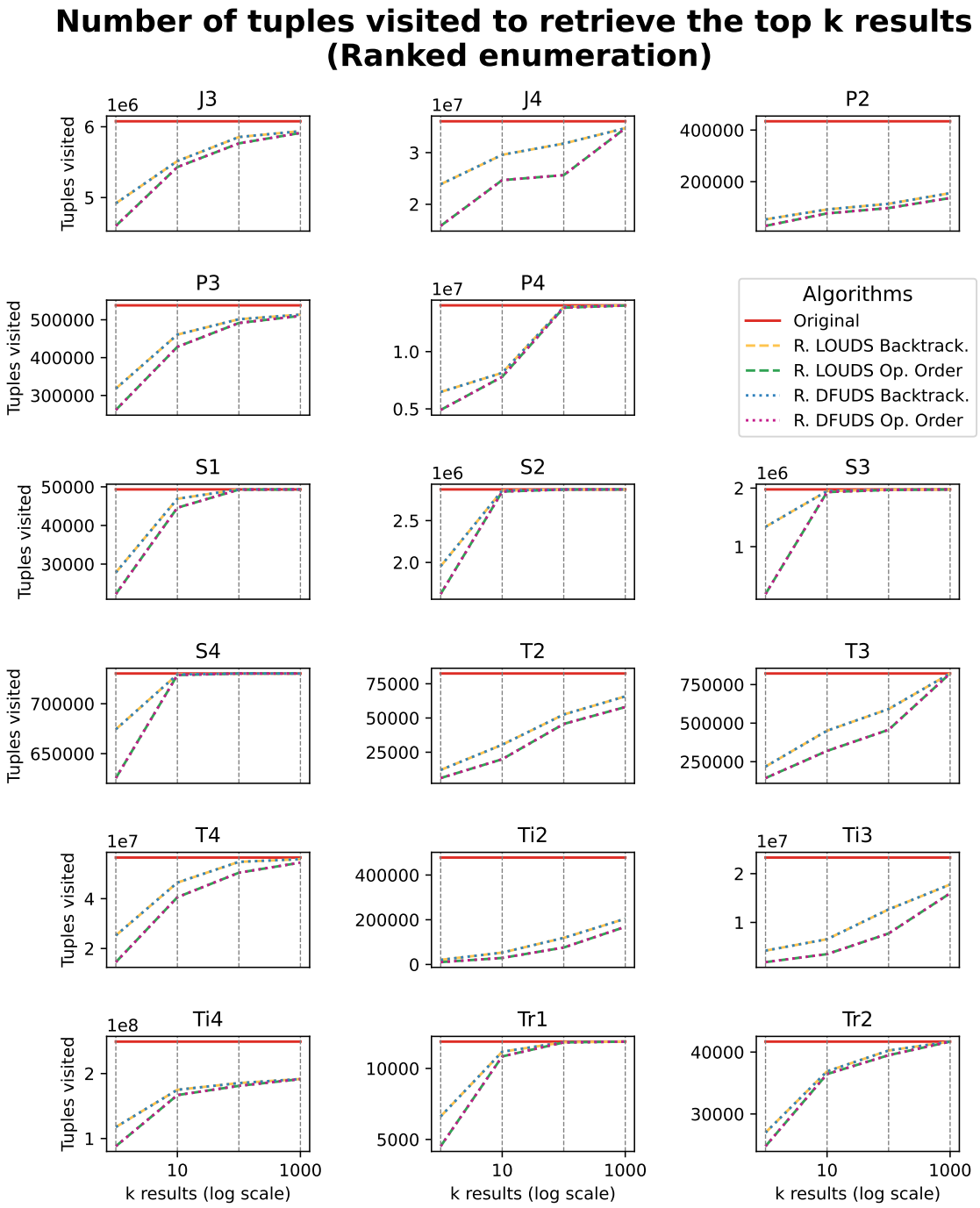


Figure 3.7: Number of tuples visited to retrieve the top k results using the ranked enumeration algorithms.

In Table 3.7, we show the percentage between the number of tuples visited for the novel algorithms divided by the number of tuples visited by the original algorithm. We can see that all the novel algorithms traverse fewer tuples than the original. The most significant difference is observed when $k = 1$, where the optimal order algorithms visit only about 37% of the tuples compared to the original algorithm.

k	R. Louds Back	R. Louds Opt	R. Dfuds Back	R. Dfuds Opt
1	47.77%	37.93%	47.77%	37.93%
10	70.86%	65.98%	70.86%	65.98%
100	79.47%	75.73%	79.47%	75.73%
1000	82.60%	81.68%	82.60%	81.68%

Table 3.7: Percentage of tuples visited by ranked enumeration algorithms relative to the original algorithm.

Second scenario: join produces many results

While the first experiment allowed us to compare our new algorithms with the original one using the same tests presented in [6], we considered it valuable to evaluate our algorithms in a scenario where the join operation produces many results. There were not many queries that produce more than 1000 results. Then, we could only test these patterns: J3, J4, P2, P3, P4, S3, T2, T3, T4, Ti2, Ti3, Ti4, and for each pattern the number of queries were just a few ($\sim$4.5 queries per pattern).

Gradual retrieval

In Figures 3.8 and 3.9, we show the time and the number of tuples visited to retrieve the first k results using the original and the gradual retrieval algorithms. In Figure 3.8, it is evident that the original algorithm outperforms the gradual retrieval algorithms in terms of time across all values of k. The backtracking algorithms show similar behavior across various patterns, particularly for patterns J3, P2, S3, T2, T3, T4, and Ti2 when $k < 100$.

As observed in the first scenario, the backtracking approach is generally faster than the optimal order approach, and LOUDS generally outperforms DFUDS. For most patterns, the performance curve remains relatively flat for $k < 100$, after which it begins to rise. An exception is pattern P4, where the curve increases sharply between $k = 1$ and $k = 10$ before stabilizing.

As shown in Figure 3.9, the original algorithm and the backtracking algorithms visit fewer tuples compared to the optimal order algorithms for all patterns when $k < 10$. For the patterns J3, P2, S3, T2, T3, Ti2 and Ti3, the backtracking algorithms outperform the original one. Generally, the optimal order algorithms tend to visit a higher number of tuples across most patterns. However, for $k = 1000$ in the patterns J4 and T2, the new algorithms achieve the best performance in terms of the number of tuples visited.

In general, we can observe a slight increase in the number of tuples visited as k rises and when retrieving more than 100 results, the number of tuples visited rises significantly.

In Figures 3.10 and 3.11 we plot the execution time and the number of tuples visited, normalized by the time and the number of tuples visited by the original algorithm, respectively. It is clear that many queries take up to ten times longer to retrieve the first result using the gradual retrieval algorithms compared to the original algorithm, as illustrated in Figure 3.10. Furthermore, Figure 3.11 shows that many queries either visit fewer nodes or visit up to twice as many nodes as those in the original algorithm.

Table 3.8 presents the average number of tuples visited to retrieve k results for each algorithm across all patterns tested in this scenario. Overall, the backtracking strategy remains the most effective in minimizing the number of tuples visited.

k	Original	G. Louds Back	G. Louds Opt	G. Dfuds Back	G. Dfuds Opt
1	8,667,373.30	347,080.30	2,111,081.70	347,080.30	2,111,081.70
10	9,896,986.40	2,209,735.60	3,311,982.80	2,209,735.60	3,311,982.80
100	19,238,265.40	3,908,781.20	3,904,030.30	3,908,781.20	3,904,030.30
1000	100,547,947.90	7,212,520.10	7,951,830.90	7,212,520.10	7,951,830.90

Table 3.8: Mean of the number of tuples visited to retrieve k results for each algorithm of a join that produces numerous outputs (gradual retrieval).

In Table 3.9, we show the percentage of tuples visited by the gradual retrieval algorithms relative to the original algorithm. When k =1, the backtracking strategy allows us to traverse only a 4% of the tuples visited by the original algorithm. For $k = 1000$, the percentage of tuples visited is further reduced to approximately 7%. This indicates that, in this scenario, we traverse significantly fewer tuples compared to the original algorithm, as shown in Table 3.5.

k	G. Louds Back	G. Louds Opt	G. Dfuds Back	G. Dfuds Opt
1	4.00%	24.36%	4.00%	24.36%
10	22.33%	33.46%	22.33%	33.46%
100	20.32%	20.29%	20.32%	20.29%
1000	7.17%	7.91%	7.17%	7.91%

Table 3.9: Percentage of tuples visited by gradual retrieval algorithms relative to the original algorithm of a join that produces many results.

Ranked enumeration

In Figures 3.12 and 3.13, we observe the execution time and the number of tuples visited by both the ranked enumeration algorithms and the original algorithm. The original algorithm retrieves all results to compute the top k results, which means its execution time and the number of tuples visited remain constant, regardless of the value of k.

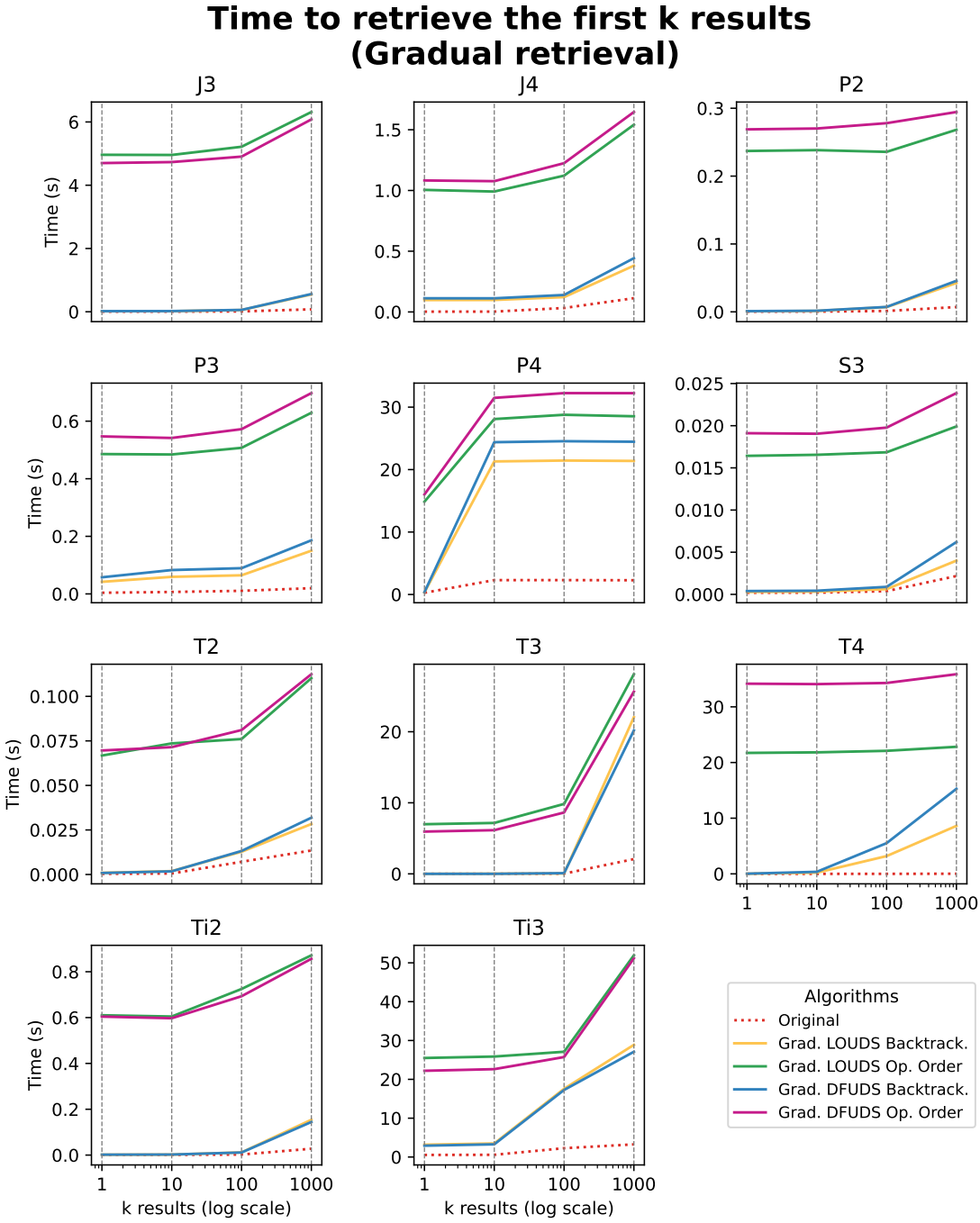


Figure 3.8: Query times (in seconds) to retrieve the first k results using the gradual retrieval algorithms.

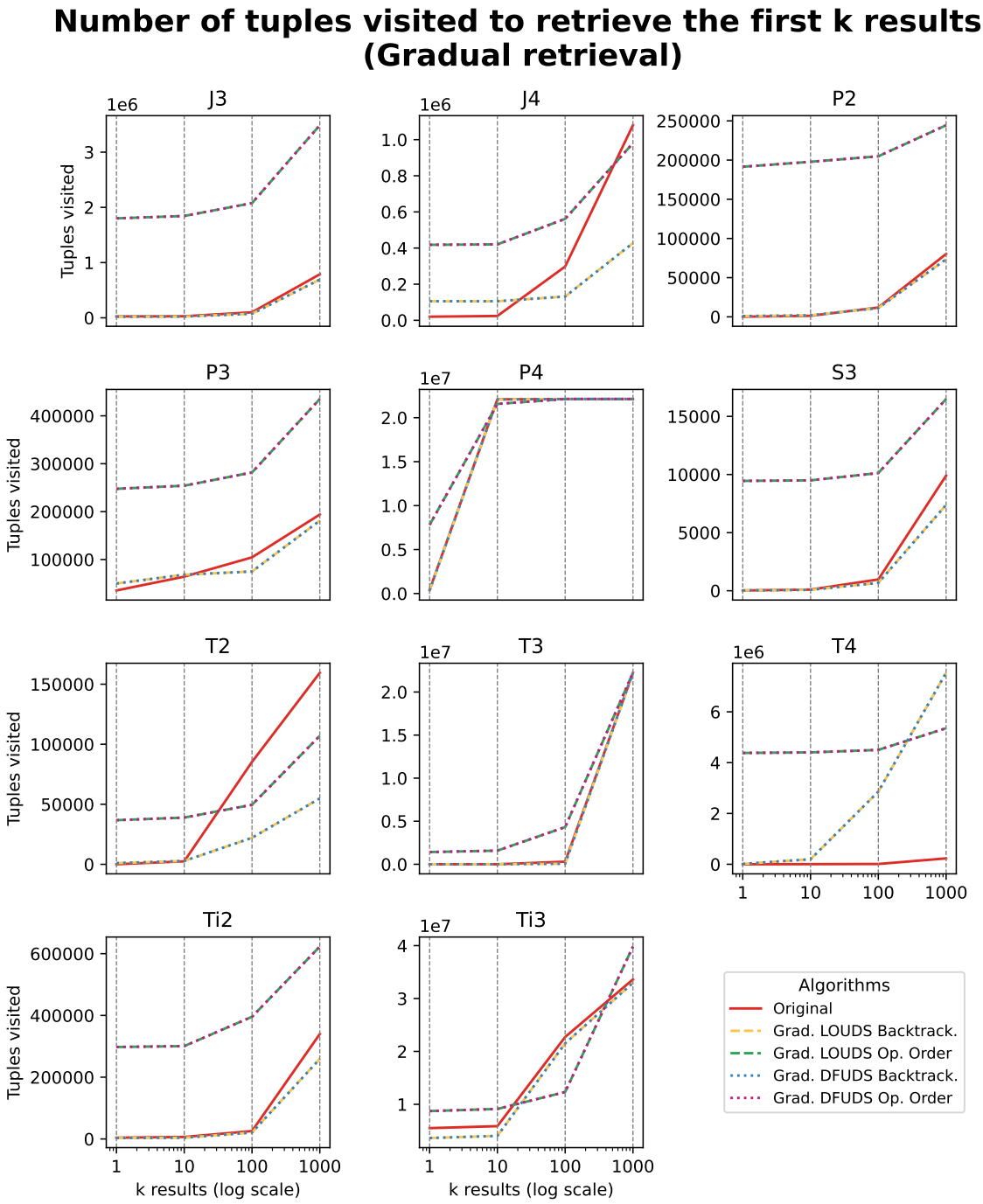


Figure 3.9: Number of tuples visited to retrieve the first k results using the gradual retrieval algorithms.

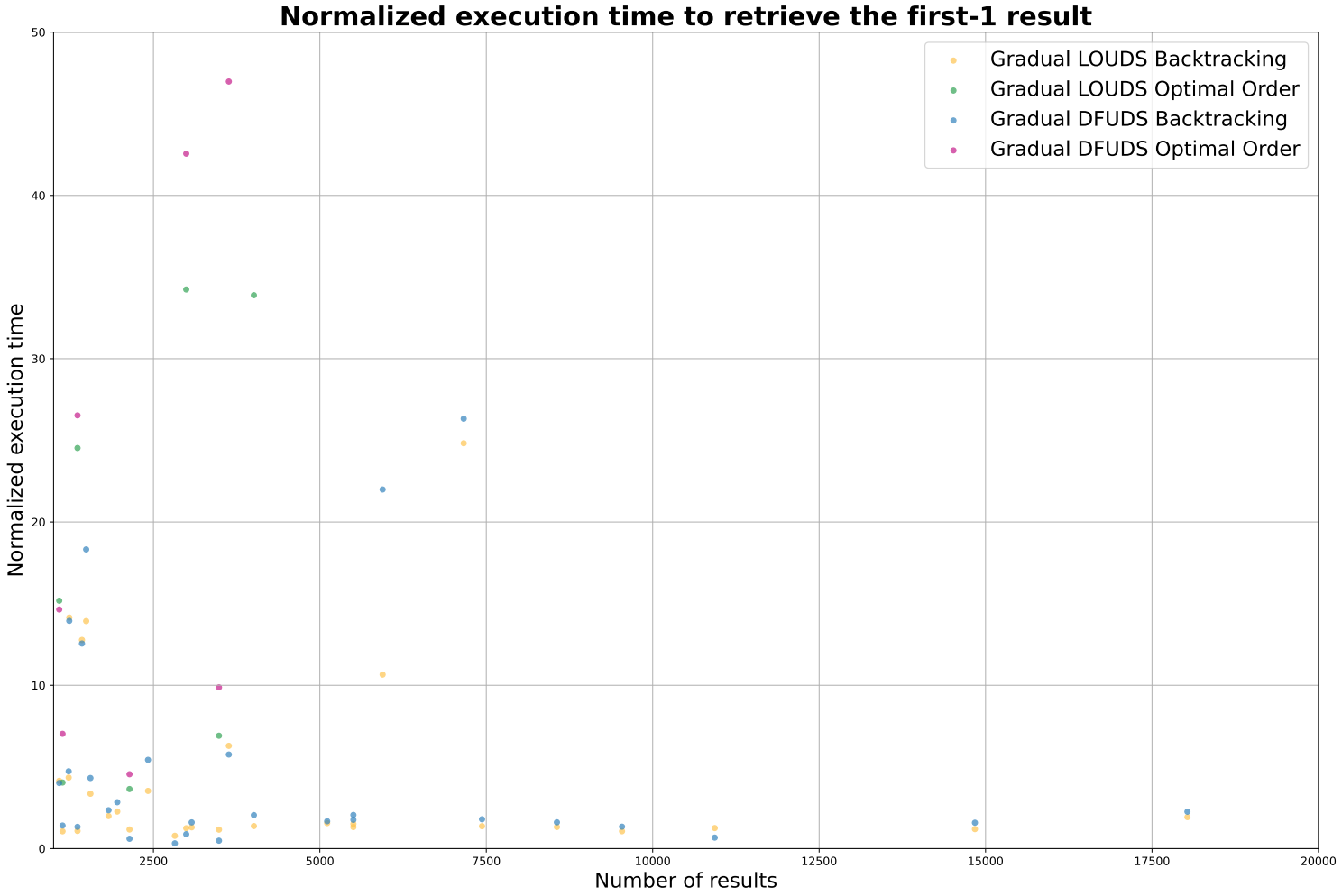


Figure 3.10: Query times to retrieve the first result ($k = 1$) using the gradual retrieval algorithms normalized by the original algorithm.

