# Ranked enumeration and lazy evaluation of graph database joins

Department of Computer Science
University of Chile
Santiago, Chile
August 2023

## 1 Introduction

Over the years, the need to store more information has dramatically increased, but external memory has became slower regarding the CPU. 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. In particular, we are interested in representing large databases in a compact form.

Compact data structures [16] significantly reduce the space and support fast operations. The main objective in compact data structure is to use space close to the data entropy while retaining, if possible, the classic time complexity in the operations.

The $K^2$-tree is a data structure commonly used to represent grids, web graphs, or to store geographic information, among others. While the classic version needs $O(1)$ pointers per node, the compact representation [16] use only $O(k^2)$ bits per internal node, and support the basic navigation operations in constant time.

This data structure also proved helpful to represent databases. With a $K^2$-tree, we can represent a relation $R$ of $d$ attributes as a $d$-dimensional grid, where each tuple is represented as a point. Relational and graph databases need to support join operations which is usually the most expensive operation. Graph databases tend to feature multijoins, among many relations [1]. Using this structure, we can perform join and increase efficiency.

Recent works has shown that solving multi-joins via pair wise joins, which has been the classic approach, is suboptimal [18]. New multi-join algorithms that are worst-case optimal (wco), like LeapFrog Trie Join (LTJ) [20] or Tetris [13], need a heavy index data structure that has to be stored on disk [2]. Arroyuelo et al. (2018) proposed this first worst-case optimal multi-join algorithm focused **on both time and space**. It uses a version of compressed quadtrees, called qdags, based on $k^2$-trees, that supports join, union, intersection and negation in wco time. Qdags could also be extended to all relational algebra operations, yet not guaranteeing wco time.

Most research in this areas focuses on maximizing the throughput, or equivalently, minimizing the query completion time. In many cases, especially in human interfaces, one is more concerned about the time it takes to show the first $k$ results. This problem is close to that of ranked retrieval [19], where more relevant results must be obtained first. **The first part of this work** aims to work on this area, using

qdags to compute (i) partial results in less time and (ii) computing the $k$ top results given a priority. We will resort to prioritized retrieval techniques known in similarity search [8] [9].

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). In these cases, we will be interested in studying a lazy version of the qdags, also called *lqags* [2], which compute only the needed nodes. *Lqdags* also permit extending qdags to the relational algebra. **The second part of this work** will implement and evaluate the *lazy qdags*. This could worsen throughput, TODO: porque empeoraría? qdags paper but improve the time to obtain the first results, and extend qdags to the more general relational algebra.

## 2 Related works and concepts

In this section, we introduce some key concepts and related work, such as some compact data structures, that are useful for our work, and the state-of-the-art on worst-case optimal joins.

### 2.1 $K^2$-tree

**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 ocurrences of $v$ in $B[1,i]$) and SELECT$_v$(B,$i$) (the position $j$ of the i-th $v$ in $B$). This data structure is the core of many other compact data structures, such as the $k^2$-tree.

A compressed representation encodes $B$ in $n+o(n)$ bits, while supporting the operations in constant time [6] [11] [14] [16]. We can achieve lower space for very sparse bitvectors [16] 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 RANK in $O(\min(\log m, \log\frac{n}{m}))$. Therefore, the time and space are proportional to $m$ and not the bitvector size.

**$K^2$-trees** [5] are a compact quadtree representation (in the case $k=2$). It is used to save the data of a matrix $M_{l\,x\,l}$ by dividing it recursively into $k^2$ quadrants of the same area. If the matrix side is not a multiple of $k$, the matrix is completed with 0s. The height of a $k^2$-tree is $h=\lceil \log_k l \rceil$. The nonempty quadrants are recursively subdivided.

The $k^2$-tree is represented as a compact cardinal tree [5]. This data structure called LOUDS for "Level-Order Unary Degree Sequence", represents each level of the cardinal tree using only a bitvector: each node is represented by its children using $k^2$ bits. For example, in a quadtree ($k=2$), a leaf will be represented with four zeros 0000, a node with 4 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 on it. We can traverse the tree using the basic operations in bitvectors: ACCESS, RANK and SELECT. We can see an example of a matrix displayed in Figure 1, represented by the quadtree in Figure 2. Also, a LOUDS example is shown in Figure 3.

