Optimal Joins using Compressed Quadtrees

DIEGO ARROYUELO, DIINF, UTFSM & IMFD, Chile
GONZALO NAVARRO, DCC, U. of Chile & IMFD, Chile
JUAN L. REUTTER, DCC, PUC & IMFD, Chile
JAVIEL ROJAS-LEDESMA, DCC, U. of Chile & IMFD, Chile,

Worst-case optimal join algorithms have gained a lot of attention in the database literature. We now count several algorithms that are optimal in the worst case, and many of them have been implemented and validated in practice. However, the implementation of these algorithms often requires an enhanced indexing structure: to achieve optimality one either needs to build completely new indexes, or must populate the database with several instantiations of indexes such as B+-trees. Either way, this means spending an extra amount of storage space that is typically one or two orders of magnitude more than what is required to store the raw data.

We show that worst-case optimal algorithms can be obtained directly from a representation that regards the relations as point sets in variable-dimensional grids, without the need of any significant extra storage. Our representation is a compressed quadtree for the static indexes, and a quadtree built on the fly that shares subtrees (which we dub a qdag) for intermediate results. We develop a compositional algorithm to process full join queries under this representation, which simulates navigation of the quadtree of the output, and show that the running time of this algorithm is worst-case optimal in data complexity.

We implement our index and compare it experimentally with state-of-the-art alternatives. Our experiments show that our index uses even less space than what is needed to store the data in raw form (and replaces it), and one or two orders of magnitude less space than the other indexes. At the same time, our query algorithm is competitive in time, even sharply outperforming other indexes in various cases.

Finally, we extend our framework to evaluate more expressive queries from relational algebra, including not only joins and intersections but also unions and negations. To obtain optimality on those more complex formulas, we introduce a lazy version of qdags we dub lqdags, which allow us navigate over the quadtree representing the output of a formula while only evaluating what is needed from its components. We show that the running time of our query algorithms on this extended set of operations is worst-case optimal under some constraints. Moving to full relational algebra, we also show that lqdags can handle selections and projections. While worst-case optimality is no longer guaranteed, we introduce a partial materialization scheme that extends results from Deep and Koutris regarding compressed representation of query results.

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

Additional Key Words and Phrases: Join algorithms, Compact data structures, Quadtrees, AGM bound

1 INTRODUCTION

The state of the art in query processing has recently been shaken by a new generation of join processing algorithms with strong optimality guarantees based on the AGM bound of queries: the maximum size of the

Authors' addresses: Diego Arroyuelo, darroyue@@inf.utfsm.cl, DIINF, UTFSM & IMFD, Chile, , , , ; Gonzalo Navarro, gnavarro@dcc.uchile.cl, DCC, U. of Chile & IMFD, Chile, , , , ; Juan L. Reutter, , DCC, PUC & IMFD, Chile, , , , ; Javiel Rojas-Ledesma, , DCC, U. of Chile & IMFD, Chile, , ,

0362-5915/2022/8-ART111 $15.00
https://doi.org/10.1145/3514231

output of the query over all possible relations with the same cardinalities [6]. One of the basic principles of these algorithms is to disregard the traditional notion of a query plan favoring a strategy that not only takes into account the size of the relations in the database but can also take advantage of the structure of the query [31, 34].

Our focus is on join processing algorithms that compute all answers of a given query over a database instance, and specifically in those algorithms that are guaranteed to run in time bounded by the AGM bound of the query. Several of these algorithms have been implemented and tested in practice with positive results [16, 35], especially when handling queries with several joins. Because they differ from what is considered standard in relational database systems, the implementation of these algorithms often requires additional data structures, a database that is heavily indexed, or heuristics to compute the best computation path given the indexes that are present. For example, algorithms such as Leapfrog [41], Minesweeper [32], or InsideOut [19] must select a *global order* on the attributes, and assume that relations are indexed in a way that is consistent with this order [35]. However, an ordering that is good for one query may induce a sub-optimal performance on a different query [32]. Thus, in practice, these algorithms need to work with several possible attribute orderings, which is commonly achieved with several combinations of B+ trees or other indexes [16]. On the other hand, more involved algorithms such as Tetris [18] or Panda [20] require heavier data structures that allow reasoning over potential tuples in the answer, and as far as we know there is no evidence that these heavier structures can be successfully deployed in practice.

Our goal is to develop worst-case optimal join algorithms that eliminate the need to store additional indexes in the database and compete in practice with the current alternatives. The key here is the combination of good theoretical and practical performance: current worst-case optimal solutions depending on heavily indexed databases would remain optimal if the indexes were built on-the-fly before evaluating each query (it usually takes linear time in data complexity to do so). Such index construction, however, is in practice orders of magnitude slower than indexed query evaluation (minutes to hours versus milliseconds to seconds, see our experiments). Practical solutions are thus forced to compute and store the indexes beforehand so as to achieve competitive query evaluation times. We address this issue by resorting to compact data structures [27]: representations using a nearly-optimal amount of space –indeed, almost none on top of compactly storing the raw data– while supporting all operations we need to answer join queries in worst-case optimal time, *without any need of further indexing*.