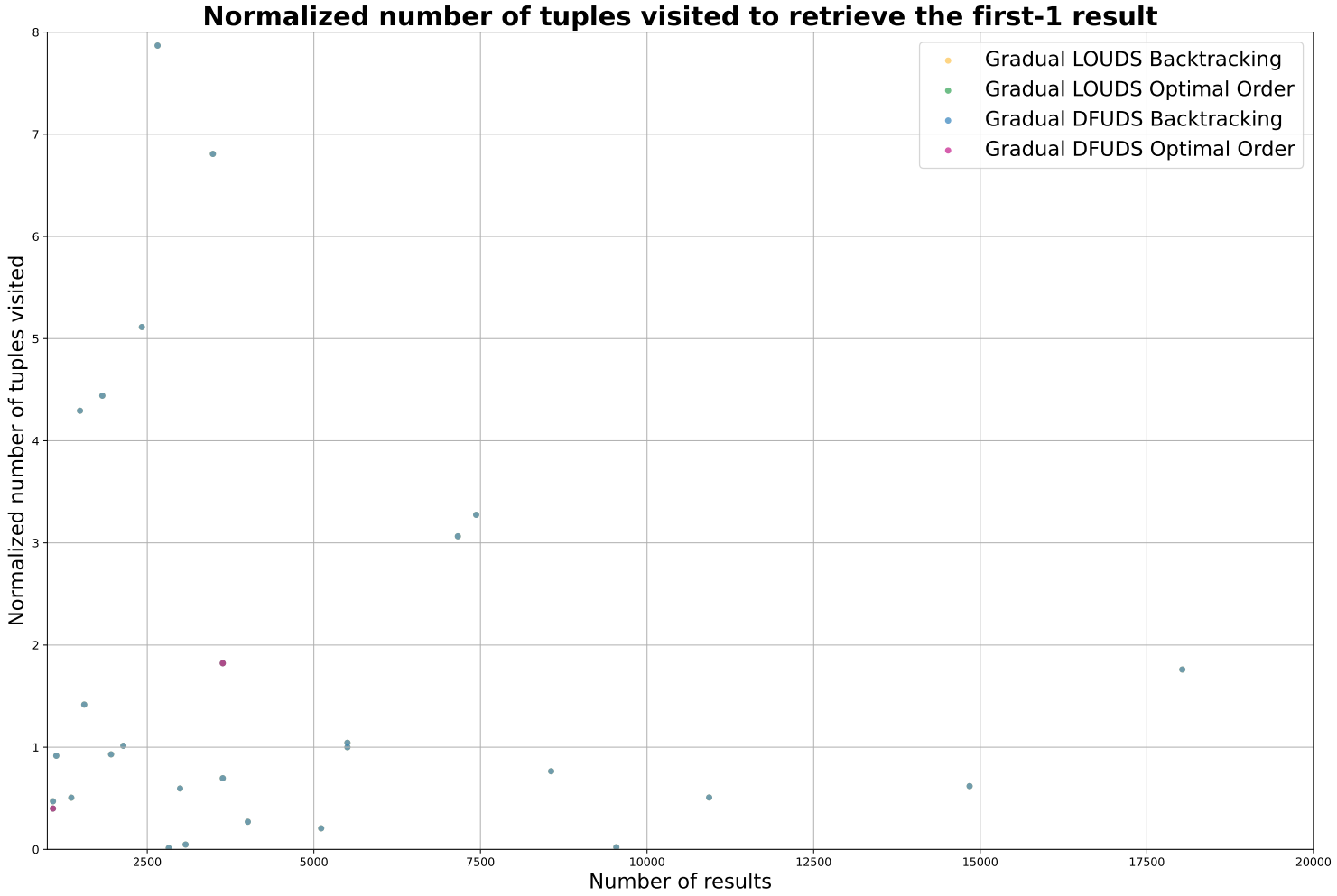


Figure 3.11: Number of tuples visited to retrieve the first result ($k = 1$) using the gradual retrieval algorithms normalized by the original algorithm.

Despite this, Figure 3.12 indicates that the original algorithm outperforms the ranked enumeration algorithms across almost all patterns. In the patterns P2, S3, T2, T3 and Ti2, the ranked enumeration algorithms perform similarly to the original algorithm or even better for $k = 1$.

For patterns J4 and P4, the performance curves of the ranked enumeration algorithms remain primarily flat. In contrast, the other patterns show an increase in execution time as k rises, with a particularly notable increase occurring between $k = 1$ and $k = 100$.

Among the novel algorithms, the DFUDS-Backtracking algorithm generally performs better, followed by the LOUDS-Backtracking algorithm. The algorithms that use the optimal order strategy show similar performance characteristics to one another.

Figure 3.13 shows that the original algorithm requires traversing significantly more tuples to retrieve the top k results. Additionally, it is clear that as k increases, the number of tuples visited by the new algorithms also rises. Overall, all ranked enumeration algorithms perform similarly in terms of number of tuples visited.

In Figures 3.14 and 3.15, we plot the execution time and the number of tuples visited to retrieve the top-1 result normalized by the time and number of tuples visited of the original algorithm when retrieving all the results. The data shows that a significant number of queries require less time to obtain the top result compared to the original algorithm, and many queries visit fewer tuples as well.

In Table 3.10, we present the average number of tuples visited to retrieve k results for each algorithm across the different patterns. The original algorithm requires the highest number of node visited compared to the other algorithms, demonstrating a significant difference between the original and the novel approaches. Among the ranked enumeration algorithms, the optimal order algorithms consistently visit fewer tuples for all values of k, as anticipated.

In Table 3.11, we show the percentage of tuples visited by the ranked enumeration algorithms relative to the original algorithm. For $k = 1$, , the optimal order algorithms only visit about 1% of the tuples that the original algorithm processes. To retrieve the top-1000 results, we only need to visit approximately 4% of the tuples required by the original algorithm. Additionally, this scenario demonstrates a significant reduction in the number of tuples visited compared to what is shown in Table 3.7 for the first scenario.

k	Original	R. L. Back	R. L. Opt. Order	R. D. Back	R. D. Opt. Order
1	205,487,271.30	4,156,613.20	2,508,591.20	4,156,613.20	2,508,591.20
10	205,487,271.80	5,988,946.30	3,202,413.80	5,988,946.30	3,202,413.80
100	205,487,276.80	9,958,415.20	5,276,732.00	9,958,415.20	5,276,732.00
1000	205,487,326.80	13,327,577.40	8,984,728.50	13,327,577.40	8,984,728.50

Table 3.10: Mean of the number of tuples visited to retrieve k results for each algorithm of a join that produces numerous outputs.

k	R. Louds Back	R. Louds Opt	R. Dfuds Back	R. Dfuds Opt
1	2.02%	1.22%	2.02%	1.22%
10	2.91%	1.56%	2.91%	1.56%
100	4.85%	2.57%	4.85%	2.57%
1000	6.49%	4.37%	6.49%	4.37%

Table 3.11: Percentage of tuples visited by ranked enumeration algorithms relative to the original algorithm of a join that produces many results.

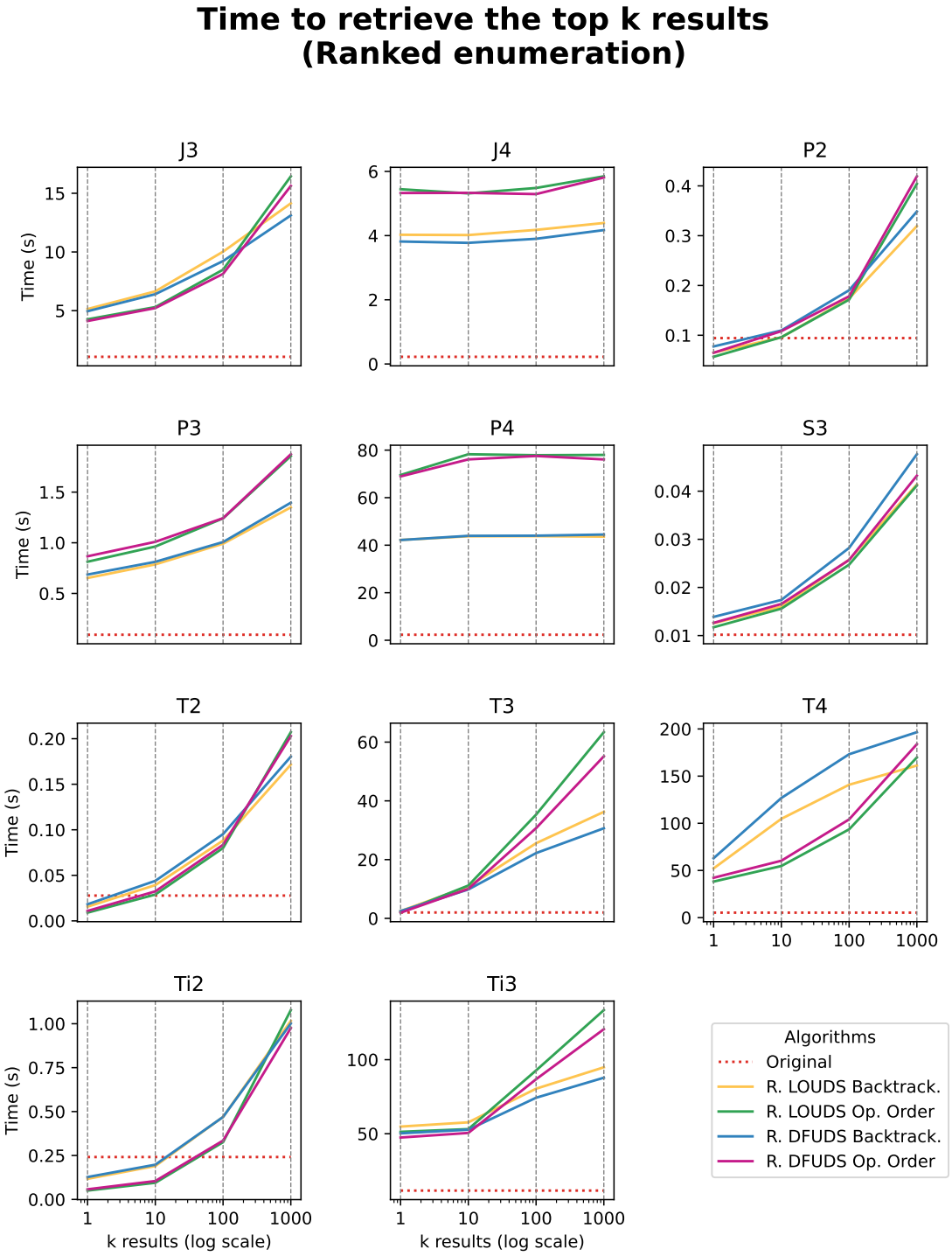


Figure 3.12: Query times (in seconds) to retrieve the first k results using the ranked enumeration algorithms.

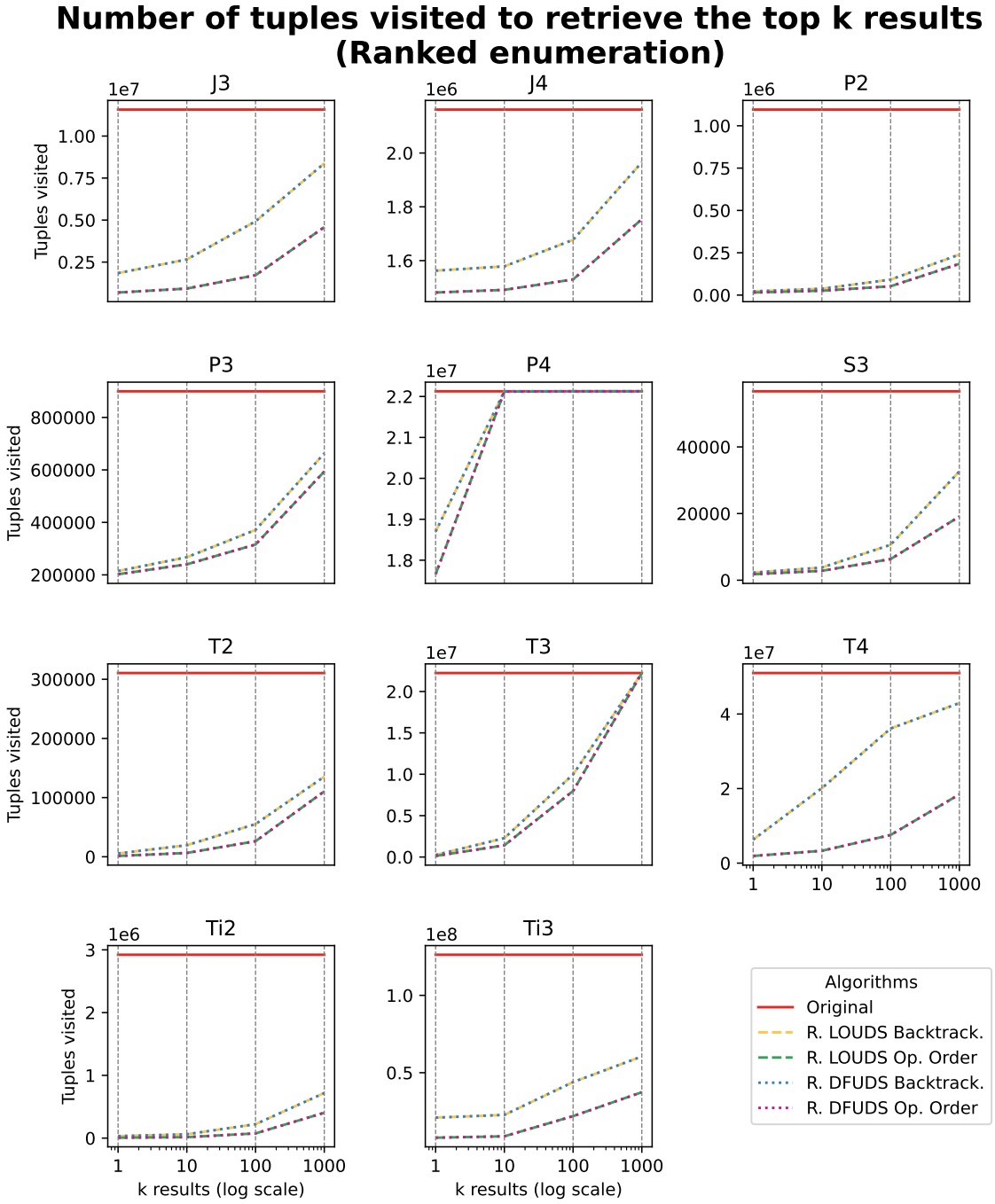


Figure 3.13: Number of tuples visited to retrieve the first k results using the ranked enumeration algorithms.

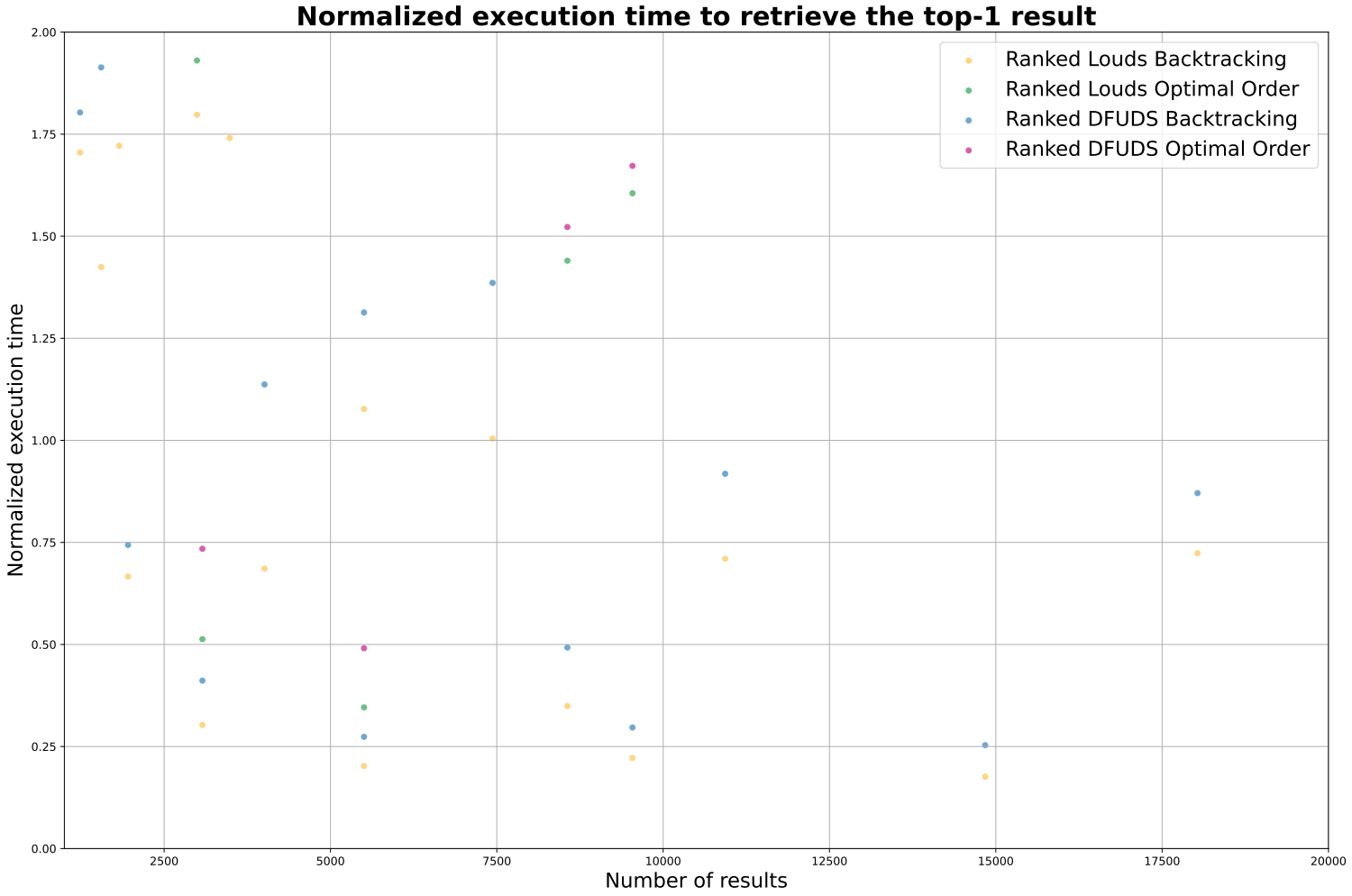


Figure 3.14: Query times to retrieve the top 1 result using the ranked enumeration algorithms normalized by the original algorithm.

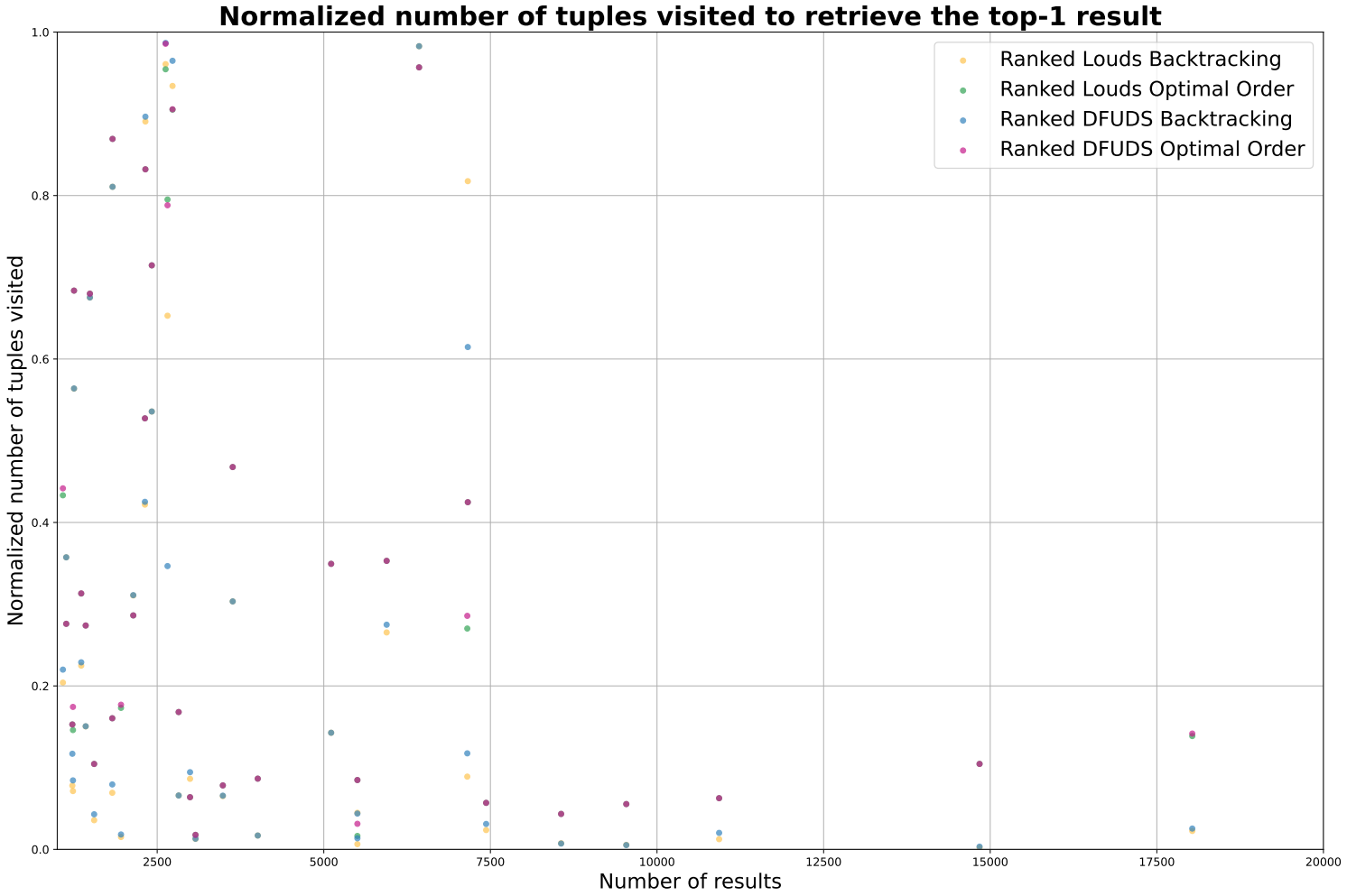


Figure 3.15: Number of tuples visited to retrieve the top 1 result using the ranked enumeration algorithms normalized by the original algorithm

Chapter 4

Lazy qdags

4.1 Motivation

In the same context of gradually retrieving a subset of the join results rather than all or a significant part, it is often desirable to process the join only for a few tuples or those of specific interest. In such cases, performing the join requires computing a limited number of qdag tuples, eliminating the need to process the entire qdag. For instance, if a quadrant of a quadtree is empty, any AND operation involving that quadrant will yield an empty result, making further computation unnecessary. In this scenario, it is helpful to adopt a lazy evaluation strategy to efficiently handle such operations without generating redundant children of the qdag.