A LOUDS representation for cardinal trees with $n$ internal nodes will use $k^2n+o(n)$ bits and support RANK and SELECT in constant time [16]. With these two essential operations, this representation supports in constant time: CHILDREN($v$), FIRSTCHILD($v$), LASTCHILD($v$), TCHILD($v,t$), NEXTSIBLING($v$),


Figure 1: Example of a 16 x 16 matrix, with 14 points.

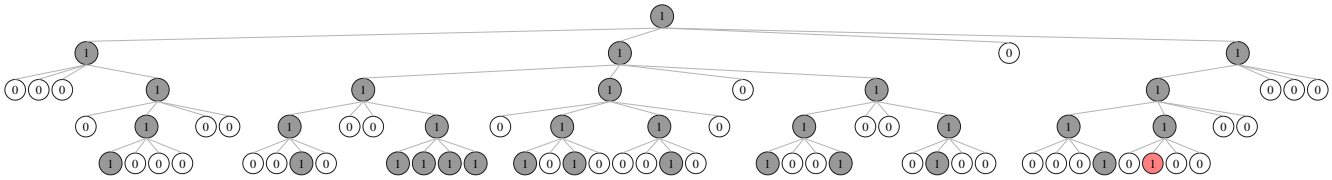

Figure 2: Quadtree of the matrix of Figure 1.

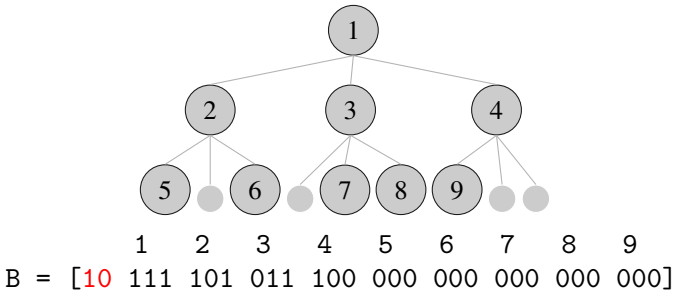


Figure 3: LOUDS representation of a cardinal tree ($k$=3).

PREVIOUSSIBLING($v$) and PARENT($v$). In Table 1, we can see a summary of the LOUDS operations. A particularity of the $k^2$-tree representation is that we do not need to store the leaves at the last level (only 0s) because we already know there are leaves at this depth.

Using the compressed representation of sparse bitvectors mentioned before, it is possible to reach the worst-case entropy of $k^2$-trees [16]. The entropy is a lower bound of the number of bits needed to repre-

| 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 $v$ |

Table 1: Summary of some of the operations in LOUDS.

sent 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 $U$.

Analyzing the worst-case entropy $H_{wc}$, the number of cardinal trees of arity $k^2$ with $n$ nodes is [16]

$$|T_n^{k^2}| = \binom{k^2 n+1}{n} / (k^2 n+1)$$

And so, the worst-case entropy is:

$$H_{wc}(T_n^{k^2}) = \log \left( \frac{\binom{k^2 n+1}{n}}{k^2 n+1} \right) = \left( k^2 \log \frac{k^2}{k^2-1} + \log \left(k^2-1\right) \right) n - O(\log n)$$

With the compressed representation of a sparse bitvector $B[1,k^2 n]$ with $n-1$ `1`s (one per node, except the root), it is possible to achieve $n \log k^2 + O(n)$ bits, which is asymptotically optimal.

## 2.2 Parentheses

A balanced sequence of parentheses [15] is represented as a bitvector $B[1,n]$ where `1` represents '(' and `0` ')'. There are $n/2$ matching pairs of parentheses, so each '(' can be associated with another ')' and viceversa. This structure supports operations: 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; and ENCLOSE($B,i$), that "returns the rightmost position $k < i$ such that $[k, \text{close}(B,k)]$ contains $i$" [16].

**DFUDS** [4] is another way to represent a cardinal tree using only $k^2 + o(1)$ bits per node. The structure is based on parentheses (so it supports its operations) and consists in 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 the description in $B$ 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 that gives the same node in formation of LOUDS [16]. In Figure 4, we can see these two bitvectors on an example.

We are interested in this data structure because the classic operation CLOSE is useful to count the leaves of a node.