This approach also allows for more flexible query evaluation compared to traditional qdags. It supports the combination of various relational operations, such as intersection, union, and complement, which enables the evaluation of Boolean formulas and extends its applicability to more complex and generalized queries in relational algebra.

In this context, we aim to implement a lazy version of qdags [6], which embodies this idea by maintaining the syntax tree of a query, where the leaves are quadtrees and the internal nodes are operators (functors). The laziness stems from how we evaluate a formula. Instead of eagerly solving the formula and evaluating the entire dataset or performing all the operations upfront, we compute only the necessary portions of the data and evaluate the formula as needed. This approach allows us to progressively obtain results, which is particularly advantageous when we do not need to compute all the results. Additionally, evaluating only what is needed helps us to obtain optimality on those more complex formulas.

While we still guarantee worst-case optimality for evaluating Boolean formulas, we can no longer ensure the same for the relational algebra. As we will see, lazy qdags can improve space and time complexity for certain operations, though in some cases, the time cost may increase due to the need to traverse the syntax tree. Nonetheless, this approach enables faster retrieval of the first results in a more general context.

This chapter outlines the implementation of the syntax tree and evaluates its performance

across Boolean algebra, including join queries, and relational algebra operations.

4.2 Definition

Let us begin by defining what is a Boolean formula and the syntax tree used to construct a lazy qdag.

Definition 10 (Boolean formula)**.** A Boolean formula $\mathcal{F}$ is composed by join (`AND` and `EXTEND`), union (`OR`), and complement (`NOT`) operations over relations.

Definition 11 (Lazy qdags)**.** A lazy qdag [6] (lqdag) $\mathcal{L}$ is a recursive data structure defined as a pair (f, o) consisting of a functor and a list of operands, which follows the following rules:

1. $\mathcal{L} = (\texttt{QTREE}, Q_R)$, where Q_R is a quadtree that represents the relation R, represents the relation R.
2. $\mathcal{L} = (\texttt{NOT}, Q_R)$, where Q_R is a quadtree that represents the relation R, represents the complement $\overline{R}$ of R.
3. $\mathcal{L} = (\texttt{AND}, \mathcal{L}_1, \mathcal{L}_2)$, where $\mathcal{L}_1$ and $\mathcal{L}_2$ are lqdags, represents the intersection of the relations represented by $\mathcal{L}_1$ and $\mathcal{L}_2$.
4. $\mathcal{L} = (\texttt{OR}, \mathcal{L}_1, \mathcal{L}_2)$, where $\mathcal{L}_1$ and $\mathcal{L}_2$ are lqdags, represents the union of the relations represented by $\mathcal{L}_1$ and $\mathcal{L}_2$.
5. $\mathcal{L} = (\texttt{EXTEND}, \mathcal{L}_1, \mathcal{A})$, where $\mathcal{L}_1$ is an lqdag, represents the relation of $\mathcal{L}_1$ with its attributes extended to $\mathcal{A}$.
6. $\mathcal{L} = (\texttt{LQDAG}, \mathcal{L}_1)$, where $\mathcal{L}_1$ is an lqdag, represents the same lqdag $\mathcal{L}_1$.

The leaves of a formula are lazy qdags with functors `QTREE`, `NOT`, and `LQDAG`, where their operand is a quadtree or another lqdag. By using lazy qdags as leaves (not considered in the original design [6]), we can construct more complex formulas and avoid redundant computations of previously evaluated formulas. Internal nodes are lazy qdags with functors like `AND`, `OR`, and `EXTEND`. This structure can be visualized as a syntax tree, where each node represents a functor and edges connect its operands.

For example, a join between the relations R, S, and T, where $\mathcal{A} = \{A \cup B \cup C\}$:

$$R(A, B) \bowtie S(B, C) \bowtie T(A, C)$$

will have the following formula $\mathcal{L}_{R \bowtie S \bowtie T}$, illustrated in Figure 4.1.

$$\begin{aligned}
&\texttt{AND}(\\
&\quad \texttt{AND}(\\
&\qquad (\texttt{EXTEND}(\texttt{QTREE}, Q_R), \mathcal{A}), \\
&\qquad (\texttt{EXTEND}(\texttt{QTREE}, Q_S), \mathcal{A})), \\
&\quad (\texttt{EXTEND}(\texttt{QTREE}, Q_T), \mathcal{A}))
\end{aligned} \tag{4.1}$$

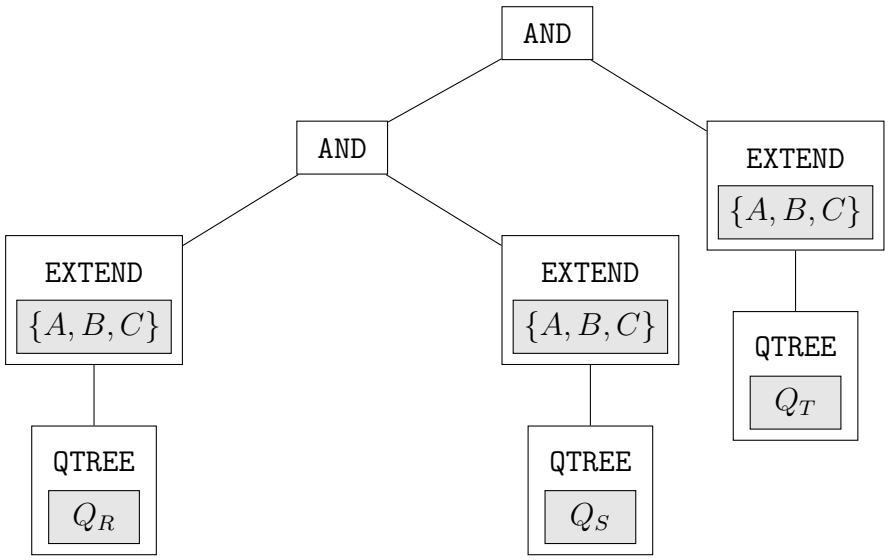


Figure 4.1: Example of the lazy qdag $\mathcal{L}_{R \bowtie S \bowtie T}$.

As another example, the next formula represents the difference between $\mathcal{L}_1$ and $\mathcal{L}_2$ and is illustrated in Figure 4.2.

$$(\texttt{DIFF}, \mathcal{L}_1, \mathcal{L}_2) = \texttt{AND}(\mathcal{L}_1, \texttt{NOT}(\mathcal{L}_2)) \tag{4.2}$$

We can see the use of the `NOT` functor together with a lazy qdag $\mathcal{L}$ instead of a quadtree. We always push down the `NOT` functor until its operand is a quadtree (`QTREE`) or another lqdag (`LQDAG`). This transformation allows us to maintain the benefits of lazy evaluation while simplifying the logical structure of the query.

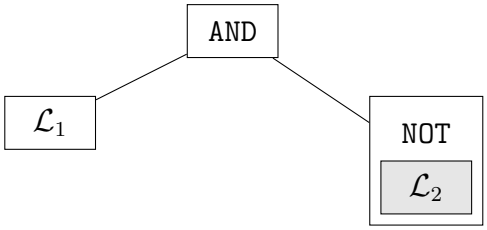


Figure 4.2: Example of the syntax tree of the formula in Equation 4.2.

Traversing a lazy qdag results in materializing its output as a traditional quadtree. Lazy qdags implicitly represent the outcome of a Boolean formula, using qdags only for the leaves of the syntax tree. To materialize a lazy qdag, we follow the recursive rules outlined earlier, utilizing traditional quadtrees with pointers to represent the output.

Definition 12 (Materialization of a lazy qdag)**.** The Materialization of $\mathcal{L} = (f, o)$, a lazy qdag, is a traditional quadtree representation (using pointers) that encodes the outcome of traversing and evaluating the syntax tree of $\mathcal{L}$.

For example, the materialization of the formula described in Equation 4.1, where the quadtrees Q_R, Q_S and Q_T are the ones shown in Figures 2.10 and 2.11, is the quadtree shown in Figure 4.3.

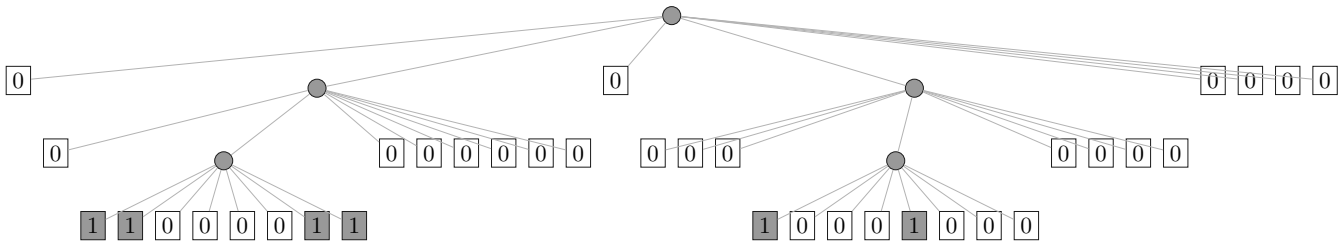
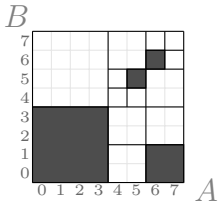


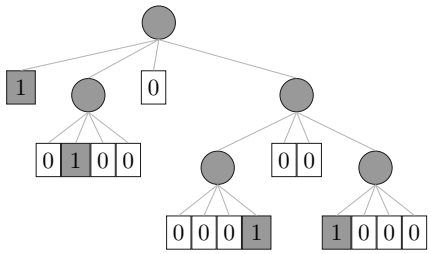
Figure 4.3: Full materialization of the lazy qdag described in Equation 4.1 on the quadtrees shown in Figures 2.10 and 2.11.

In traditional quadtrees, a leaf is either at the last level, representing a single cell with a value of zero or one, or at a higher level, representing an empty quadrant with a zero value. In the lazy version, we introduce a new concept: a *full leaf*, which represents a quadrant filled with ones (see Figure 4.4). This allows leaves to exist at higher levels of the tree, representing fully populated sub-grids. This generalization of the leaf concept enables the worst-case optimal evaluation of Boolean formulas including the `NOT` operator.

Definition 13 (Full leaf)**.** A full leaf is a leaf in a quadtree that represents a quadrant completely filled with ones.



(a) A grid representing the relation $R(A, B)$.



(b) Quadtree representation of the relation $R(A, B)$ with full leaves.

Figure 4.4

As we introduced a new concept of leaf, we need to redefine the value of a quadtree as follows:

- Return zero if the grid is empty.
- Return one if the grid is completely filled with ones (full leaf).
- Return ½ if the grid is partially filled with ones.

The updated definition of the value of a quadtree is shown in Algorithm 18.

Algorithm 18 Value on quadtrees for lqdags

Require: A quadtree (Q, M) with grid side l.
Ensure: Value 0 or 1 if the grid represented by Q is totally empty or full, respectively, otherwise ½.
1: **if** Q is a leaf **then return** the integer 0 or 1 associated with Q
2: **return** ½

4.3 Boolean algebra

To evaluate a lazy qdag, we apply a recursive approach based on the rules of its syntax tree. We determine whether the value of the lqdag is zero (indicating an empty quadrant), one (meaning a full quadrant), or $\diamondsuit$ (implying that further computation is necessary and we need to compute the value of its children). The detailed procedure for determining the value of a lazy qdag is presented in Algorithm 19. This approach ensures that we do not evaluate the entire formula unless necessary. For instance, if evaluating an OR operation and the left branch already yields 1, there is no need to evaluate the right branch, as the result is already determined to be 1.

Algorithm 19 VALUE ($\mathcal{L}$)

Require: An lqdag $\mathcal{L}$.
Ensure: Value of the root of $\mathcal{L}$.
1: **if** $\mathcal{L} = (\texttt{QTREE}, Q)$ **then return** VALUE(Q)
2: **if** $\mathcal{L} = (\texttt{NOT}, Q)$ **then return** $1 -$ VALUE(Q)
3: **if** $\mathcal{L} = (\texttt{AND}, \mathcal{L}_1, \mathcal{L}_2)$ **then**
4: **if** VALUE($\mathcal{L}_1 = 0$) or VALUE($\mathcal{L}_2 = 0$) **then return** 0
5: **if** VALUE($\mathcal{L}_1 = 1$) **then return** VALUE($\mathcal{L}_2$)
6: **if** VALUE($\mathcal{L}_2 = 1$) **then return** VALUE($\mathcal{L}_1$)
7: **return** $\diamond$
8: **if** $\mathcal{L} = (\texttt{OR}, \mathcal{L}_1, \mathcal{L}_2)$ **then**
9: **if** VALUE($\mathcal{L}_1 = 1$) or VALUE($\mathcal{L}_2 = 1$) **then return** 1
10: **if** VALUE($\mathcal{L}_1 = 0$) **then return** VALUE($\mathcal{L}_2$)
11: **if** VALUE($\mathcal{L}_2 = 0$) **then return** VALUE($\mathcal{L}_1$)
12: **return** $\diamond$
13: **if** $\mathcal{L} = (\texttt{EXTEND}, \mathcal{L}_1, \mathcal{A})$ **then**
14: **return** VALUE($\mathcal{L}_1$)
15: **if** $\mathcal{L} = (\texttt{LQDAG}, \mathcal{L}_1)$ **then return** VALUE($\mathcal{L}_1$)

Evaluating an lqdag $\mathcal{L}_F$, which represents the formula $\mathcal{F}$, results in the materialization of $\mathcal{L}_F$. The process for computing the materialization of a lazy qdag is described in Algorithm 20, and the running time of this algorithm is $\mathcal{O}(|Q_F^+|)$.

Algorithm 20 MATERIALIZE ($\mathcal{L}_F$)

Require: An lqdag $\mathcal{L}_F$ whose materialization represents a formula F over relations with d attributes.
Ensure: The materialization Q_F of $\mathcal{L}_F$.
1: **if** VALUE($\mathcal{L}_F$) $\in \{0, 1\}$ **then return** a leaf with value VALUE($\mathcal{L}_F$)
2: **for** $i \leftarrow 0, ..., 2^d - 1$ **do**
3: $C_i \leftarrow$ MATERIALIZE(CHILD($\mathcal{L}, i$))
4: **if** $\max\{\text{VALUE}(C_0), ..., \text{VALUE}(C_{2^d-1})\} = 0$ **then return** a leaf with value 0
5: **if** $\min\{\text{VALUE}(C_0), ..., \text{VALUE}(C_{2^d-1})\} = 1$ **then return** a leaf with value 1
6: **return** a quadtree with value ½ and children $C_0, ..., C_{2^d-1}$

To obtain the value of a node in the materialization of an lqdag, we navigate the syntax tree without fully evaluating the formula. For example, to retrieve the first quadrant of the result from a join query, we would navigate directly to the first child of the lqdag representing that join rather than evaluating the entire result immediately. To browse lazy qdags efficiently, we define the CHILD function, which returns the i-th child of a lazy qdag. The original CHILD function [6] is detailed in Algorithm 21. It is important to note that when we call the CHILD function, we also evaluate part of the formula, as it involves a call to the VALUE function.

Algorithm 21 CHILD $(\mathcal{L}, i)$

Require: An lqdag $\mathcal{L}(\mathcal{A})$ and an integer $0 \leq i < 2^{|\mathcal{A}|}$.
Ensure: An lqdag for the i-th child of $\mathcal{L}$.

if $\mathcal{L} = (\texttt{QTREE}, Q)$ **then return** $(\texttt{QTREE}, \text{CHILD}(Q, i))$
if $\mathcal{L} = (\texttt{NOT}, Q)$ **then return** $(\texttt{NOT}, \text{CHILD}(Q, i))$
if $\mathcal{L} = (\texttt{AND}, \mathcal{L}_1, \mathcal{L}_2)$ **then**
    if $\text{VALUE}(\mathcal{L}_1) = 1$ **then return** $\text{CHILD}(\mathcal{L}_2, i)$
    if $\text{VALUE}(\mathcal{L}_2) = 1$ **then return** $\text{CHILD}(\mathcal{L}_1, i)$
    return $(\texttt{AND}, \text{CHILD}(\mathcal{L}_1, i), \text{CHILD}(\mathcal{L}_2, i))$
if $\mathcal{L} = (\texttt{OR}, \mathcal{L}_1, \mathcal{L}_2)$ **then**
    if $\text{VALUE}(\mathcal{L}_1) = 0$ **then return** $\text{CHILD}\ (\mathcal{L}_2, i)$
    if $\text{VALUE}(\mathcal{L}_2) = 0$ **then return** $\text{CHILD}(\mathcal{L}_1, i)$
    return $(\texttt{OR}, \text{CHILD}(\mathcal{L}_1, i), \text{CHILD}(\mathcal{L}_2, i))$
if $\mathcal{L} = (\texttt{EXTEND}, \mathcal{L}_1(\mathcal{A}'), \mathcal{A})$ **then**
    $d \leftarrow |\mathcal{A}|$, $d' \leftarrow |\mathcal{A}'|$
    $m_d \leftarrow$ the d-bits binary representation of i
    $m_{d'} \leftarrow$ the projection of m_d to the positions in which the attributes of $\mathcal{A}'$ appear in $\mathcal{A}$
    $i' \leftarrow$ the value in $[0, 2^{d'} - 1]$ corresponding to $m_{d'}$
    return $(\texttt{EXTEND}, \text{CHILD}(\mathcal{L}_1, i'), \mathcal{A})$
if $\mathcal{L} = (\texttt{LQDAG}, \mathcal{L}_1)$ **then return** $\text{CHILD}(\mathcal{L}_1, i)$

For example, the materialization of the lqdag from Equation 4.1 after consulting all the children of the root is represented in Figure 4.5. We can see that the second and fourth children are represented by $\diamond$ nodes, as they need further computation (we will need to evaluate its children if we want to have more information about these nodes).

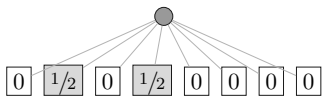


Figure 4.5: Materialization (not completed) of the lazy qdag described in Equation 4.1 after consulting the children of the root.

The main problem of Algorithm 21 is that it recreates the entire formula for each child, leading to unnecessary duplication of work and resource wastage. We can see in Algorithm 21 that the output is an entirely new lazy qdag, where we may recreate the exact internal nodes of the syntax tree and only modify the leaves: the pointer to the child of the quadtrees. Additionally, retrieving information about a quadtree, such as the number of attributes,

requires traversing the entire syntax tree to reach a leaf. This process is inefficient, as it necessitates evaluating parts of the formula to obtain this type of information that might be needed later.

To illustrate, assume we want to obtain the output of the second quadrant of the join between the relations R, S, and T (represented as the quadtrees shown in Figures 2.10 and 2.11), that is, $\textsc{child}(\mathcal{L}_{R \bowtie S \bowtie T}, 1)$ [1]. In that case, we need to descend through the syntax tree to obtain the second child of the materialization of $\mathcal{L}_{R \bowtie S \bowtie T}$, that is, the second quadrant of the quadtree representing the formula in Equation 4.1. The formula in Equation 4.3 represents the call to the CHILD function.

$$
\begin{aligned}
\textsc{child}(\mathcal{L}_{R \bowtie S \bowtie T}, 1) = \; & \textsc{child}(\\
& \quad \texttt{AND}(\\
& \quad\quad \texttt{AND}(\\
& \quad\quad\quad \texttt{EXTEND}(\texttt{QTREE}(Q_R), \mathcal{A}), \\
& \quad\quad\quad \texttt{EXTEND}(\texttt{QTREE}(Q_S), \mathcal{A})), \\
& \quad\quad \texttt{EXTEND}(\texttt{QTREE}(Q_T), \mathcal{A})), \\
& \quad 1)
\end{aligned}
\tag{4.3}
$$

With this version of CHILD, we will recreate and return the formula for the child. The process to generate the output for this example is as follows:

$$
\begin{aligned}
& \textsc{child}(\mathcal{L}_{R \bowtie S \bowtie T}, 1) \\
& \quad = \; \texttt{AND}(\textsc{child}(\mathcal{L}_{R \bowtie S}, 1), \\
& \qquad\quad \textsc{child}(\mathcal{L}_{\texttt{EXTEND}_T}, 1)) \\
& \quad = \; \texttt{AND}(\texttt{AND}(\textsc{child}(\mathcal{L}_{\texttt{EXTEND}_R}, 1), \\
& \qquad\quad\quad \textsc{child}(\mathcal{L}_{\texttt{EXTEND}_S}, 1)), \\
& \qquad\quad \texttt{EXTEND}(\textsc{child}(\mathcal{L}_T, 1), \mathcal{A})) \\
& \quad = \; \texttt{AND}(\texttt{AND}(\texttt{EXTEND}(\textsc{child}(\mathcal{L}_R, 1), \mathcal{A}), \\
& \qquad\quad\quad \texttt{EXTEND}(\textsc{child}(\mathcal{L}_S, 0), \mathcal{A})), \\
& \qquad\quad (\texttt{QTREE}, \texttt{EXTEND}(\boxed{0}\ \boxed{1}\ \boxed{0}\ \boxed{1}, \mathcal{A}))) \\
& \quad = \; \texttt{AND}(\texttt{AND}((\texttt{QTREE}, \texttt{EXTEND}(\boxed{0}\ \boxed{1}\ \boxed{0}\ \boxed{0}, \mathcal{A})), \\
& \qquad\quad\quad (\texttt{QTREE}, \texttt{EXTEND}(\boxed{1/2}\ \boxed{0}\ \boxed{0}\ \boxed{0}, \mathcal{A}))), \\
& \qquad\quad (\texttt{QTREE}, \texttt{EXTEND}(\boxed{0}\ \boxed{1}\ \boxed{0}\ \boxed{1}, \mathcal{A})))
\end{aligned}
$$

[1] The index starts with 0

It is important to note that the quadtrees illustrated in the example are pointers to a subtree of the quadtree. In the end, `CHILD` will return an entirely new lazy qdag, and it will rewrite the entire syntax tree, not only the leaves that are the ones that were modified.

Figure 4.6 illustrates the process of obtaining the **first** and the **second** child. We can observe that evaluating the entire formula is not necessary for the first child; specifically, evaluating $\texttt{EXTEND}(\texttt{QTREE}(Q_S), \mathcal{A})$ and $\texttt{EXTEND}(\texttt{QTREE}(Q_T), \mathcal{A})$ is unnecessary because the intersection of $\texttt{EXTEND}(\texttt{QTREE}(Q_R), \mathcal{A})$ and $\texttt{EXTEND}(\texttt{QTREE}(Q_S), \mathcal{A})$ already results in an empty quadrant. The blue and red nodes in the figure highlight the parts of the formula that need to be evaluated to retrieve the second child. Additional values are included in the figure solely for illustration purposes and are not required for the computation of the second child.

In Figure 4.7, we present the materialization of the lazy qdag after evaluating the formula for all the children of the root node.

New version of lqdags. We modified the definition of lazy qdags to improve efficiency by avoiding copying the formula for each child node. Instead of recreating the entire syntax tree for every child, we separate the information of the original lqdag into four parts. This allows us to store only the necessary changes on the leaves (a quadtree or another lazy qdag) while maintaining a pointer to the parent node's formula.

The new data structure consists of four fields:

1. The first part consists of the formula of the lqdag, i.e., the syntax tree as defined in Definition 4.2, without storing the quadtrees or lqdags as leaves.
2. The second part is an array that stores the leaves of the syntax tree (quadtrees or lqdags).
3. The third part is an array indicating the positions in the quadtrees; if a leaf is an lqdag, this part remains empty.
4. The fourth part holds the current materialization of the lqdag. It includes the value of that node, its position in the materialized quadtree (level and coordinates), and an array of pointer to lqdags representing its children. If a child is not yet materialized, the pointer is null; otherwise, it points to another lqdag with the same formula and array of quadtrees or lqdags.

The first and second fields remain constant for all children of a node and recursively throughout its descendants, while the third and fourth fields vary for each child. This design allows us to access crucial information more efficiently, such as the number of attributes, grid size, and the number of children, without the need to traverse the entire formula to reach the quadtree or lqdag leaves.

In Figure 4.8, we show the new version of the lqdag for Equation 4.3 (second child of $\mathcal{L}_{R \bowtie S \bowtie T}$). The coordinates in *position* and *materialization* fields represent the starting position of the quadrant. When we reach the final level of the qdags, these coordinates indicate the exact point corresponding to the result of evaluating that part of the formula.

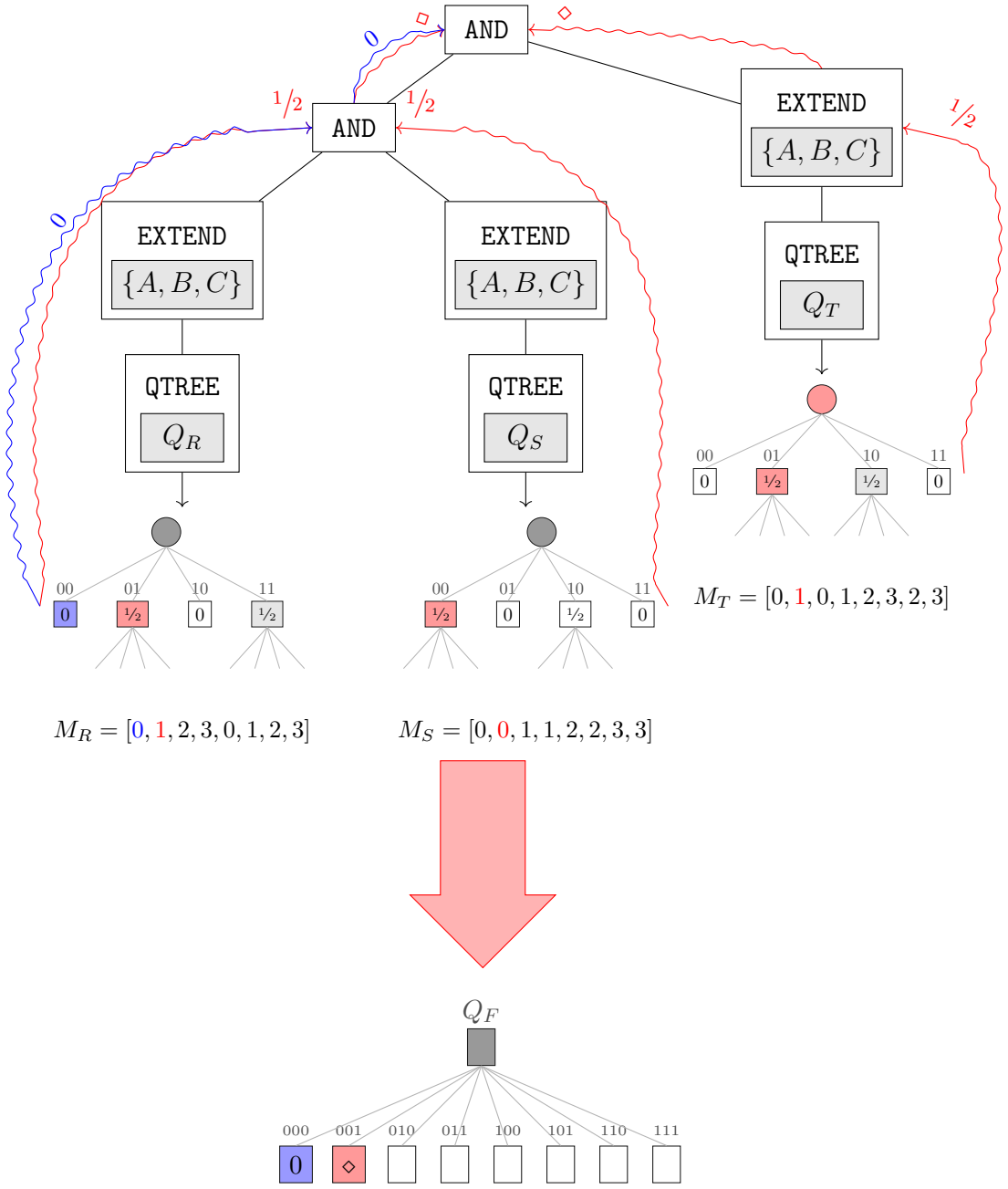


Figure 4.6: Join $R \bowtie S \bowtie T$. *At the top, we illustrate the traversal of the syntax tree of the join* $R \bowtie S \bowtie T$ *to get the first and the second child. We show the mapping function of each qdag at the leaves of the syntax tree. At the bottom, we show the materialization of the lazy qdag after computing the second child. Note that we did not need to evaluate the extension of the relation* T.

This approach enables efficient navigation through the syntax tree without duplicating and storing the original formula. Additionally, each lazy qdag includes supplementary information, such as the number of attributes, grid size, and the number of children. This extra information facilitates the retrieval of relevant details and further enhances navigation.

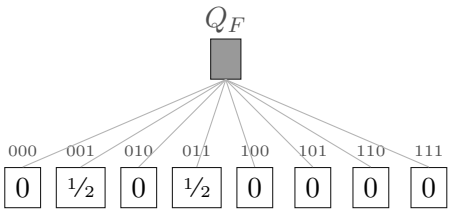


Figure 4.7: Result of materializing all the children of the root of the join $R \bowtie S \bowtie T$.

Lqdag as a leaf To extend our functionalities even more, we can use lqdags as leaves instead of qdags. This way, we can compute more complicated formulas and share intermediate results. For example, if we want to compute the union of $\mathcal{L}_{R \bowtie S \bowtie T}$ and a qdag Q_U, that represents the relation $U(A, B, C)$, we can use an lqdag and a qdag as leaves. We illustrate this lqdag in Figure 4.9.

Worst-case optimality In Section 2.6, we showed that the full join query $J = R_1 \bowtie ... \bowtie R_n$ over a database D is computed in time $\mathcal{O}(2^{\rho^*(J,D)} \cdot 2^d n \log \min(l, S)) = \tilde{\mathrm{O}}(2^{\rho^*(J,D)})$. Let us define $\mathcal{F}$, a $\mathcal{J}UC$ query [2], on the relations $\{R_1, ..., R_n\}$ over a database D of the domain $[0, l-1]$, $S = \sum_i |\mathcal{A}_i|$, and $\mathcal{F}(D)^*$ the maximum size of the output of $\mathcal{F}$ over instances D' with relations $R'_1, ..., R'_n$ of respective sizes $|R'_i| = \mathcal{O}(|R_i|)$. We can evaluate this formula in $\mathcal{O}(\mathcal{F}(D)^* \cdot 2^d |F| \log \min(l, S)) = \tilde{\mathrm{O}}(\mathcal{F}(D)^*)$, which indicates that the algorithm is worst-case optimal in terms of data complexity [6].

4.4 Full relational algebra

As we saw in the previous section, we can construct more complex formulas by combining simpler ones using the syntax tree. We will now extend lqdags to handle the full relational algebra, including selection and θ-join operations, as well as projection and its derivatives. It is important to note that, unlike Boolean formulas, we can no longer guarantee worst-case optimality for these extensions.

Definition 14 (Relational algebra formula)**.** A Relational algebra formula $\mathcal{F}$ is composed by selection, projection and the Cartesian product operations over relations.

Selection and θ-join

We use selection to retrieve a subset of tuples that satisfy certain condition(s). The predicate is a logical expression on the attributes of F and acts as a filter for the results.

Definition 15 (Selection)**.** The selection operation (or restriction) is defined as a function $\sigma_\theta(F)$ that takes a subexpression F and a predicate θ, where F is a formula over the relation

[2] A formula composed by Join, Union and Complement operations, that is, the Boolean algebra.

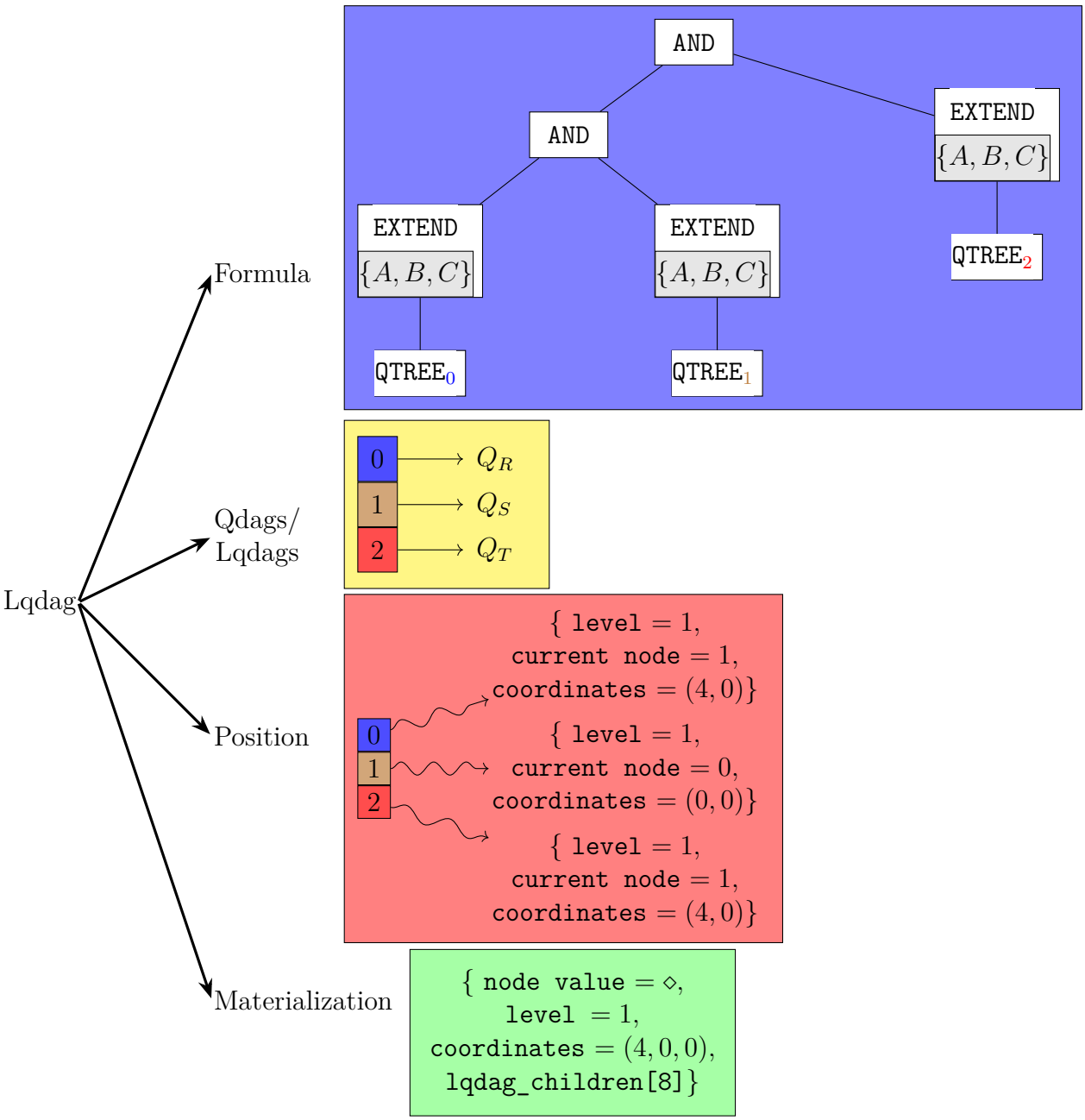


Figure 4.8: New version of lqdags. *We show the lqdag of the second child of $\mathcal{L}_{R \bowtie S \bowtie T}$. Formula is the syntax tree of the lqdag, Qdags is an array with the leaves of the syntax tree, Position is an array with the position in the quadtrees, and Materialization is the current materialization of the lqdag.*

$R(\mathcal{A})$. The predicate can involve any binary comparison operation such as $=, \neq, <, >, \leq$, and $\geq$, applied to attributes of the relation R or values within that domain [46].

In lqdags, the selection function operates on an lqdag $\mathcal{L}_F$ whose materialization is Q_F, returning another lqdag $\mathcal{L}_{F'}$ whose materialization is $Q_{F'}$, where $|Q_{F'}| \leq |Q_F|$. The selection operation can be defined as follows:

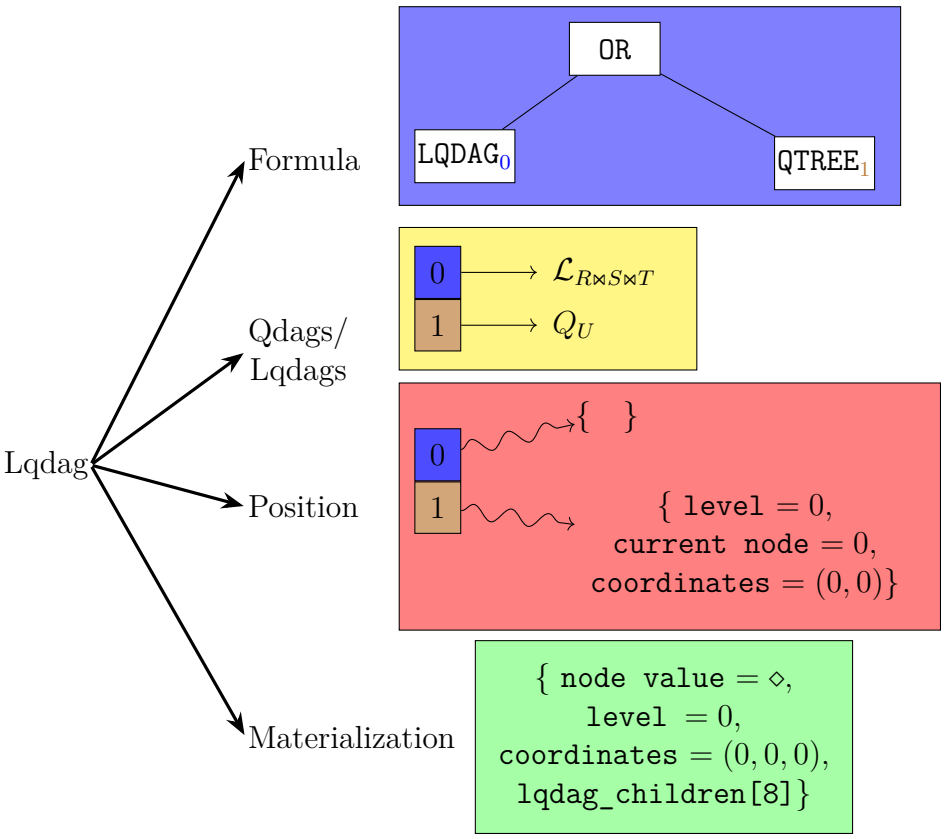


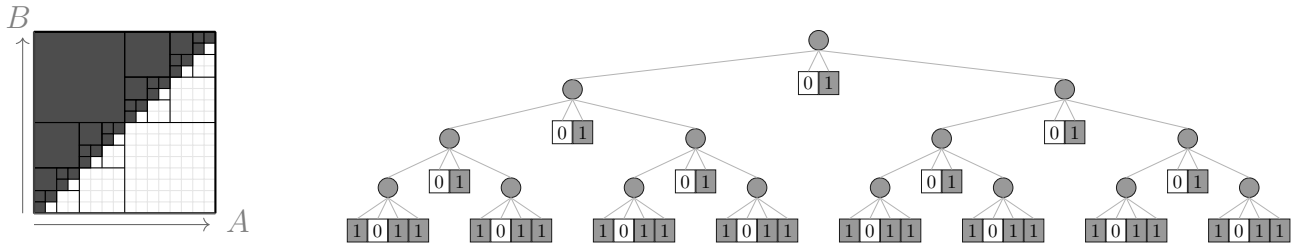
Figure 4.9: Union between $\mathcal{L}_{R\bowtie S\bowtie T}$ and Q_U. *We show the lqdag of the union between a qdag and another lqdag.*

$$(\texttt{SELECT}, \mathcal{L}_F(\mathcal{A}), \theta) = (\texttt{AND}, \mathcal{L}_F(\mathcal{A}), pred(\theta, \mathcal{A})) \tag{4.4}$$

where $pred(\theta, \mathcal{A})$ represents a virtual lqdag (never materialized) that has the same attributes $\mathcal{A}$ and includes only those cells that satisfy the predicate θ. The AND operation is then used to combine the two lqdags, filtering the results according to the specified predicate.

The predicate is a logical expression using the operators $=, \neq, <, >, \leq$, and $\geq$, as defined. These operators can be applied either over a pair of attributes (e.g., $A < B$) or over a pair consisting of an attribute and a constant within that attribute's domain (e.g., $B \geq 5$).

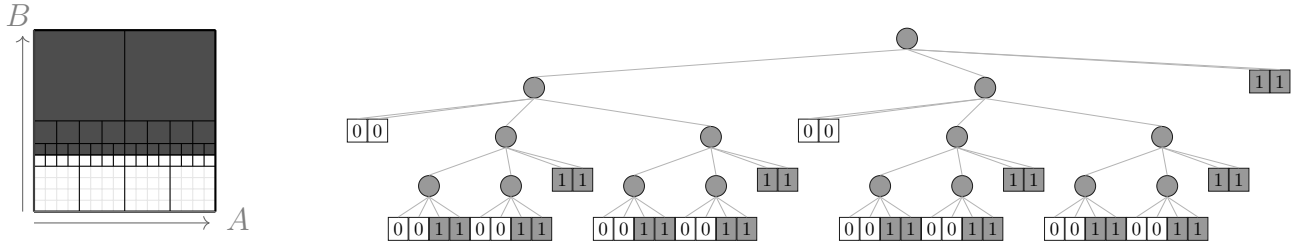
The predicate represents a virtual lqdag that can be defined as $(\texttt{QTREE}, Q_\theta)$. Instead of building the quadtree, we simulate the navigation through the lazy qdag $(\texttt{QTREE}, Q_\theta)$ and compute the value directly. Figures 4.10 and 4.11 show two examples of predicates and their corresponding quadtree representations (if we materialize them). In Algorithm 22, we show how to do this [6].