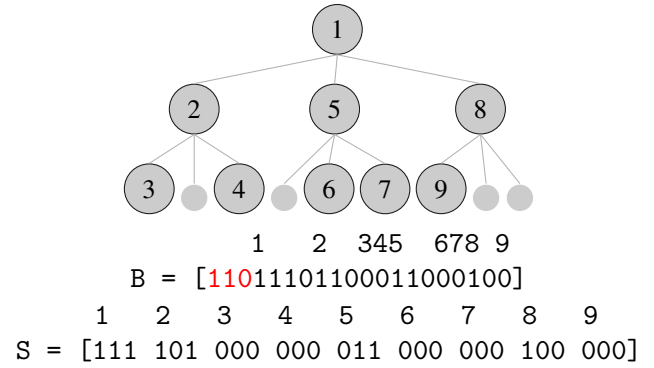


Figure 4: DFUDS representation of a cardinal tree ($k$=3).

## 2.3 Worst Case Optimal Joins

**Join operation** is the basis of many queries, especially on graph data bases, because graph queries tend to feature many joins [1]. Joins form the core of basic graph patterns, which are the basic building block of standard graph query languages like SPARQL. Thus, improving time and space in join operations directly impacts on a large set of database queries.

Traditional database engines perform the join using a **pair-wise join** algorithm [18], whose cost will depend on the plan to compute which pairs are joined first. For example, a join between the relations $R(A,B)$, $S(B,C)$, $T(C,A)$ can be done as $(R \bowtie S) \bowtie T$, or $R \bowtie (S \bowtie T)$, or also $(R \bowtie T) \bowtie S$, because the join is commutative. It has been demonstrated that joining pair-wise is *suboptimal* [18]. In fact, if the size of $S$, $R$, $T$ is $N$, our query abone done pair-wise could take $\Omega(N^2)$ time, while its maximum output possible is $O(N^{3/2})$. Still, many engines use these sub-optimal algorithms to perform joins.

A recent approach to perform a join is a **multi-way join** algorithm, meaning to join several tables all at once. This could reduce intermediate results and, therefore, time and space. Worst-case optimal multi-way join algorithms have demonstrated to have a high performance on complex queries [12]. In particular, they can solve the given join $(R \bowtie S \bowtie T)$ in the $O(N^{3/2})$ optimal time.

We are interested on this type of algorithm due to its performance, especially on cyclics queries. If the query es acyclic, that is, it has a join tree, using Yanakakis' algorithm produces the output in time linear in the size of the input and the output [21]. When we are dealing with a cyclic query, we need other algorithms to achieve worst-case optimality, such as Leapfrog Trie Join [20] or Tetris [13].

A **worst-case optimal algorithm** (wco) means that the algorithm is optimal in the worst case: there is a result whose size is proportional to the time the algorithm takes. In other words, "(...) the runtime of the algorithm is bounded by the worst-case cardinality of the query result." [10]. We can prove an algorithm is wco if it satisfies the **AGM bound**, which takes into account the structure (graph) and the size of tables, that is, it considers both types of information [18], and defines a bound according to these variables.

## 2.4 Compressed quadtrees

Arroyuelo et al. [2] proposed the first wco multi-way join algorithm using a compact data structure without any need of a heavy index data structure, unlike the other algorithms of the time. It is the first algorithm of this kind that focuses on time and space at the same time.

A tuple of the relation $R(A)$ 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 to represent this grid of $l^d$ cells, where each child represents a subgrid of size $(l/2)^d$. That is, we use $k = 2$, so each node will be described with $2^d$ bits.

The algorithm is based on a version of a quadtree ($k^2$-tree), called *qdag*, which represents lifting the dimension of some quadtree. A $d$-dimensional qdag $Q$ is a pair ($Q'$, $M$), where $Q'$ is a $d'$-dimensional quadtree (with $d' \leq d$) and $M$ is a mapping function $M : [0, 2^{d-1}] \rightarrow [0, 2^{d'-1}]$. The $i$-th child of a node in $Q$ corresponds to the $M[i]$-th of the corresponding $Q$' node. Therefore, the qdag has a higher dimension than the quadtree, and many nodes share subtrees. We only need a quadtree of a lower dimension and a mapping function to virtual traverse the quadtree of a higher dimension.