(a) Grid with the values of A and B.

(b) Q_θ representing the predicate $A < B$.

Figure 4.10: An illustration of the quadtree for the predicate $A < B$.



(a) Grid with the values of A and B.

(b) Q_θ representing the predicate $B \geq 5$.

Figure 4.11: An illustration of the quadtree for the predicate $B \geq 5$.

Algorithm 22 VALUE OF Q_θ

Require: A predicate θ and a grid of side l.

Ensure: Value 1 or 0 if the quadtree satisfies or not the predicate for every cell (or for any cell) of the grid. Otherwise ½.

1: **if** θ does not hold for any cell in the grid **then return** the integer 0

2: **if** θ holds for every cell in the grid **then return** the integer 1

3: **return** ½

We define the predicate as the following object:

$$\begin{aligned} \texttt{predicate} = \{&\texttt{att}_1, \\ &\texttt{att}_2, \\ &\texttt{constant}, \\ &\texttt{operator}\} \end{aligned} \tag{4.5}$$

The process of computing the value of a predicate operates in $\mathcal{O}(|\theta|)$ time because each logical expression that compares attributes and constants using the defined operators can be evaluated in constant time for entire subgrids, as we can see in Algorithm 23. This algorithm takes the predicate θ, the coordinates (`coord`) of the starting point where the subquadrant begins, the size of the quadrant (`quadrant_side`) being evaluated, and the number of attributes (`nAtt`).

For example, if `nAtt` $= 2$, `quadrant_side`$= 4$, and `coord` $= (0, 4)$, the algorithm evaluates the subgrid that goes from $(0, 4)$ to $(3, 7)$. Although not all operators are explicitly listed, they follow a similar logic, and the algorithm has been implemented for all the binary operators mentioned earlier.

Additionally, we can perform a θ-join, which involves selecting tuples from the Cartesian product (cross join) of two relations that could or could not share attributes to combine data from different relations. The Cartesian product concatenates the tuples from relation R_1 with the tuples from relation R_2. Then, the θ-join is a binary operation that selects the tuples from this new relation resulting from the Cartesian product between R_1 and R_2 and satisfies the predicate θ. This operation is defined as follows:

$$(\texttt{THETAJOIN}, \mathcal{L}_1, \mathcal{L}_2, \theta) = (\texttt{SELECT}(\texttt{JOIN}, \mathcal{L}_1, \mathcal{L}_2), \theta) \tag{4.6}$$

where:

$$(\texttt{JOIN}, \mathcal{L}_1(\mathcal{A}_1), \mathcal{L}_2(\mathcal{A}_2)) = (\texttt{AND}, (\texttt{EXTEND}, \mathcal{L}_1, \mathcal{A}_1 \cup \mathcal{A}_2), (\texttt{EXTEND}, \mathcal{L}_2, \mathcal{A}_1 \cup \mathcal{A}_2)) \tag{4.7}$$

Worst-case optimality The AGM bound for a formula $\mathcal{L}_F = (\texttt{AND}, (\texttt{QTREE}, R), (\texttt{QTREE}, Q_\theta))$ is $F(D)^* = \min(|R|, |Q_\theta|)$. A general selection formula is defined as $\mathcal{L}_F = (\texttt{SELECT}, \mathcal{L}, \theta)$. We saw that Algorithm 22 operates in time $\mathcal{O}(|\theta|)$. If we include this cost as part of $|F|$, then the output of the selection formula can be computed in time $\mathcal{O}(|Q_F^+| \cdot 2^d |F|)$. As a result, since $|Q_F^+|$ is bounded by $F(D)^*$, we can achieve worst-case optimal performance when the predicate can be pushed down to the leaves of the syntax tree, i.e., when the predicate selects tuples from a single relation. However, we can no longer guarantee worst-case optimality when the predicates combine two or more tables [6].

Similarly, evaluating a θ-join formula using lqdags is not worst-case optimal due to the join predicates involving attributes from different relations.

Projection and derivatives

We use the projection operation to retrieve only a subset of attributes from a relation. This function takes a relation and a list of attributes as input and produces a new relation that includes only the specified attributes. The resulting relation contains a subset of the original tuples, ensuring each tuple appears only once by eliminating duplicates.

Definition 16 (Projection)**.** In relational algebra, a projection is a function $\pi_{\mathcal{A}'} : R(\mathcal{A}) \to R(\mathcal{A}')$, where $\mathcal{A}' \subseteq \mathcal{A}$ and takes a relation and a list of attributes, and outputs another relation with the specified attributes. A tuple will be included in the projection if there is an extension of that tuple in the original relation. For example, $R(A, B, C) = \{(0, 1, 1), (2, 5, 4), (0, 1, 3)\}$, $\mathcal{A}' = \{A, B\}$, $\pi_{\mathcal{A}'}(R) = \{(0, 1), (2, 5)\}$ [46].

Let F be a formula and its lazy qdag $L_F(\mathcal{A})$. The lazy qdag of the projection is defined as follows:

Algorithm 23 EVALUATION OF θ

Require: A predicate θ, the coordinates `coord` of the quadrant that represents that node, the size of the quadrant represented by the node `quadrant_side`, and the number of attributes `nAtt`.

Ensure: Value 1 or 0 if the quadtree satisfies or not the predicate for every cell (or for any cell) of the grid. Otherwise ½.

```
1: for i in nAtt do
2:     min_att[i] = coord[i]
3:     max_att[i] = coord[i] + quadrant_side −1
4: if θ is applied over a pair of attributes then
5:     min_att_1 = min_att[θ.att_1]
6:     max_att_1 = max_att[θ.att_1]
7:     min_att_2 = min_att[θ.att_2]
8:     max_att_2 = max_att[θ.att_2]
9:     switch θ.operator do
10:        case EQUAL
11:            if quadrant_side == 1 then return (min_att_1 == min_att_2)
12:            else if min_att_1 == min_att_2 then return 1/2
13:            else
14:                return 0
15:        //... other operators
16:        case GREATER
17:            if quadrant_side == 1 then return (min_att_1 > min_att_2)
18:            else if min_att_1 > max_att_2 then return 1
19:            else if max_att_1 > min_att_2 then return 1/2
20:            else
21:                return 0
22: else //θ is applied over an attribute and a constant
23:    switch θ.operator do
24:        case EQUAL
25:            if quadrant_side == 1 then return (min_att_1 == θ.constant)
26:            else
27:                if min_att_1 ≤ θ.constant and max_att_1 ≥ θ.constant then return 1/2
28:                else
29:                    return 0
30:        //... other operators
31:        case GREATER
32:            if quadrant_side == 1 then return (min_att_1 > θ.constant)
33:            else if min_att_1 > θ.constant then return 1
34:            else if max_att_1 ≥ θ.constant then return 1/2
35:            else
36:                return 0
```

$$\mathcal{L} = (\texttt{PROJECT}, \mathcal{L}_F(\mathcal{A}), \mathcal{A}') \tag{4.8}$$

where $|\mathcal{A}| = d, |\mathcal{A}'| = d', \mathcal{A}' \subseteq \mathcal{A}$.

To begin, we define how to compute the value of the projection formula. Algorithm 24 describes this process. This algorithm operates in $\mathcal{O}(1)$ time because the value of the projection formula is directly derived from the value of the formula $\mathcal{L}_F$.

Algorithm 24 VALUE OF $\mathcal{L} = (\texttt{PROJECT}, \mathcal{L}_F(\mathcal{A}), \mathcal{A}')$

Require: A formula $\mathcal{L}_F$ of the relation $R(\mathcal{A})$ and a list of attributes $\mathcal{A}'$.
Ensure: Value 0 or 1 if the value of the original formula $\mathcal{L}_F$ is 0 or 1. Otherwise $\diamond$.
1: **if** VALUE($\mathcal{L}_F$) is 0 or 1 **then return** VALUE($\mathcal{L}_F$)
2: **return** $\diamond$

To compute the materialization of the quadtree $Q_{\mathcal{L}}$, we need to determine the children of the nodes. This involves performing an `OR` operation between the children that share the same values for the attributes in $\mathcal{A}'$. In Figure 4.12, we illustrate how to divide the qdag, and in Example 6, we demonstrate how to project $\mathcal{L}$ onto attributes A or B.

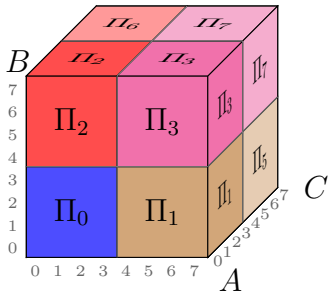


Figure 4.12: Illustration of how we divide the qdag into eight quadrants to recursively apply projection.

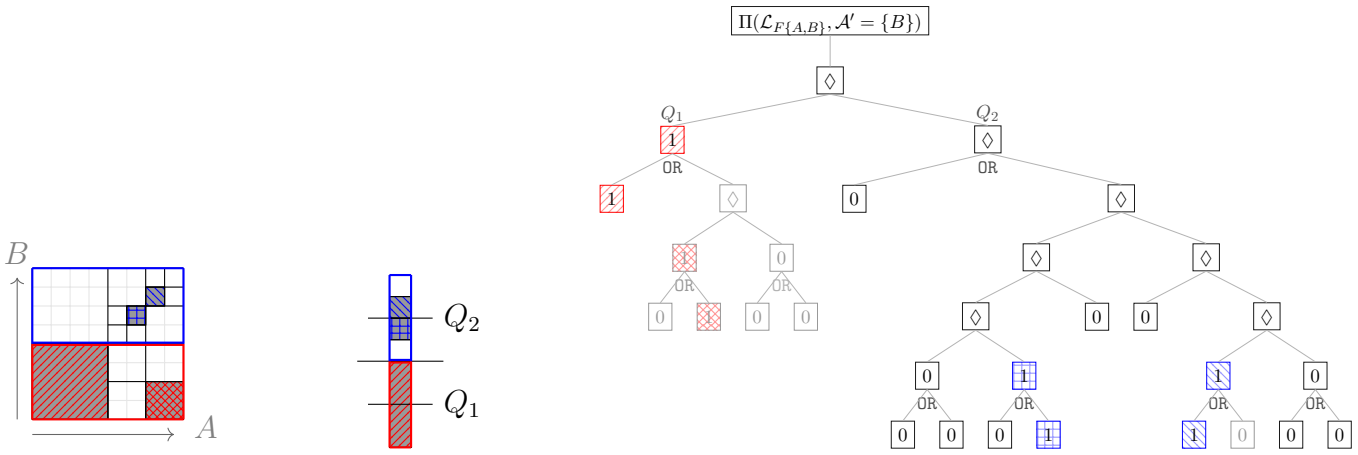
Example 6 (Projection to A and B)**.** For example, if VALUE($\mathcal{L}_F(\{A, B\})) \notin \{0, 1\}$, and $\mathcal{L} = (\texttt{PROJECT}, \mathcal{L}_F, \{A\})$, then:

$$\text{CHILD}(\mathcal{L}, 0) = (\texttt{PROJECT}, (\texttt{OR}, \text{CHILD}(\mathcal{L}_F, 0), \text{CHILD}(\mathcal{L}_F, 2)), \{A\})$$
$$\text{CHILD}(\mathcal{L}, 1) = (\texttt{PROJECT}, (\texttt{OR}, \text{CHILD}(\mathcal{L}_F, 1), \text{CHILD}(\mathcal{L}_F, 3)), \{A\})$$

For $\mathcal{L} = (\texttt{PROJECT}, \mathcal{L}_F, \{B\})$, it would be:

$$\text{CHILD}(\mathcal{L}, 0) = (\texttt{PROJECT}, (\texttt{OR}, \text{CHILD}(\mathcal{L}_F, 0), \text{CHILD}(\mathcal{L}_F, 1)), \{B\})$$
$$\text{CHILD}(\mathcal{L}, 1) = (\texttt{PROJECT}, (\texttt{OR}, \text{CHILD}(\mathcal{L}_F, 2), \text{CHILD}(\mathcal{L}_F, 3)), \{B\})$$

We can see an illustration of the projection of $\mathcal{L} = (\texttt{PROJECT}, \mathcal{L}_F, \{B\})$ in Figure 4.13. The full evaluation of this operation is shown in Figure 4.14



(a) Relation $R(A, B)$.

(b) Projection on B.

(c) $\mathcal{L} = (\texttt{PROJECT}, \mathcal{L}_F, \{B\})$. Syntax tree of the projection with its non-pruned quadtree. *The nodes that are not materialized are shown in grey, just for illustration.*

Figure 4.13: Projection of $\mathcal{L}_F(A, B)$ onto B.

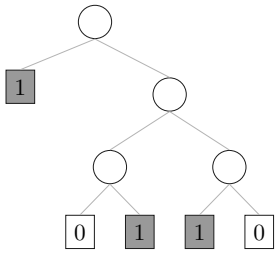


Figure 4.14: Full materialization of projection $\mathcal{L} = (\texttt{PROJECT}, \mathcal{L}_F, \{B\})$ from Figure 4.13.

In the original version of lqdags [6], the formula would grow exponentially as we descend through the syntax tree, as shown in Figure 4.13. We implement the projection operation based on the new version of lqdags (see Figure 4.8), where the formula remains the same across all children. However, the materialized children still grow exponentially because evaluating them involves a recursive `OR` operation.

Figures 4.15 and 4.16 illustrate this new projection implementation. In the first figure, we show the lqdag for the result of projecting the relation $R(A, B)$ onto A. In the second figure, we show the lqdag of the projection of the relation $R(A, B, C)$ onto B.

When we project a relation from $|\mathcal{A}|$ to $|\mathcal{A}'|$ attributes, we map the original children to their projections by performing an `OR` operation. For example, in Figure 4.15, the first child of the projection of the relation $R(A, B)$ to $R(A)$ is achieved by performing an `OR` between the first and third children of the original relation. The second child results from an `OR` between the second and fourth children. In Figure 4.16, where we project a relation of three attributes ($R(A, B, C)$) to just one attribute ($R(B)$), the first child is formed by an `OR` of the first four children, and the second child by an `OR` between the last four children. The value of a child projection is outlined in Algorithm 25. It shows that the calculation involves a multi-`OR` operation among certain children of the lqdag.

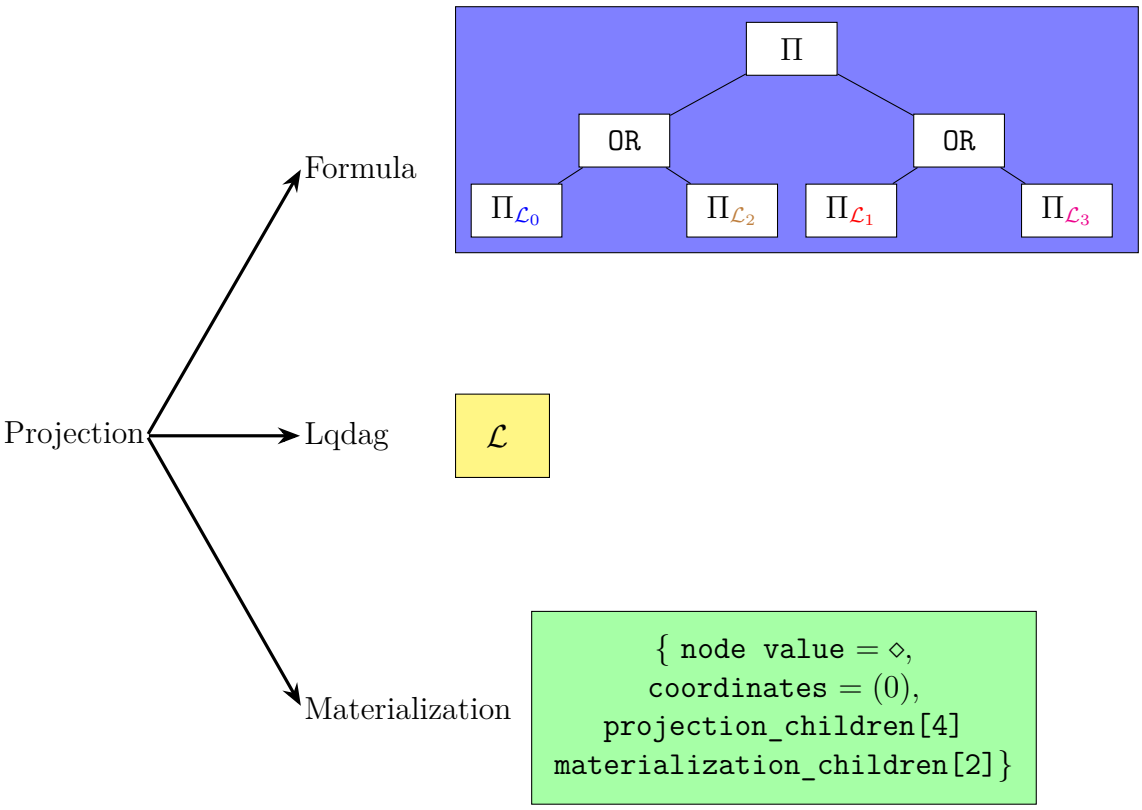


Figure 4.15: Projection $\mathcal{L}(\{A\}) = (\texttt{PROJECT}, \mathcal{L}_F(\{A, B\}), \{A\})$. *We can see in the syntax tree that we will have four leaves of projections, each corresponding to the projection of the first quadrant, the third, the second and the fourth, respectively. In the materialization, we will have a node with its coordinates, a pointer to the projection children if the value of the node is ⋄, and a pointer to its two children if the value of the node is ⋄.*

Algorithm 25 VALUE of Child of $\mathcal{L} = (\texttt{PROJECT}, \mathcal{L}_F(\mathcal{A}), \mathcal{A}')$

Require: A formula $\mathcal{L}_F$ of the relation $R(\mathcal{A})$ and a list of attributes $\mathcal{A}'$, i representing the i-th child. Additionally, a list of indexes $\mathcal{J}$ identifies the corresponding children to perform the `OR` operation to obtain the i-th child.

Ensure: Value 0 if all the children are empty, 1 if one lqdag is a full leaf or ⋄ otherwise.

```
1: m ← 0
2: for j ← J do
3:     val_j ← VALUE(CHILD(L_F, j))
4:     if val_j = 1 then return 1
5:     m ← max(m,val_j)
6: if m = 0 then return 0
7: return ⋄
```

If the value of a child projection is represented as ⋄, we need further computation. This means we need to calculate the projections for the involved children. For instance, in the example illustrated in Figure 4.16, we must compute the projections of the first four children to proceed with the calculation of the first child's value. However, if the value is either 0 or 1, we do not need to compute the projections for the children. Algorithm 26 demonstrates

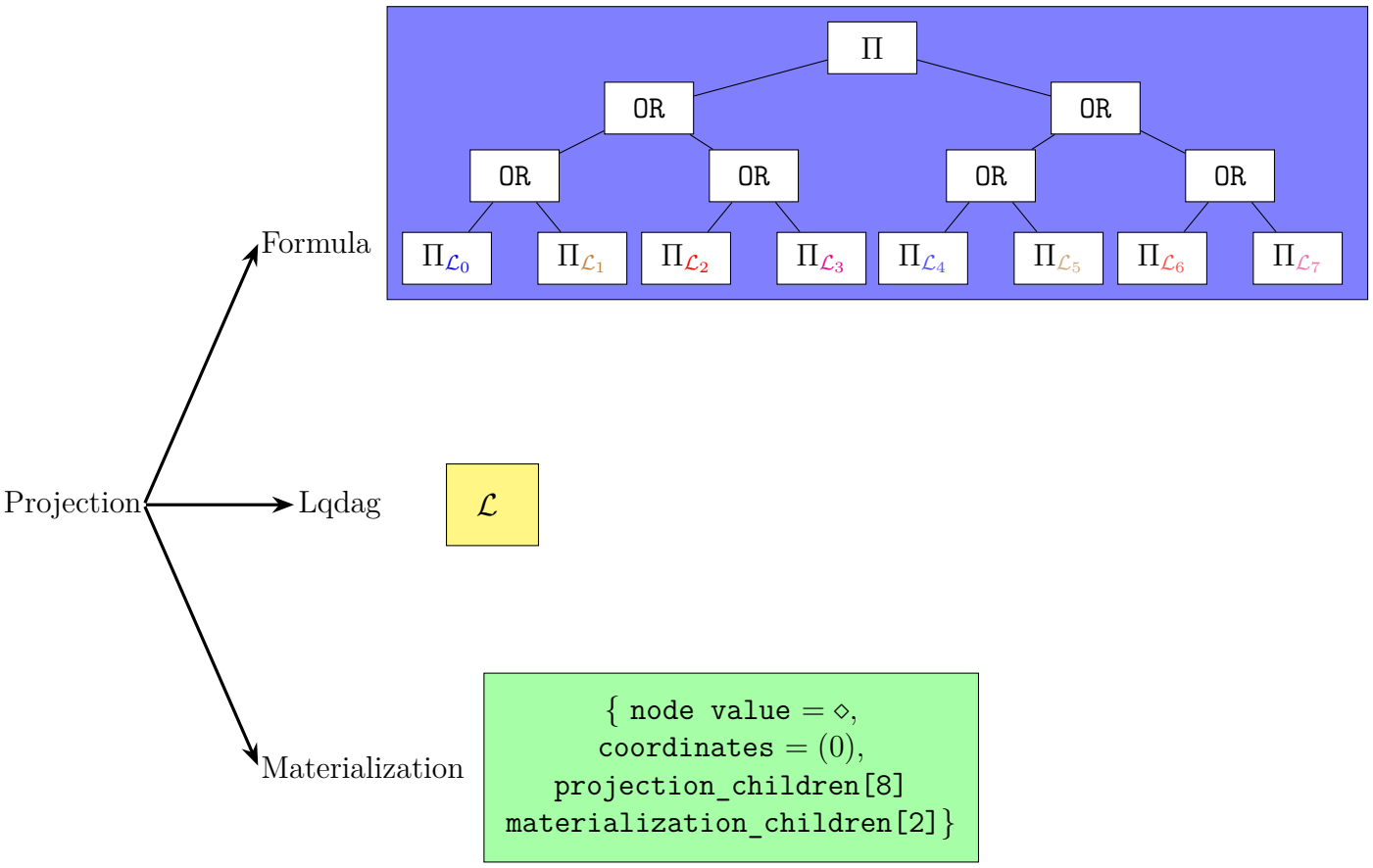


Figure 4.16: Projection $\mathcal{L}(\{B\}) = (\texttt{PROJECT}, \mathcal{L}_F(\{A, B, C\}), \{B\})$. *In this case, the syntax tree will have eight leaves of projections, each corresponding to one of the quadrants of the cube. In the materialization, we will have a node with its coordinates, a pointer to the projection children if the value of the node is ⋄, and a pointer to its two children if the value of the node is ⋄.*

how to compute the projection leaves of a child when its value is ⋄.

Algorithm 26 CHILD of $\mathcal{L} = (\texttt{PROJECT}, \mathcal{L}_F(\mathcal{A}), \mathcal{A}')$

Require: A formula $\mathcal{L}_F$ of the relation $R(\mathcal{A})$ and a list of attributes $\mathcal{A}'$, i representing the i-th child. Additionally, a list of indexes $\mathcal{J}$ identifies the corresponding children to perform the OR operation to obtain the i-th child.

Ensure: An array of projections to compute the i-th child of $\mathcal{L}$.

```
for j ← J do
    val_j ← VALUE(CHILD(L_F, j))
    if val_j = 0 then
        projections_i[j] ← null
    else
        projections_i[j] ← new projection
```

Projection also allows us to perform other relational algebra operations such as semijoin, antijoin, and division. The semijoin operation returns the tuples from the first relation that have a match in the second relation. The antijoin operation returns the tuples from the first relation that do not have a match in the second relation. The division operation returns the

tuples from the first relation that are not in the second relation, with the attributes from the first one that are not in the second one. These operations are defined as follows:

$$(\texttt{SEMIJOIN}, \mathcal{L}_1(\mathcal{A}_1), \mathcal{L}_2(\mathcal{A}_2)) = (\texttt{PROJECT}, (\texttt{JOIN}, \mathcal{L}_1, \mathcal{L}_2), \mathcal{A}_1)$$

$$(\texttt{ANTIJOIN}, \mathcal{L}_1(\mathcal{A}_1), \mathcal{L}_2(\mathcal{A}_2)) = (\texttt{PROJECT}, (\texttt{JOIN}, \mathcal{L}_1, (\texttt{NOT}, \mathcal{L}_2)), \mathcal{A}_1)$$

$$(\texttt{DIVISION}, \mathcal{L}_1(\mathcal{A}_1), \mathcal{L}_2(\mathcal{A}_2)) = (\texttt{DIFF}, \mathcal{L}'_1, (\texttt{PROJECT}, (\texttt{DIFF}, (\texttt{JOIN}, (\mathcal{L}'_1, \mathcal{L}_2), \mathcal{L}_1), \mathcal{A}_1 \setminus \mathcal{A}_2)))$$
$$\text{where } \mathcal{L}'_1 = (\texttt{PROJECT}, \mathcal{L}_1, \mathcal{A}_1 \setminus \mathcal{A}_2) \text{ and } \mathcal{A}_2 \subseteq \mathcal{A}_1$$

Worst-case optimality Projection is not worst-case optimal. In the worst case, we may need to traverse the entire quadtree to compute the `OR` operation between different children. Consequently, the lqdag $\mathcal{L}$ of the projection could grow as large as the lqdag of the original formula $\mathcal{L}_F$, even though the upper-bound on the output may be smaller. This is illustrated in Figure 4.13.

When projecting an lqdag from $|\mathcal{A}| = d$ dimensions to a d'-dimensional lqdag, the CHILD algorithm increases the evaluation by $2^{d-d'}$ nodes each time. Therefore, $|\mathcal{L}|$ cannot be considered a constant concerning data complexity.

4.5 Experimental results

As discussed in Section 3.6, we implemented all our algorithms in C++17 using the O3 optimization flag. We conducted the same experiments across all algorithms and data structures, utilizing the graph pattern outlined in Chapter 2.

In our **first experiment** we performed 50 different tests for each algorithm across various pattern queries (see Figure 2.15). The **second experiment** focused exclusively on tests that generated more than 1000 results. We did not test the relational algebra, so this additional experiment will remain as future work. All tests were carried out using a single thread on an Intel Xeon E5-2609 machine equipped with 128GB of DDR3 1066 ECC RAM and three SATA3 hard drives (2 x 1TB and 1 x 2TB).

Furthermore, to compare the lazy qdags join algorithm with the original algorithm, we modified the original join algorithm to build the compressed quadtree once we achieved k results. As in the previous experiment (see Section 3.6), we tested the sequential version of the original algorithm. Since the lazy qdags algorithm experienced a significant number of terminated processes in certain queries, we only included those queries that could be successfully completed by both algorithms. As a result, in the second experiment we will observe fewer patterns.

First scenario

In Figures 4.17 and 4.18, we plot the execution time and the number of tuples visited to retrieve $k = \{1, 10, 100, 1000\}$ results using both the lazy qdags and the original quadtree. Figure 4.17 shows that the original algorithm outperforms the lazy qdags across all patterns. Generally, the execution time for lazy qdags increases with k. In contrast, the original algorithm's execution time remains more stable as k increases.

In Figure 4.18, we observe that lazy qdags visit the same quantity of tuples than the original algorithm, as expected. Similar to the previous figure, the number of tuples visited increases as the value of k rises. For most patterns (J3, J4, P2, P3, S1, S2, S3, S4, T3, T4, Tr1, and Tr2), this increase is particularly noticeable between $k = 1$ and $k = 10$. However, for the pattern Ti2, a significant increase is observed between $k = 100$ and $k = 1000$.

Second scenario: join produces many results

As discussed in Section 3.6, we also evaluated our algorithms when the join generates a large number of results. In Figure 4.19, we plot the execution time required to retrieve the first $k = \{1, 10, 100, 1000\}$ results. Consistent with the previous scenario, the original algorithm demonstrates superior performance.

Notably, we observe a clear distinction as k begins to increase. When retrieving only a few results ($k < 100$), the performance of lqdags is similar to the original algorithm, specially for the patterns J3, J4, P2, S3, T2, T4, Ti2. But, as k begins to increase, the performance of lqdags deteriorates leading to a significant gap between the two algorithms.

In Figure 4.20, we observe that both algorithms visit the same number of tuples, as expected. The number of tuples visited begins to increase as k grows, but the growth becomes more pronounced for $k > 100$.

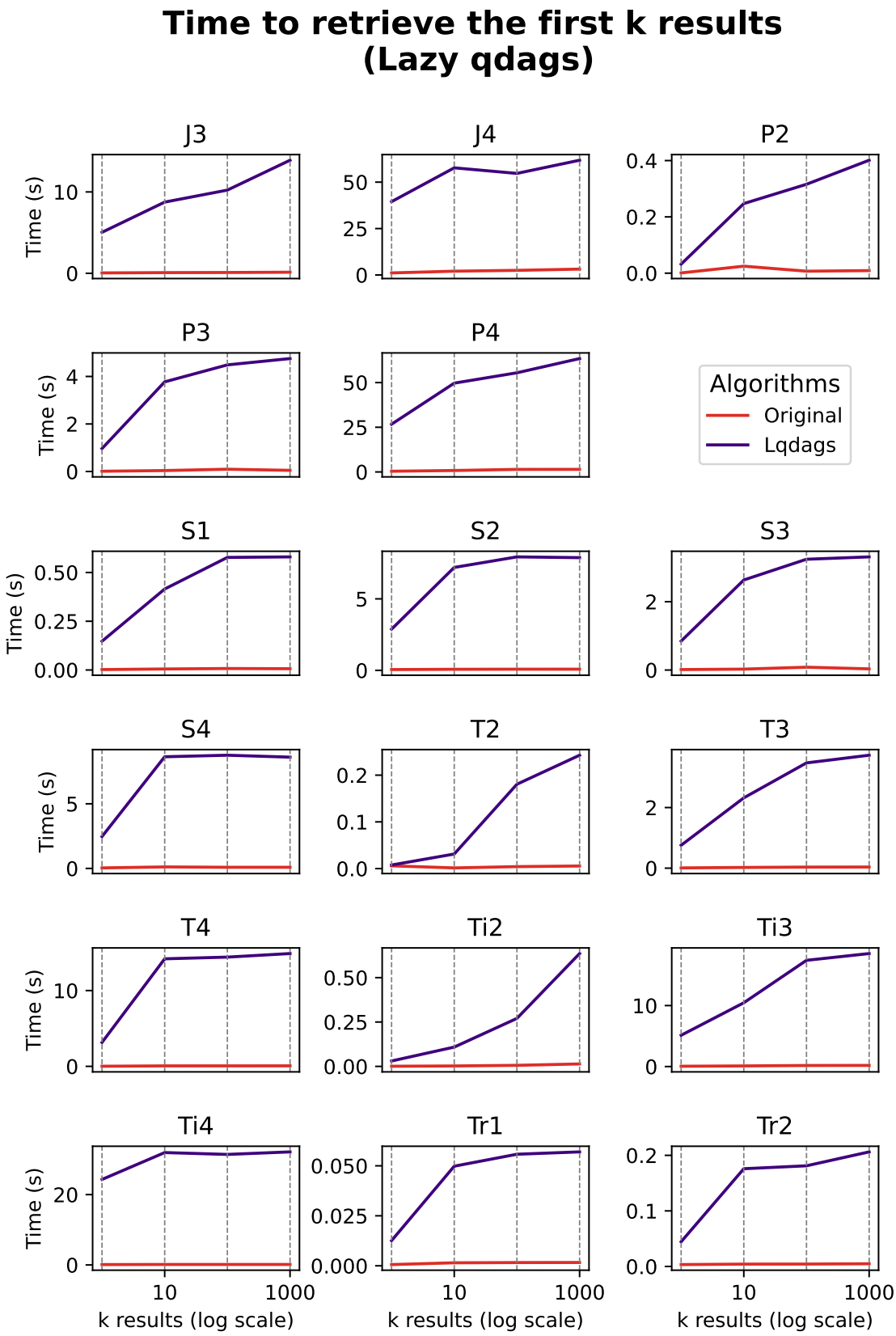


Figure 4.17: Time (in seconds) for lqdags to retrieve the first k results.

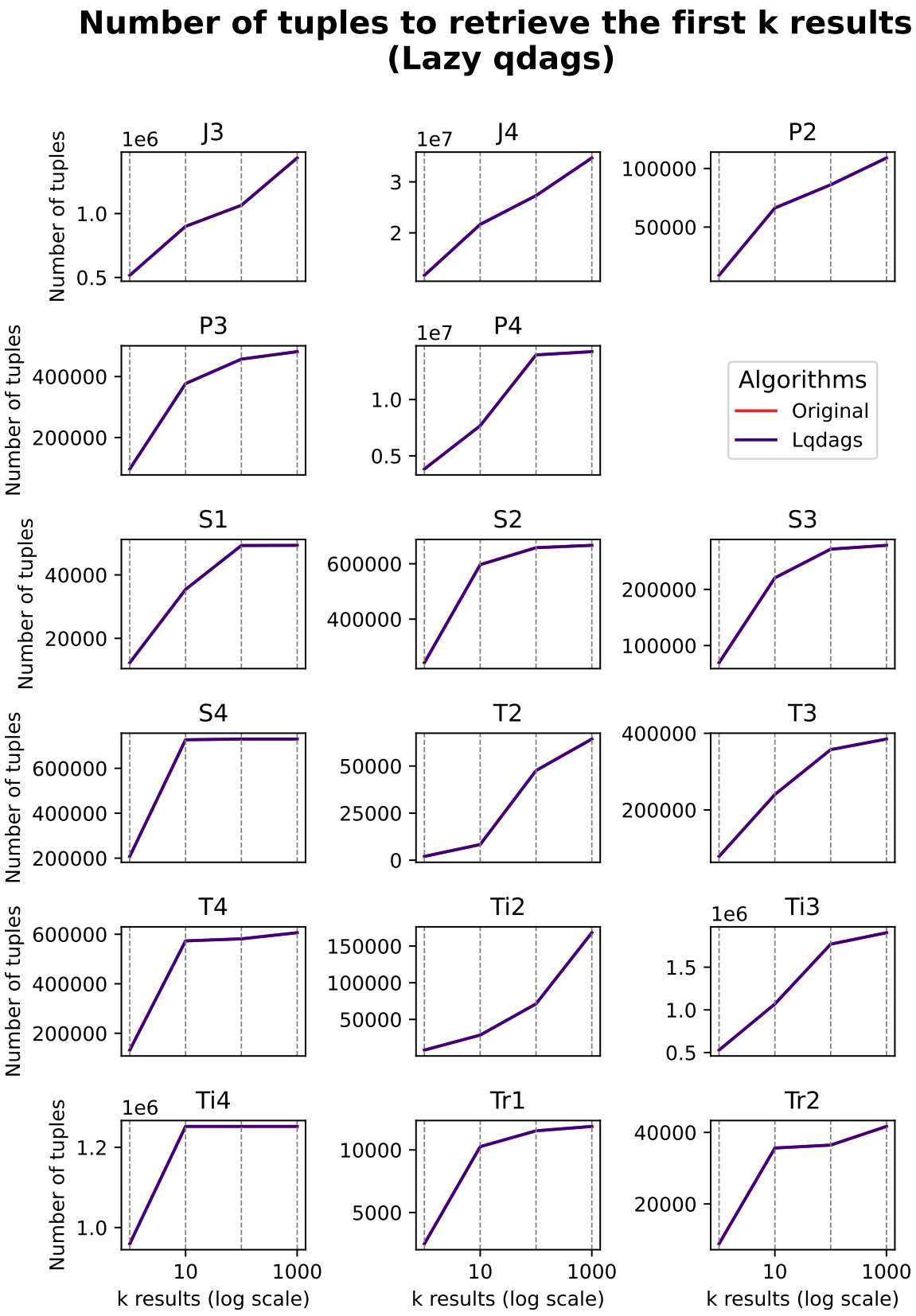


Figure 4.18: Number of tuples visited in lqdags to retrieve the first k results.

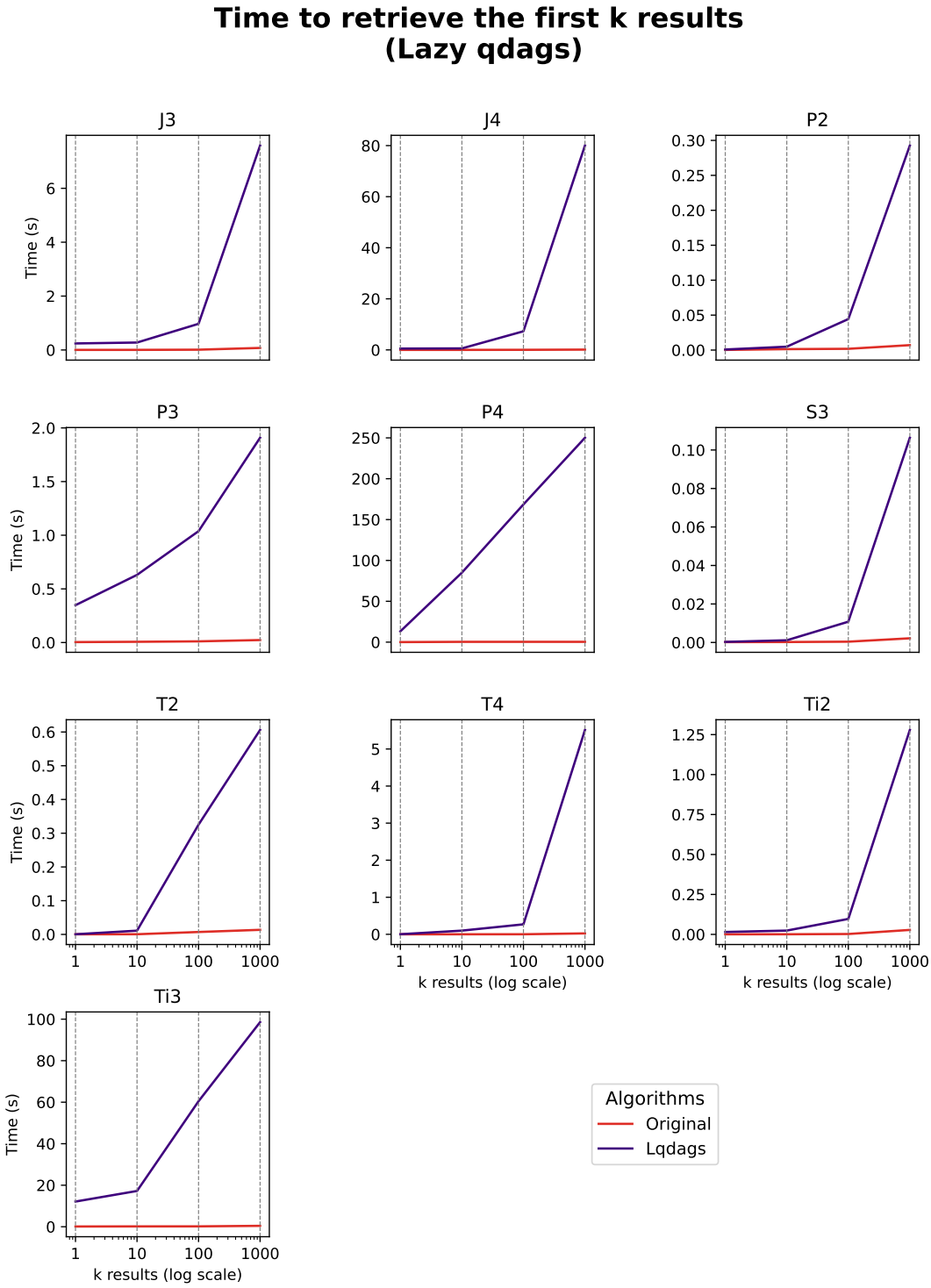


Figure 4.19: Time to retrieve the first k results using lqdags to retrieve the first k results for a join that produces many results.

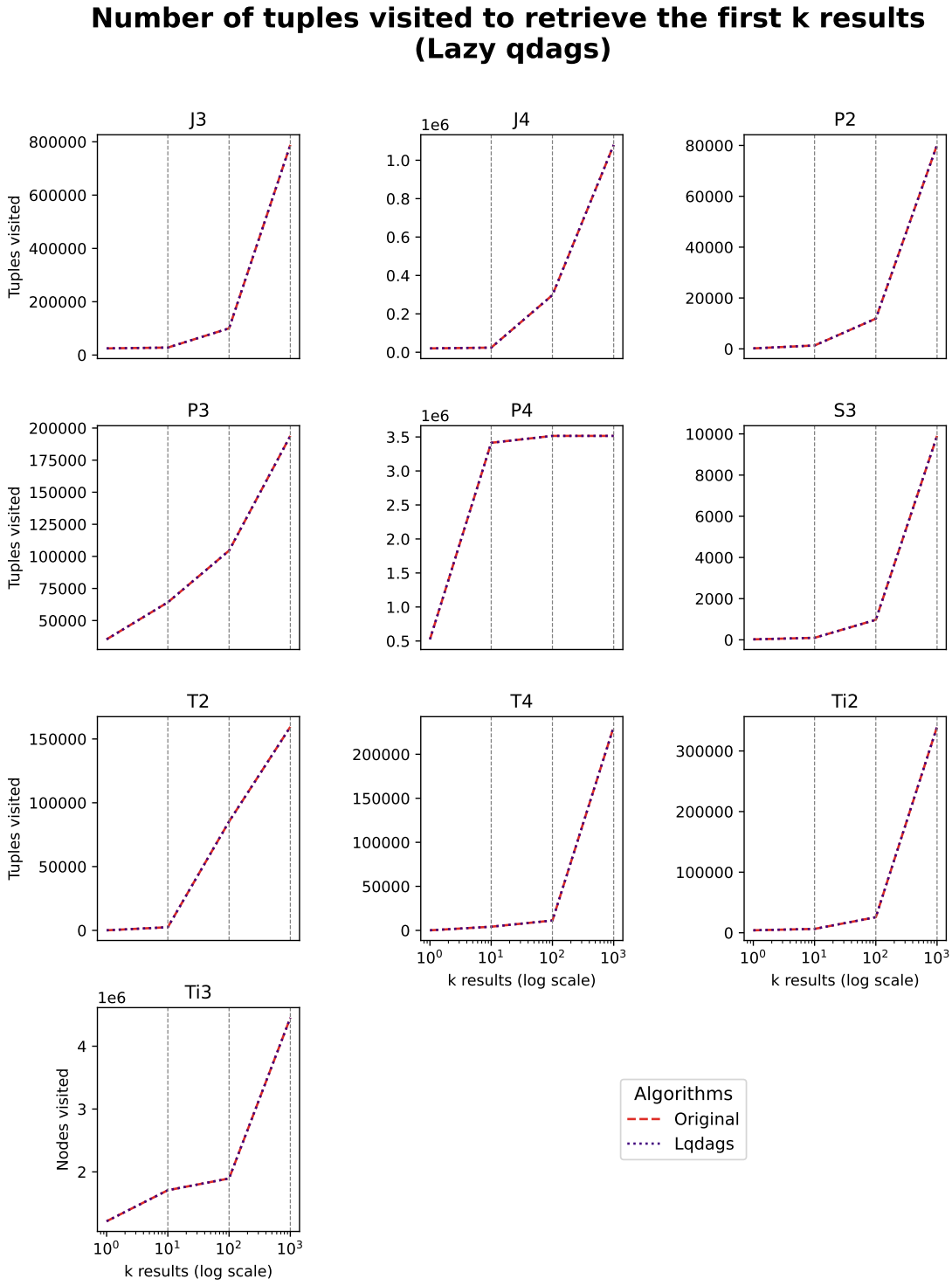


Figure 4.20: Number of tuples visited to retrieve the first k results using lqdags to retrieve the first k results for a join that produces many results.