To perform a join between multiple relations, we must do two operations: EXTEND, that "(...) lifts the quadtree representation of a grid to a higher-dimensional grid" [2], so all the relations have the same attributes (here we build the qdag for each relation); and the AND operation, that makes the intersection of the qdags. To compute the AND of all qdags, we traverse all qdags recursively till we find an empty quadrant or a leaf in one of them. The result is another compressed quadtree and the points within the structure is the join result [2]. The idea is to represent each relation using a $k^2$-tree, then, upon a multi-join, build the qdag that extends the relation to those attributes that it does not have, and finally, intersect all the qdags.

*We can see in Figure 5 the grid of a relation $R(A,B)$ and a 3-dimensional grid, which is the result of extending the relation $R(A,B)$ to the attribute C. These grids are represented by a $k^2$-tree as shown in Figure 6. We can also see many nodes sharing subtrees on the second tree: that is why we only need to store the first quadtree and a mapping function to simulate the second $k^2$-tree.*

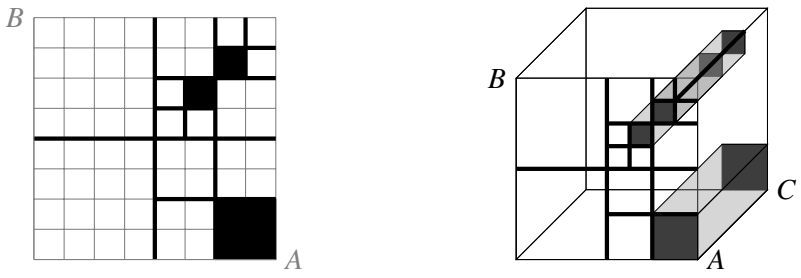


Figure 5: Extend operation of $R(A,B)$

*In Figure 7, we can see the extension of the relations $S(B,C)$ and $T(A,C)$. In Figure 8, we show the intersection of the three relations mentioned before, and in Figure 9 we show the final output of $R(A,B) \bowtie S(B,C) \bowtie T(A,C)$, that is, the remaining points at the intersection.*

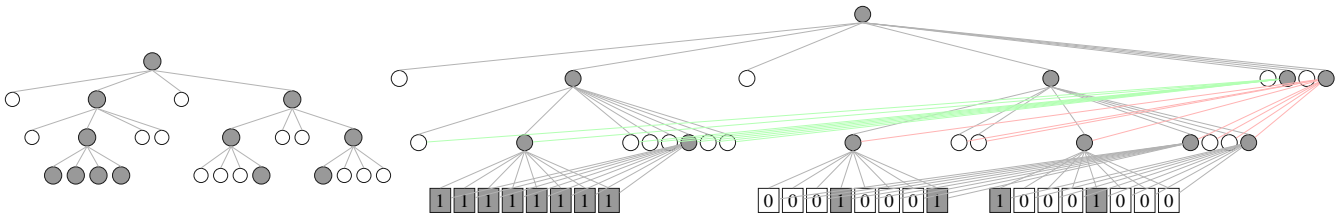


Figure 6: Qdag of $R(A,B,C)$

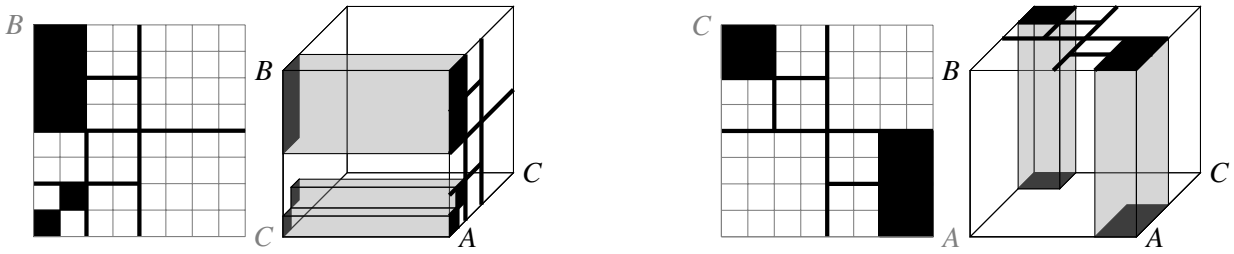


Figure 7: Extending relations $S(B,C)$ and $T(A,C)$.