Chapter 5

Discussion

In Sections 3.6 and 4.5, we presented the results of our experiments. We ran the same experiments for all our algorithms (see Figure 1.1) and data structures, including traditional qdags (the sequential version) and lqdags. We used the graph patterns described in Chapter 2 to evaluate the join performance. As mentioned, we did not evaluate Boolean algebra, which remains for future work. In this section, we compare the results of the different approaches and discuss the trade-offs between them.

Gradual retrieval

In both scenarios, the original algorithm outperformed the gradual retrieval algorithms regarding time complexity across all patterns for each value of k. This was unexpected, as we had anticipated that the new algorithms would be faster for the smaller values of k.

The estimators used to predict the most promising nodes to visit were not effective, as the number of tuples visited by the gradual retrieval algorithms was very similar to that of the original algorithm, and in some cases, even worse. Therefore, the original algorithm is clearly the best option for retrieving the first k results, as it is a simpler approach that does not require additional logic of computing the number of leaves.

In the profiler analysis of the original and the gradual retrieval algorithm (refer to Figures 5.1 and 5.2), we observe that in the **original algorithm**, the cost of the join operation is primarily driven by the cost of materializing a node, which accounts for 54% of the total cost. In contrast, the **gradual algorithm** shows that this cost is reduced to approximately 6%, with the cost of computing the number of leaves significantly increasing to 75%. This highlights that the overhead introduced by the logic at each step is considerable.

According to Table 3.2, we expected that algorithms utilizing **DFUDS** would outperform those using **LOUDS** in **terms of time**, as discussed in Section 3.2. However, LOUDS generally performed better than DFUDS in both scenarios, although the difference was not substantial. This may be explained by the fact that DFUDS implementation of the operations is $O(\log n)$, not $O(1)$ as in theory. The balanced parentheses implementation used in DFUDS,

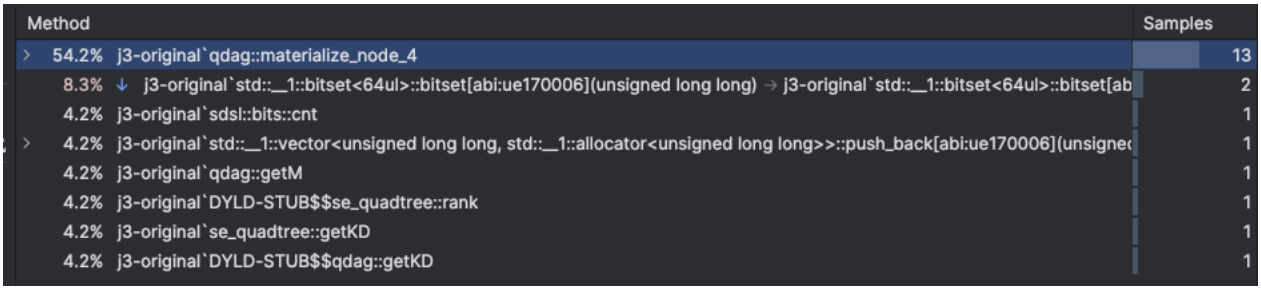


Figure 5.1: Profiler of the original algorithm to retrieve $k = 1000$ results.

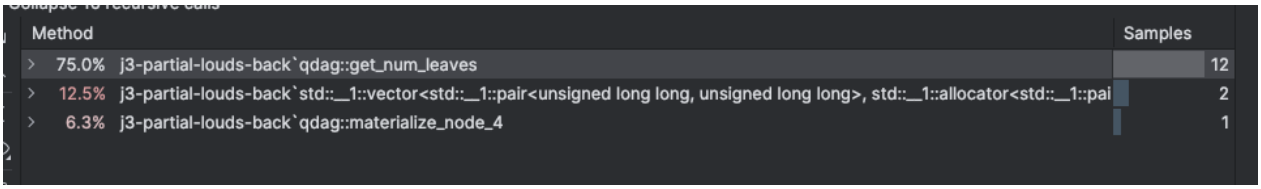


Figure 5.2: Profiler of the gradual retrieval algorithm (LOUDS-Backtracking) to retrieve $k = 1$ result.

as proposed by [47], actually results in logarithmic operations. Additionally, as discussed in Section 2, using DFUDS requires storing the tree topology, which incurs additional space usage. Therefore, in this scenario, using LOUDS is the best option in terms of time and memory complexity.

Figures 5.3 and 5.4 illustrate the profiler results for the gradual retrieval algorithm using LOUDS and DFUDS, respectively. When using LOUDS, most of the cost arises from the recursive calls to compute the number of leaves, which essentially involves a `rank` operation for each function invocation. While in the case of DFUDS, the primary cost stems from computing the excess using the parentheses structure, indicating that the cost is associated with the specifics of the DFUDS implementation.

Regarding the strategies used, we observed that the **backtracking** strategy outperformed the **optimal order** strategy in **terms of time**, and the difference in performance is quite significant in both scenarios. In the second scenario, the backtracking algorithms were nearly as efficient as the original algorithm when k was small. In fact, there was no significant performance difference for patterns such as J3, P2, S3, T2, T4, and Ti2 when $k < 100$.

There was no significant difference between the representations (LOUDS and DFUDS); however, a more notable gap existed between the different approaches (backtracking versus optimal order). Therefore, the complexity of calculating the number of leaves descending from a node was not substantial enough to favor one representation over the other. Moreover, choosing one strategy over the other yields different performance, with a more pronounced difference than the tree representations.

In terms of the **number of tuples visited**, we noticed that, in average, the **backtracking** strategy visited fewer nodes than the **optimal order** strategy. The graphs also revealed that the optimal order strategy visited fewer tuples for some patterns than the backtracking strategy. That said, we cannot draw any definitive conclusions about which approach is

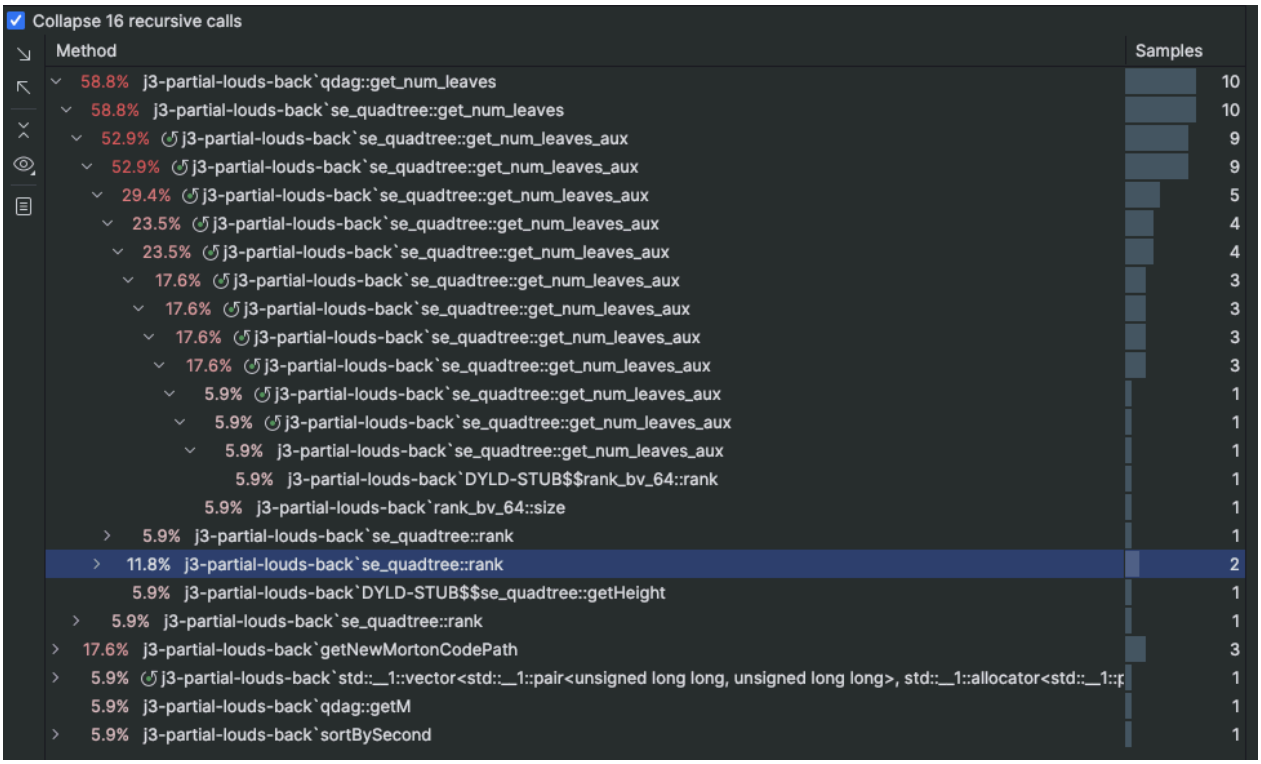


Figure 5.3: Profiler of the gradual retrieval algorithm using LOUDS-Backtracking.

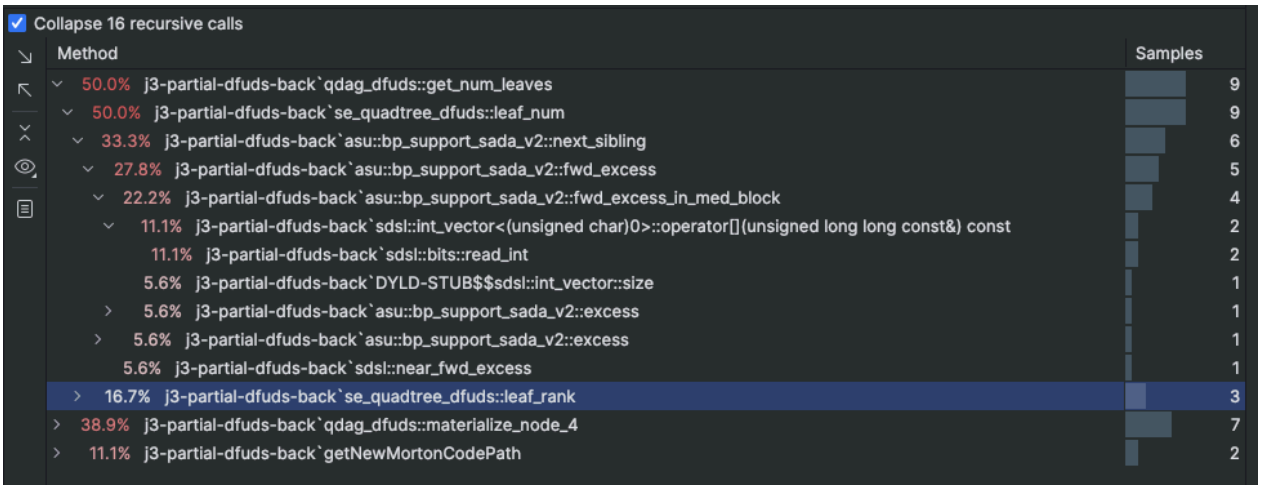


Figure 5.4: Profiler of the gradual retrieval algorithm using DFUDS-Backtracking.

superior based solely on the topology of the queries. For instance, in the case of cyclic queries like S3 and Tr1, one pattern showed that the optimal order strategy visited fewer nodes. In contrast, the backtracking strategy had fewer nodes visited for another pattern.

We observed a slight increase in the number of tuples visited as k increases, but we expected a more significant difference. We anticipated that the new algorithms would traverse fewer tuples than the original algorithm for small values of k since they are designed to focus on those nodes most likely to yield results. However, this was not the case for the backtracking strategy when $k \leq 100$. In all other cases, we successfully achieved our goal of visiting fewer nodes compared to the original algorithm.

Analyzing both metrics, we found that visiting more nodes correlates with increased time for each algorithm. However, visiting fewer nodes does not necessarily guarantee that an algorithm will be faster. For instance, in pattern J4, the original algorithm visits more nodes than any other algorithm for $k > 1$, yet it remains the fastest among them.

Furthermore, the backtracking strategy outperformed the optimal order method in terms of execution time. The slowness of the optimal order strategy can be mainly attributed to the costs associated with the size of the priority queue. Additionally, this advantage of the backtracking strategy could be attributed to its high level of optimization. For instance, instead of using a priority queue to manage the order of traversing the tuples, we utilize a vector of pairs consisting of <index, weight> for the tuples and simply sort this vector. This operation is significantly simpler than maintaining a priority queue and retrieving the maximum element multiple times.

In conclusion, the overhead introduced by the logic at each step to estimate the nodes most likely to yield results outweighed the benefit of visiting fewer nodes. This suggests that further optimization of the logic and estimators is necessary to improve the performance of the new algorithms.

Ranked enumeration

We observed in both scenarios that the ranked enumeration algorithms are slower than the original algorithm. We had anticipated that the ranked enumeration algorithms would outperform the original algorithm because they traverse the qdags using the highest-priority tuples. In contrast, the original algorithm has to maintain a heap with the top k results seen so far. Although we did not perform this last operation, the difference between the novel and original algorithms was significant, with the original algorithm remaining the fastest option even when accounting for potential overheads.

We did see a few promising results for $k = 1$ in some patterns of the "first scenario", particular for P2, S3, T2, T3, Ti2 and Ti3. In the "second scenario", we achieved better outcomes and were able to retrieve the top result faster than the original algorithm in certain patterns, including P2, T2, T3 and Ti2. However, the difference in performance was not as significant as we had hoped.

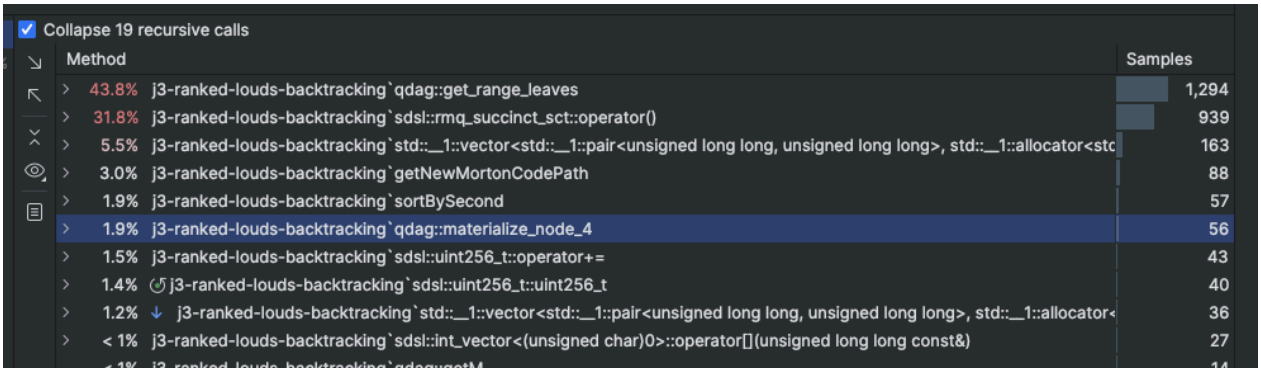


Figure 5.5: Profiler of the ranked enumeration algorithm (LOUDS-Backtracking) to retrieve $k = 1$ result.

In Figure 5.5, we observe the profiler for one of the ranked enumeration algorithms. The most costly operation is the one that calculates the priority of the tuples, referred to as `get_range_leaves`. In contrast, the most expensive operation in the original algorithm, `materialize_node` (as shown in Figure 5.1), represents only 1.9% of the total cost in the ranked enumeration algorithm. This indicates that the additional operations we introduced to compute the priorities are significantly more expensive.

Generally, **DFUDS** was faster than **LOUDS**, contrasting with the gradual retrieval algorithms. However, similarly to those algorithms, the performance difference between using LOUDS and DFUDS was not significant. Additionally, the **backtracking** approach proved to be quicker than the **optimal order** strategy.

The **number of tuples** visited indicates that our algorithms traverse significantly fewer tuples than the original algorithm, when k is small. In the "first scenario", we visit nearly 37% of the tuples that the original algorithm visits, and in the "second scenario", we visit almost 1%. This significant difference between the two scenarios suggests that our approach is effective when the join produces many results.

Despite this improvement in node traversal, the overhead associated with the new algorithms still makes them slower than the original algorithm. Nevertheless, if we decided not to store the priorities (to save space), we would need to compute them for each tuple. In this case, traversing fewer nodes could lead to better overall results.

Gradual retrieval and ranked enumeration

We observed some encouraging results in terms of time complexity for the patterns P2, T2 and Ti2 (for gradual retrieval and ranked enumeration). These three patterns share a similar morphology (see Figure 2.15); they are the only queries that require a join between two relations. However, we could not identify any particular reason for this behavior. A more exhaustive study of the patterns is needed to understand the behavior of the algorithms. Additionally, we could not identify any correlation between the number of results produced in the first scenario (see Table 3.3) and the time complexity.

We did have better results in the "second scenario" compared it with the first one, where the join operation produces numerous results. But the original algorithm continues to be the fastest generally.

We observed that our algorithms were also practical in scenarios with few results and where it was challenging to identify the intersection points between the qdags. We had not anticipated this scenario as a particularly useful case, as we designed these algorithms to quickly retrieve the first or top results when the join produces many results.

Another interesting feature of our algorithms is that they directly output the coordinates of the results. In contrast, the original algorithm produces a quadtree that must be traversed to locate the results. This can be an advantage when we are only interested in the results. However, if we want to combine the qdag output with other qdags, we need to build a quadtree based on the results.

We observed that LOUDS outperforms DFUDS in gradual retrieval algorithms, while the opposite happened for ranked enumeration algorithms. The functions `get_num_leaves` and `get_range_leaves` are very similar for both LOUDS and DFUDS. In ranked enumeration, we visit more nodes compared to gradual retrieval (see Tables 3.4 and 3.6). Consequently, we may access more tuples higher up in the tree, which could lead to increased computation time for the functions in LOUDS compared to DFUDS. However, the time difference between LOUDS and DFUDS is not significant, as previously mentioned.

Lazy qdags

Regarding the **join** in lazy qdags, it is important to note that while this approach requires us to traverse the syntax tree each time we want to retrieve a child of the formula, it performs well when only a few results are needed, as we saw in the second scenario for $k < 10$. However, retrieving the first results is not as good as the original algorithm.

When we compare execution time and the number of tuples visited, we observe that the curves for both metrics are similar—indicating that even a slight increase in the number of tuples visited results in a significant increase in execution time.

The great advantage of lazy qdags is that they allow us to build more complex relations and compute all forms of **relational algebra**, which has not been done with the original data structure (qdags). It is important to mention that when we build the output, we create a traditional quadtree, not a compact one like the original algorithm.

With lazy qdags, we can create more complex relations and perform additional operations such as selection and projection. The selection operation allows us to apply filters to the results, while projection lets us choose which attributes to display. Furthermore, we can share lazy qdags among different operations, so one evaluation of a lazy qdag can benefit other evaluations by avoiding redundant calculations.

These capabilities make lazy qdags valuable for building and evaluating complex relations. Although we could not directly compare them with the other algorithms, they remain a

strong option when dealing with more elaborate relations.

All the algorithms

Join - time complexity

When comparing all the algorithms for retrieving the first and the top results, it is still advisable to use the original algorithm and maintain a heap with the best results seen so far, in case we want to retrieve the top results.

Among the new algorithms, if our priority is to retrieve the first results as quickly as possible, we should use the gradual retrieval algorithm along with the LOUDS representation and the backtracking strategy. To obtain the top results, we should use the ranked enumeration algorithm using the DFUDS representation and the backtracking strategy.