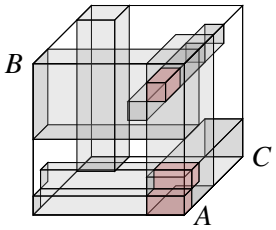


Figure 8: Intersection of relations $R'(A,B,C)$, $S'(B,C,A)$ and $T'(A,C,B)$

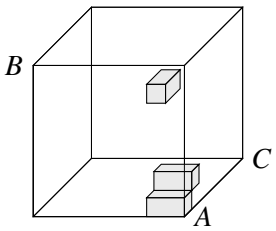


Figure 9: $R(A,B) \bowtie S(B,C) \bowtie T(A,C)$

In addition, it is possible to extend qdags to process not only joins and intersections, but also unions and negations. We could perform all the relational algebra queries, but we can no longer guarantee wco times. This is done with a lazy version of the qdags, which generalizes the idea of processing the arguments only as much as possible to build the output.

# 3 Problem Statement

We now present the problems we are willing to address in this thesis.

## 3.1 Partial and ranked results

Previous work on qdags has focused on obtaining all the query results. In many cases it is more important to find some results fast (e.g., user interfaces), or in finding more relevant tuples [19]. In this part we are interested in how to obtain some results as fast as possible (partial results), and how to output the top $k$ most prioritized results (ranked results), using qdags.

### 3.1.1 Partial results

For **partial results**, we want to perform the join so as to output some results first. We will use an upper-bound estimator and go down first by the children that we estimate produces more results. For this, we have 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.
   *For example, in Figure 10, the first child of the qdag of the relation R has 6 leaves (points), and the first child of the qdag of S has 12 leaves, thus the minimum of the first child is* $min(6,12) = 6$. *The child that has the maximum of the minima is the second child, that is* $min(16,13) = 13$. *This means that the join of this quadrant will produce at most 13 results. The other quadrants are expected to produce fewer results, so the join will start by the second children.*
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.
   *In the example of Figure 11, we are using the number of leaves of the last example (Figure 10) and we suppose the number of cells is 100. We can see in red the order used to compute the join: the second child is also the node most likely to output more results.*

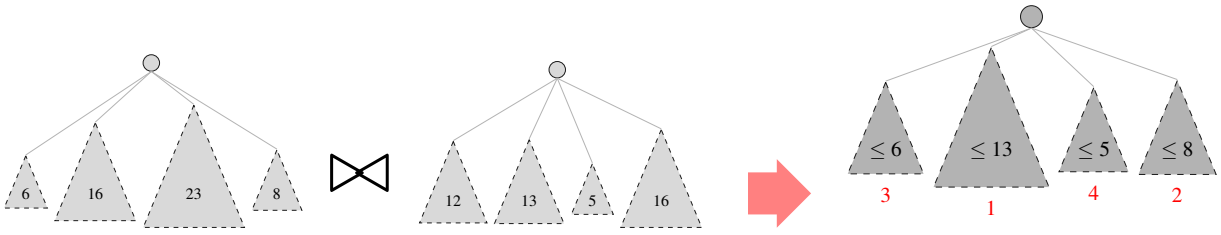


Figure 10: Order in partial results (number of leaves). Join between $R$ and $S$.

In both cases, we need to compute the number of leaves of a node. Qdags originally use a LOUDS-like representation, but there is another compact data structure that is efficient in theory (in practice, their orders are the same but still the second one should be faster). Thus, we have two possible solutions:

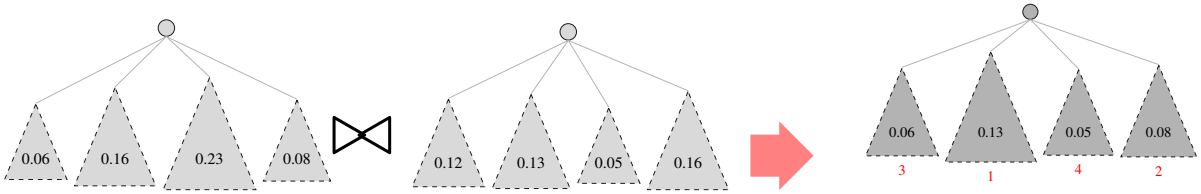


Figure 11: Order in partial results (density). Join between $R$ and $S$.

1. Use the LOUDS representation to compute the number of leaves in $O(\log l)$ time: we descend recursively by the first node that has a child until we find a leaf, and we do the same for the last node. Then, we count the 1s within this range and get the number of leaves of the subtree.
2. Using DFUDS [16], we can compute the leaves number of a node in $O(1)$ time, using the basic operation CLOSE in parentheses and $\text{RANK}_{00}$. Yet, we need more space to save this data structure, as we have seen.

*In Figure 3 we can see an example of a LOUDS representation of a cardinal tree of three children. In Figure 4 we can see a DFUDS representation of the same tree. The bits in red are necessary for some border cases.*

### 3.1.2 Ranked results

For **ranked results**, each quadtree stores an array $P[1,p]$, where $p$ is the number of points, with the weight or priority of each leaf. The main objective is to output the top $k$ most important results, for $k$ given at query time. We precompute a range maximum query (rMq), data structure on $P$, which returns the position of highest value in any given range of $P$. Given a node, we can find the range in $P$ of its descendant leaves and the rMq yields the most important descendant leaf.

Once we have calculated the maximum weight descendants from each child, we evaluate the child across all qdags by using (i) the maximum weight, (ii) the sum of the weights or (iii) by another monotone function of the weights.

*For example, in Figure 12, 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 method, we will first access to the second child because the first tree has the highest priority leaf (30) on the second child. If we proceed by addition, we will first go down by the third child, because the addition of the third nodes is 20 + 25, that is 45, which is bigger than the addition of the priorities of the second nodes (35).*

### 3.1.3 Computing the join

Either if we are interested in partial or ranked results, we need a strategy to compute the join in a certain order. We follow existing work on proximity search [8] [9].

- Prioritized backtracking: we still backtrack to generate the output tree, but the order in which the children of the nodes are visited is given by their predictors.

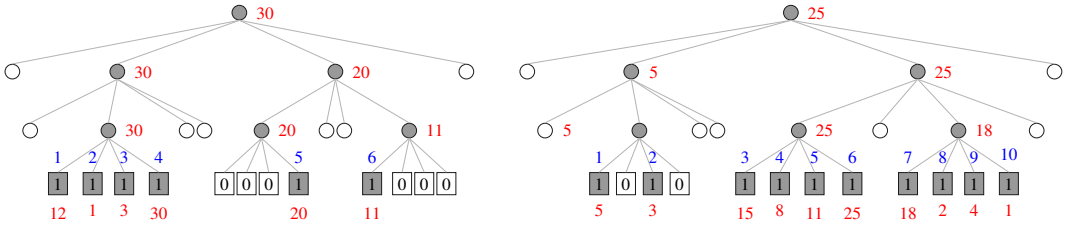


Figure 12: Order in ranked results (priority).

- Optimal order: we maintain a priority queue with all the 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.

The first strategy needs a small priority queue in the case of top-$k$ ranked results, where it stores the best results seen so far. If a node's upper bound in the backtracking is no longer than the value of the current $k$-th results, this branch is abandoned.

An advantage of the second strategy is that it visits the minimal set of nodes needed to reach the best results, and that it finds the output nodes directly in decreasing relevance order. A disadvantage is that it spends space and time for the priority queue.

Then, we will process the children of the tuple of the second children and insert them into the priority queue.*For example for the second strategy, in Figure 10, we will peek the root and process its children, saving in the priority queue:*

1. *the second children with priority 13.*
2. *the fourth children with priority 8.*
3. *the first children with priority 6.*
4. *the third children with priority 5.*

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

*Although the first strategy uses less space than the second, it could waste some time when we enter to a promising subtree and then, its leaves had not as many results (partial results) or there were no nodes with the highest priority as we thought. Furthermore, when we are working with ranked results, the first k results in the queue are not necessarily mean the top k results, so we still have to visit other siblings to ensure we have computed the most important nodes. Still, are not going to access to a node with a smaller priority of our k nodes computed so far.*

*With the second strategy, as we can see, we do not have the problem mentioned before: if we notice the node is not behaving as we expected, we can stop visiting its children and go to its siblings or to other nodes with a higher priority. Another advantage is that, when we are computing ranked results and we arrive to a leaf, we know for sure that it has the highest priority, so we can output it immediately. However, it uses more space to store the entire queue.*

### 3.2 Lazy version

*For the AND operation, if a quadrant of a quadtree is empty, the result of that quadrant will also be empty. This can be generalized to other operations in the relational algebra: in general, we do not need to compute all the branches of the tree, only evaluate what is needed. The lazy qdags [2] implement this idea by maintaining the syntax tree of a query, where the leaves are the compressed quadtrees and the internal nodes are operators. This enables a mechanism to progressively obtain results in a more general context. For this thesis, we want to implement this tree (see Figure 13) and evaluate its performance.*

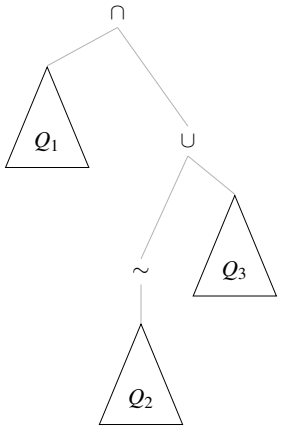


Figure 13: Example syntax tree.