Lazy qdags are effective for minimal result sets (e.g., when $k < 10$). However, even in these scenarios, the gradual retrieval algorithm using the backtracking strategy often performs better.

Both gradual retrieval and ranked enumeration focus on outputting only the coordinates of the tuples, avoiding the need to build the entire output set. Therefore, if we need to store the results for further processing, we should consider using either the original algorithm or lazy qdags.

Join - space complexity

When considering space complexity, it is important to account for the memory the output requires. The original algorithm utilizes a compact quadtree for output storage, while the gradual retrieval algorithms and ranked enumeration algorithms store the output as a list of coordinates. Consequently, the original algorithm is the most memory-efficient since it uses qdags and maintains the output as a k^2-tree.

Lazy qdags require the storage of the syntax tree, satellite information, and a pointer to the qdags. When retrieving a child, we construct a traditional quadtree with pointers, which does not significantly reduce memory usage. Therefore, when space complexity is a priority, lazy qdags are the least favorable option for performing join operations.

Among the novel algorithms, if we aim to optimize space complexity and achieve faster results, we should consider using the gradual retrieval algorithm, using LOUDS over DFUDS. In general, using the backtracking strategy is more memory-efficient than using the optimal order strategy, as its priority queue is smaller.

To obtain the top results, if we use the ranked enumeration algorithm, we should use the LOUDS representation and the backtracking strategy. But we should also consider the

additional overhead added due to the Range Maximum Query data structure used to compute priorities.

Relational algebra

As mentioned, only lazy qdags can implement the full relational algebra operations and construct more complex relations. Therefore, we could not compare them directly with the other algorithms. Comparing lazy qdags with traditional data systems remains an area for future work.

Chapter 6

Conclusions

Recent studies on compact data structures have advanced the understanding of join algorithms in graph databases. These structures enhance space efficiency while maintaining time performance during data operations.

We have extended a join algorithm based on recent compact data structures, specifically a compact version of the k^2-trees known as qdags, developed by Arroyuelo et al. [7]. Our work involves creating and evaluating an extension of qdags that allows traversing fewer nodes to return the first results. Additionally, the results can be prioritized based on a specified weight or priority. We also implemented a lazy version that supports Boolean queries and full relational algebra, all while ensuring worst-case optimality, on the former. Moreover, we adapted the original join algorithm to stop after computing the first k results and build the quadtree output.

Our experiments have shown that the **gradual retrieval** algorithms using the backtracking strategy can improve the number of tuples visited to obtain the first results in a join that produces many results. However, this approach increased space complexity and the time required to compute the join, while still guaranteeing worst-case optimality.

On the other hand, the **ranked enumeration** join algorithm effectively retrieved the top k results by traversing significantly fewer nodes than the baseline method, which obtains all results before selecting the best ones for small k. This method also maintained worst-case optimality. Nevertheless, it had disadvantages, including longer overall computation times and higher memory usage due to the Range Maximum Query data structure.

We also evaluated two methods for computing the number of leaves and a range of leaves of a node: **LOUDS**, which achieves a logarithmic time operation, and **DFUDS**, which is theoretically constant time. Our experiments demonstrated that LOUDS outperformed DFUDS in practice during gradual retrieval algorithms. However, in the case of ranked enumeration, **DFUDS** had a better performance than **LOUDS**. The slower performance of DFUDS in gradual retrieval is mainly due to suboptimal implementations of the data structure that included some operations with logarithmic time complexity. In future work, we could explore a faster implementation of DFUDS, which could be more efficient than LOUDS.

These operations introduced some overhead to our gradual retrieval and ranked enumeration algorithms. Also, when using the optimal order approach, we need to operate over a priority queue. All these factors contributed to the overall overhead of the new algorithms.

The logic added at each step to compute gradual retrieval and ranked enumeration results in a non-negligible overhead, leading to longer processing times. However, this comes with the benefit of traversing fewer nodes. Therefore, to improve our results, we should optimize the additional logic incorporated in the new algorithms and improve the estimators for the gradual retrieval algorithms to improve the number of tuples visited. For example, we could explore a balance between using faster estimators that may be less accurate.

When considering which technique for obtaining partial or ranked results is more effective, we must weigh up the trade-off between time and memory to determine the best algorithm to optimize. If conserving space is our primary objective, we should choose the LOUDS representation for trees over DFUDS and use backtracking with a priority queue of fixed size k. This approach is more efficient than an optimal order technique, which employs a non-fixed size priority queue.

On the other hand, if our goal is to optimize time complexity, we should select DFUDS in conjunction with backtracking. A practical recommendation is to use LOUDS and the backtracking strategy for gradual retrieval and ranked enumeration, as it offers superior performance without significantly increasing memory usage. However, if we can develop faster estimators, the optimal order strategy may yield better overall results.

We implemented **lazy qdags** and extended their functionality to support all relational algebra operations. Additionally, we developed an API for operations such as join, difference, complement, and other queries. While lazy qdags may increase the time required to compute a complete query, they still perform well for retrieving initial results and enabling the implementation of full relational algebra.

Although some queries in lazy qdags are not worst-case optimal, studying this data structure is still worthwhile. Using a syntax tree facilitates the construction of more complex relations and allows for efficient computation by avoiding unnecessary evaluations, especially when a quadrant of the formula is already empty. However, if our only concern is to compute the join, traditional qdags are the best choice regarding time and memory efficiency.

We compared different approaches for computing the join, including traditional joins using qdags, gradual retrieval, ranked enumeration, and lazy qdags. Our findings indicate that the original representation is the most efficient in time and space. Nevertheless, ranked enumeration is still noteworthy for retrieving the top results due to the reduced number of tuples traversed. Furthermore, lazy qdags are the preferred choice for computing queries beyond joins. We can also speculate that the performance of these new implementations in disk usage would be better than the original approach.

In this thesis, we explored various representations of the quadtree, going from the compact form used in the original algorithm to gradual retrieval and ranked enumeration, as well as a non-compact data structure representation employed in lazy qdags.

A notable disadvantage of these algorithms and of using qdags, in general, is that they

require data to be stored in a specific format. This complicates their implementation in current systems.

In conclusion, our main contributions include five different algorithms for efficiently computing partial results and four algorithms for obtaining ranked results. We also implemented two methods for calculating the number of leaves using LOUDS and DFUDS, including a new, more compact version of DFUDS. Additionally, we implemented and improved the design of lazy qdags and developed an API for essential queries such as join, difference, and complement.

Future work

We did not evaluate relational algebra using lazy qdags or compare it to other database systems like Postgres. This analysis should be conducted in the future to assess the performance of lazy qdags when evaluating selection, projection, and other relational algebra formulas.

In this thesis, we did not explore the delay in results, which could be an interesting topic for future research. We could also compare the results' distribution, to better understand the algorithms' behavior.

Furthermore, we have not designed or implemented an algorithm for constructing qdags from coordinates, which could be advantageous for gradual retrieval and ranked enumeration algorithms.

We demonstrated that using qdags allows us to estimate the size of the subgrid or its priority, which is not as straightforward with other data structures. This approach could also be extended to additional operations.

Another promising direction is to explore alternative data structures for priority queues, such as Fibonacci heaps or binomial heaps, to improve the time or space efficiency of our algorithms. Finally, integrating techniques from lazy qdags into the original algorithm could be beneficial. For example, introducing the concept of "full leaves" to improve time complexity.

There are several important topics worth revisiting and studying further. One area of interest is parallelization. Although the original algorithm addressed this aspect, we did not test it, and we have yet to incorporate parallelization into our work. Investigating the potential for parallelizing our algorithms could provide valuable insights. It would be beneficial to determine whether parallelization is feasible and, if so, how it could enhance the performance of our algorithms. For example, we could parallelize the AND function, although this might compromise the advantage of visiting fewer nodes.

Another interesting subject is the analysis of clustered datasets and their entropy to see if we can achieve more compact data representations. Additionally, exploring using very sparse bitvectors on sparse grids could improve time and space efficiency.

More broadly, further exploration of graph theory and isomorphism could help simplify complex queries, reduce time complexity, and enable approximate query matches with speci-

fied distances or error tolerances. It could also be essential to investigate the characteristics of the patterns we studied and understand why some patterns exhibit greater complexity than others.

Bibliography

[1] Renzo Angles. The property graph database model. In *AMW*, 2018.

[2] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. Foundations of modern query languages for graph databases. *ACM Comput. Surv.*, 50(5), sep 2017.

[3] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1), feb 2008.

[4] 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 '21, page 102–114, New York, NY, USA, 2021. Association for Computing Machinery.

[5] Diego Arroyuelo, Gonzalo Navarro, Juan L. Reutter, and Javiel Rojas-Ledesma. Optimal joins using compressed quadtrees. *ACM Trans. Database Syst.*, 37(4), aug 2018.

[6] Diego Arroyuelo, Gonzalo Navarro, Juan L. Reutter, and Javiel Rojas-Ledesma. Optimal joins using compressed quadtrees. *ACM Trans. Database Syst.*, 47(2), may 2022.

[7] Diego Arroyuelo, Gonzalo Navarro, Juan L. Reutter, and Javiel Rojas-Ledesma. Optimal joins using compressed quadtrees. *ACM Trans. Database Syst.*, 47(2), May 2022.

[8] Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. *SIAM J. Comput.*, 42(4):1737–1767, jan 2013.

[9] Pablo Barceló, Leonid Libkin, and Juan L. Reutter. Querying graph patterns. In *Proceedings of the Thirtieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '11, page 199–210, New York, NY, USA, 2011. Association for Computing Machinery.

[10] Pablo Barceló Baeza. Querying graph databases. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '13, page 175–188, New York, NY, USA, 2013. Association for Computing Machinery.

[11] Holger Bast, Debapriyo Majumdar, Ralf Schenkel, Christian Theobalt, and Gerhard Weikum. Io-top-k: Index-access optimized top-k query processing. 2006.

[12] David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, dec 2005.

[13] Jon Louis Bentley. K-d trees for semidynamic point sets. In *Proceedings of the Sixth Annual Symposium on Computational Geometry*, SCG '90, page 187–197, New York, NY, USA, 1990. Association for Computing Machinery.

[14] Alain Bretto. Hypergraph theory. *An introduction. Mathematical Engineering. Cham: Springer*, 1, 2013.

[15] Nieves R. Brisaboa, Susana Ladra, and Gonzalo Navarro. Compact representation of web graphs with extended functionality. *Inf. Syst.*, 39:152–174, jan 2014.

[16] D. R. Clark. *Compact PAT Trees*. PhD thesis, University of Waterloo, Canada, 1996.

[17] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, 1967.

[18] Guillermo de Bernardo, Travis Gagie, Susana Ladra, Gonzalo Navarro, and Diego Seco. Faster compressed quadtrees. *J. Comput. Syst. Sci.*, 131(C):86–104, feb 2023.

[19] Shaleen Deep, Xiao Hu, and Paraschos Koutris. Ranked enumeration of join queries with projections. *Proc. VLDB Endow.*, 15(5):1024–1037, jan 2022.

[20] Shaleen Deep and Paraschos Koutris. Ranked enumeration of conjunctive query results. *arXiv preprint arXiv:1902.02698*, 2019.

[21] Héctor Ferrada and Gonzalo Navarro. Improved range minimum queries. *Journal of Discrete Algorithms*, 43:72–80, 2017.

[22] Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011.

[23] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014.

[24] Georg Gottlob, Nicola Leone, and Francesco Scarcello. The complexity of acyclic conjunctive queries. *Journal of the ACM (JACM)*, 48(3):431–498, 2001.

[25] A. Gupta, W.-K. Hon, R. Shah, and J. S. Vitter. Compressed data structures: Dictionaries and data-aware measures. *Theoretical Computer Science*, 387(3):313–331, 2007.

[26] Olaf Hartig and Jorge Pérez. Semantics and complexity of graphql. In *Proceedings of the 2018 World Wide Web Conference*, WWW '18, page 1155–1164, Republic and Canton of Geneva, CHE, 2018. International World Wide Web Conferences Steering Committee.

[27] Meng He, J. Ian Munro, and Patrick K. Nicholson. Dynamic range selection in linear space. In *Proceedings of the 22nd International Conference on Algorithms and Computation*, ISAAC'11, page 160–169, Berlin, Heidelberg, 2011. Springer-Verlag.

[28] Gísli R. Hjaltason and Hanan Samet. Incremental distance join algorithms for spatial databases. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, page 237–248, New York, NY, USA, 1998. Association for Computing Machinery.

[29] Gísli R. Hjaltason and Hanan Samet. Incremental distance join algorithms for spatial databases. In Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 237–248. ACM Press, 1998.

[30] Gísli R. Hjaltason and Hanan Samet. Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24(2):265–318, 1999.

[31] Gísli R. Hjaltason and Hanan Samet. Index-driven similarity search in metric spaces. *ACM Trans. Database Syst.*, 28(4):517–580, 2003.

[32] Aidan Hogan, Cristian Riveros, Carlos Rojas, and Adrián Soto. A worst-case optimal join algorithm for sparql. In *The Semantic Web – ISWC 2019: 18th International Semantic Web Conference, Auckland, New Zealand, October 26–30, 2019, Proceedings, Part I*, page 258–275, Berlin, Heidelberg, 2019. Springer-Verlag.

[33] Xiao Hu and Ke Yi. Instance and output optimal parallel algorithms for acyclic joins. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '19, page 450–463, New York, NY, USA, 2019. Association for Computing Machinery.

[34] G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.

[35] Nikolaos Karalis, Alexander Bigerl, and Axel-Cyrille Ngonga Ngomo. Native execution of graphql queries over rdf graphs using multi-way joins. In *Knowledge Graphs: Semantics, Machine Learning, and Languages*, pages 77–93. IOS Press, 2023.

[36] Mahmoud Abo Khamis, Hung Q. Ngo, Christopher Ré, and Atri Rudra. Joins via geometric resolutions: Worst case and beyond. *ACM Trans. Database Syst.*, 41(4), nov 2016.

[37] A. Marian, S. Amer-Yahia, N. Koudas, and Divesh Srivastava. Adaptive processing of top-k queries in xml. In *21st International Conference on Data Engineering (ICDE'05)*, pages 162–173, 2005.

[38] G.M. Morton. *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing*. International Business Machines Company, 1966.

[39] J. I. Munro. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, LNCS 1180, pages 37–42, 1996.

[40] J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

[41] Gonzalo Navarro. *Compact Data Structures: A Practical Approach*. Cambridge University Press, USA, 1st edition, 2016.

[42] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. *J. ACM*, 65(3), mar 2018.

[43] 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, feb 2014.

[44] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 60–70, 2007.

[45] Giulio Ermanno Pibiri and Rossano Venturini. Dynamic elias-fano representation. In *28th Annual symposium on combinatorial pattern matching (CPM 2017)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2017.

[46] Alain Pirotte. A precise definition of basic relational notions and of the relational algebra. *SIGMOD Rec.*, 13(1):30–45, sep 1982.

[47] Kunihiko Sadakane and Gonzalo Navarro. Fully-functional succinct trees. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, page 134–149, USA, 2010. Society for Industrial and Applied Mathematics.

[48] Nikolaos Tziavelis, Deepak Ajwani, Wolfgang Gatterbauer, Mirek Riedewald, and Xiaofeng Yang. Optimal algorithms for ranked enumeration of answers to full conjunctive queries. In *Proceedings of the VLDB Endowment. International Conference on Very Large Data Bases*, volume 13, page 1582. NIH Public Access, 2020.

[49] Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. Optimal join algorithms meet top-k. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2659–2665, New York, NY, USA, 2020. Association for Computing Machinery.

[50] Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. Any-k algorithms for enumerating ranked answers to conjunctive queries. *arXiv preprint arXiv:2205.05649*, 2022.

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

[52] W3C. Sparql query language for rdf. `https://www.w3.org/2001/sw/DataAccess/rq23/#BasicGraphPatternMatching`, 2024. Accessed: 2024-06-18.

[53] Qichen Wang, Qiyao Luo, and Yilei Wang. Relational algorithms for top-k query evaluation. *Proc. ACM Manag. Data*, 2(3), may 2024.

[54] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*, VLDB '81, page 82–94. VLDB Endowment, 1981.

Annex A

Comparison of both approaches of the estimators for gradual retrieval

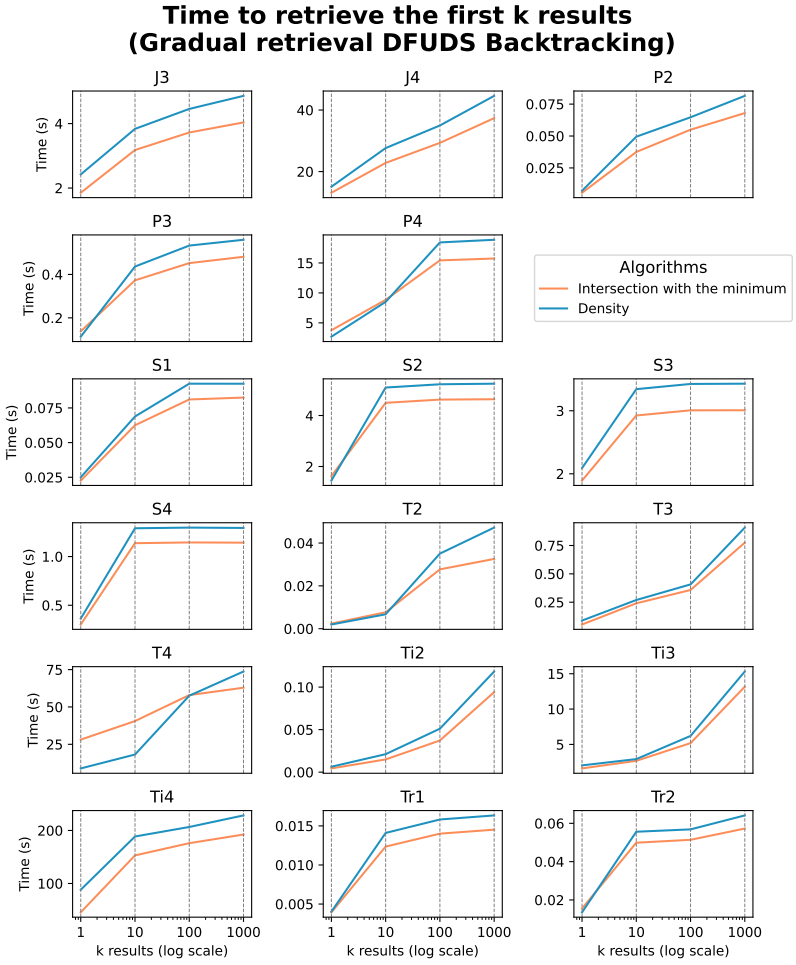


Figure A.1: Query times (in seconds) for the different approaches of gradual retrieval algorithms to retrieve the first k results.

Annex B

Comparison of both approaches of the estimators for ranked enumeration retrieval

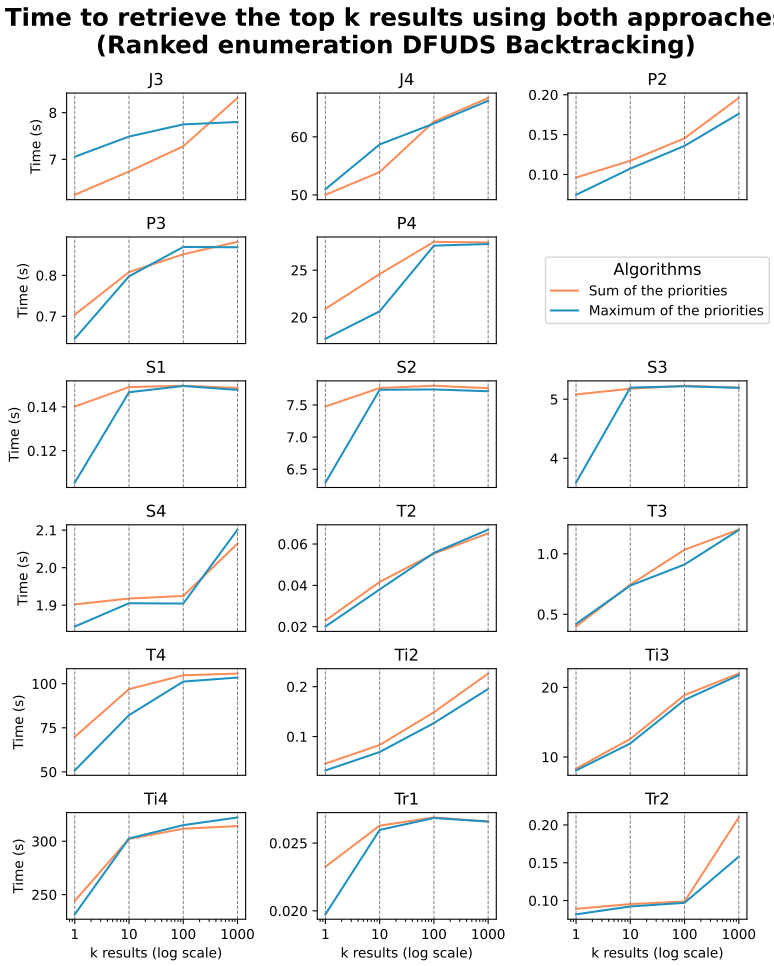


Figure B.1: Query times (in seconds) for the different approaches of ranked enumeration algorithms to retrieve the top k results.