*Lazy qdags could improve space and time-complexity for some operations, while for some others it could increase the time cost. We expect it to retrieve the first results faster in a more general context.*

### 3.3 Both versions

*As related topic, it is also worth to study, for sparse grids with a higher dimension, the alternative for the $k^2$-tree that replaces the bitvector by very sparse bitvectors to save space, though this could increase time complexity.*

## 4 Research questions

- *Which is the best way to compute the number of leaves of a node?*
- *How can partial/ranked results affect time and space in the join operation?*
- *Which technique used in partial/ranked results will work better?*
- *Could lazy qdags implement the relational algebra with reasonable efficiency?*
- *What technique (classic or lazy qdags) will perform better in which scenarios?*
- *Is it possible to improve time and/or space using very sparse bitvectors on sparse grids?*

# 5 Hypothesis

**H1** *The **partial results** join algorithm will improve time to obtain the first results, but it will increase the total time and space complexity, yet it will still guarantee worst-case optimality. The **ranked results** join algorithm will obtain the k best results faster than the baseline of obtaining all and choosing the best, for small enough k.*

**H2** *Lazy qdags will increase time to compute a complete query, but it will obtain the first results faster, and it will implement in a reasonably efficient way the full relational algebra.*

# 6 Goals

## 6.1 General goal

*The main goal of this work is to study and develop new versions of a compact data structure (qdags) that extend the basic multi-way join algorithm towards ranked retrieval, gradual retrieval, and the full relational algebra.*

## 6.2 Specific goals

*The goal for the first part of this thesis is to modify the original multi-join algorithm for compressed quadtrees, and perform it in a certain order to either return partial results in less time, or to take into account the node's priority to output ranked results, while still guaranteeing wco times. The goal for the second part of this thesis is to implement the lazy qdags and extend it to all relational algebra operations as well. In particular:*

1. *Develop and evaluate a new version of qdags that returns first results faster.*
2. *Develop and evaluate a new version of qdags that returns results in an order given by a priority.*
3. *Develop lazy qdags and extend them to support full relational algebra.*

# 7 Methodology

*First of all, it is necessary to do a bibliographic search on the state-of-the-art of compressed quadtrees and joins in databases.*

*For the **partial/ranked results**:*

1. *Implement the LOUDS-based variant that outputs more results first.*
2. *Evaluate the different heuristics as full evaluation.*
3. *Implement the DFUDS-based version and evaluate it.*
4. *Implement the versions for returning most important results.*
5. *Evaluate it and determine where it outperforms a basic solution.*

*For the **lazy version**:*

1. *Design and implement the syntax tree with operators as internal nodes and quadtrees as leaves, for the case of joins.*
2. *Evaluate its performance as full evaluation.*
3. *Extend with selection and compare lqdags with other systems in general Basic Graph Patterns (BGP).*
4. *Extend to the full relational algebra.*
5. *Implement the output with sparse bitvectors and compare.*

# 8 Expected results

*We expect to extend qdags functionalities to efficiently:*

- *Return first results faster.*
- *Return prioritized results.*
- *Expand qdags to full BGPs, Boolean queries (guaranteeing worst-case optimality) and relation algebra (not guaranteeing wco).*

# 9 Contributions

*In general, we will study and implement a version of the qdags that returns partial results in less time and another version that allows return ranked results. We will also implement the lazy version of the qdags and study its time and space. In general, we will extend the use of qdags. Additionally, the code with all the implementations will remain in a public repository.*

# References


*[1] Diego Arroyuelo, Aidan Hogan, Gonzalo Navarro, Juan L. Reutter, Javiel Rojas-Ledesma & Adrián Soto (2021):* Worst-Case Optimal Graph Joins in Almost No Space. *In: Proceedings of the 2021 International Conference on Management of Data, SIGMOD '21, Association for Computing Machinery, New York, NY, USA, p. 102–114, doi:10.1145/3448016.3457256. Available at* `https://doi.org/10.1145/3448016.3457256`.

*[2] Diego Arroyuelo, Gonzalo Navarro, Juan L. Reutter & Javiel Rojas-Ledesma (2022):* Optimal Joins Using Compressed Quadtrees. *ACM Trans. Database Syst. 47(2), doi:10.1145/3514231. Available at* `https://doi.org/10.1145/3514231`.

*[3] Albert Atserias, Martin Grohe & Dániel Marx (2013):* Size Bounds and Query Plans for Relational Joins. *SIAM J. Comput. 42(4), p. 1737–1767, doi:10.1137/110859440. Available at* `https://doi.org/10.1137/110859440`.

*[4] David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman & S. Srinivasa Rao (2005):* Representing Trees of Higher Degree. *Algorithmica 43(4), p. 275–292.*

*[5] Nieves R. Brisaboa, Susana Ladra & Gonzalo Navarro (2014):* Compact Representation of Web Graphs with Extended Functionality. *Inf. Syst. 39, p. 152–174, doi:10.1016/j.is.2013.08.003. Available at* `https://doi.org/10.1016/j.is.2013.08.003`.

*[6] D. R. Clark (1996):* Compact PAT Trees. *Ph.D. thesis, University of Waterloo, Canada.*

[7] *Olaf Hartig & Jorge Pérez (2018):* Semantics and Complexity of GraphQL. *In: Proceedings of the 2018 World Wide Web Conference, WWW '18, International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, p. 1155–1164, doi:10.1145/3178876.3186014. Available at* `https://doi.org/10.1145/3178876.3186014`.

[8] *Gísli R. Hjaltason & Hanan Samet (1998):* Incremental Distance Join Algorithms for Spatial Databases. *In: Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data, SIGMOD '98, Association for Computing Machinery, New York, NY, USA, p. 237–248, doi:10.1145/276304.276326. Available at* `https://doi.org/10.1145/276304.276326`.

[9] *Gísli R. Hjaltason & Hanan Samet (1998):* Incremental Distance Join Algorithms for Spatial Databases. *SIGMOD Rec. 27(2), p. 237–248, doi:10.1145/276305.276326. Available at* `https://doi.org/10.1145/276305.276326`.

[10] *Aidan Hogan, Cristian Riveros, Carlos Rojas & Adrián Soto (2019):* 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, Springer-Verlag, Berlin, Heidelberg, p. 258–275, doi:10.1007/978-3-030-30793-*$6_1$*5. Available at* `https://doi.org/10.1007/978-3-030-30793-6_15`.

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

[12] *Nikolaos Karalis, Alexander Bigerl & Axel-Cyrille Ngonga Ngomo (2023):* Native Execution of GraphQL Queries over RDF Graphs Using Multi-Way Joins. *In: Knowledge Graphs: Semantics, Machine Learning, and Languages, IOS Press, pp. 77–93.*

[13] *Mahmoud Abo Khamis, Hung Q. Ngo, Christopher Ré & Atri Rudra (2016):* Joins via Geometric Resolutions: Worst Case and Beyond. *ACM Trans. Database Syst. 41(4), doi:10.1145/2967101. Available at* `https://doi.org/10.1145/2967101`.

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

[15] *J. I. Munro & V. Raman (2001):* Succinct Representation of Balanced Parentheses and Static Trees. *SIAM Journal on Computing 31(3), pp. 762–776.*

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

[17] *Hung Q. Ngo, Ely Porat, Christopher Ré & Atri Rudra (2018):* Worst-Case Optimal Join Algorithms. *J. ACM 65(3), doi:10.1145/3180143. Available at* `https://doi.org/10.1145/3180143`.

[18] *Hung Q Ngo, Christopher Ré & Atri Rudra (2014):* Skew Strikes Back: New Developments in the Theory of Join Algorithms. *SIGMOD Rec. 42(4), p. 5–16, doi:10.1145/2590989.2590991. Available at* `https://doi.org/10.1145/2590989.2590991`.

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

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

[21] *Mihalis Yannakakis (1981):* Algorithms for Acyclic Database Schemes. *In: Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7, VLDB '81, VLDB Endowment, p. 82–94.*