We show that worst-case optimal algorithms can be obtained when one assumes that the input data is represented as quadtrees, and stored under a compact representation for cardinal trees [9]. Quadtrees are geometric structures used to represent data points in grids of size $\ell \times \ell$ (which can be generalized to any dimension). Thus, a relation $R(\mathcal{A})$ with attributes $\mathcal{A} = \{A_1, \ldots, A_d\}$ can be naturally viewed as a set of points over grids of dimension d, one point per tuple of R: the value of each attribute A_i is the i-th coordinate of the corresponding point.

To support queries under this representation, our main tool is a new version of quadtrees, which we denote qdags, where some nodes may share complete subtrees. Using qdags, we can reduce the computation of a full join query $J = R_1 \bowtie \cdots \bowtie R_n$ with d attributes, to an algorithm that first extends the quadtrees for $R_1, \ldots, R_n$ into qdags of dimension d, and then intersects them to obtain a quadtree. Our first result shows that such algorithm is indeed worst-case optimal:

Theorem 1.1. *Let $R_1(\mathcal{A}_1), \ldots, R_n(\mathcal{A}_n)$ be n relations. We can then represent each relation R_i using $|\mathcal{A}_i| + 2 + o(1)$ words per entry, so that the join $R_1 \bowtie \cdots \bowtie R_n$ can be computed in $\tilde{O}(AGM)$ time*[1].

Note that just storing the tuples in every R_i requires $|\mathcal{A}_i|$ words per entry, thus our representation adds only a small extra space of basically two words per tuple. Instead, any classical index on the raw data (such as hash

[1]The notation $\tilde{O}$ hides poly-log S factors, S being the total input size, as well as factors that just depend on d (the number of attributes) and n (the query size), which are assumed to be constant. We provide a precise bound in Section 3.3.

tables or B+-trees) would pose a linear extra space, $O(|\mathcal{A}_i|)$ words, often multiplied by a non-negligible constant (especially if one stores multiple indexes on the data).

Our join algorithm works in a rather different way than the most popular worst-case optimal algorithms. To illustrate this, consider the triangle query $J = R(A,B) \bowtie S(B,C) \bowtie T(A,C)$. The most common way of processing this query optimally is to follow what Ngo et al. [34] define as the *generic* algorithm: select one of the attributes of the query (say A), and iterate over all elements $a \in A$ that could be an answer to this query, that is, all $a \in \pi_A(R) \cap \pi_A(T)$. Then, for each of these elements, iterate over all $b \in B$ such that the tuple (a,b) can be an answer: all (a,b) in $(R \bowtie \pi_B(S)) \bowtie \pi_A(T)$, and so on. Instead, with quadtrees we divide the output space (which if the domain of the attributes is $[0, \ell-1]$ corresponds to a grid of size ℓ^3) into 8 subspaces (or subgrids of size $(\ell/2)^3$), and for each of these we recursively evaluate the query. This can be regarded as traversing the output space, while computing only what is needed from the joined relations in order to proceed at each step.

In general, the algorithm of Ngo et al. [34] chooses a subset of the attributes at each step, not necessarily one. Our algorithm can also be understood as splitting each attribute A_i into $\log \ell$ binary attributes[2] $A_i^1, \ldots, A_i^{\log \ell}$ that hold the highest to the lowest bit of A_i, and then using the generic algorithm by choosing all the attributes A_i^1 together, for all i, then all the attributes A_i^2, and so on. As it turns out, this strategy is as good as the generic strategy defined by Ngo et al. [34] on the original attributes, with the advantage that it can always use the same ordering.[3]

Our join algorithm boils down to two simple operations on quadtrees: an Extend operation that lifts the quadtree representation of a grid to a higher-dimensional grid, and an And operation that intersects trees. The strategy can be extended to other relational operations. For example, the synchronized Or of two quadtrees gives a compact representation of their union, and complementing the quadtree values implements a Not operation. We integrate all these operations in a single framework, and use it to answer more complex queries given by the combination of these expressions, in a fully compositional form.

To support these more complex queries in optimal time we traverse the output space using lazy evaluation. The idea is to be able to delay the computation of an expression until we know such computation is needed to navigate the output. For this purpose we introduce lazy qdags, or lqdags for short, in which nodes may be additionally labeled with query expressions. To analyze our framework we extend the idea of a worst-case optimal algorithm to arbitrary queries: If a *worst-case optimal algorithm* to compute the output of a formula F takes time t over relations $R_1, \ldots, R_n$ of a database D, then there exists a database D' with relations $R'_1, \ldots, R'_n$ of sizes $|R'_i| = O(|R_i|)$, and their complements of sizes $|\overline{R'_i}| = O(|\overline{R_i}|)$, where the output of F over $R'_1, \ldots, R'_n$ is of size $\Omega(t)$. Our framework remains worst-case optimal with lqdags considering joins, union, and negation operators under some conditions:

Theorem 1.2. *Let $R_1(\mathcal{A}_1), \ldots, R_n(\mathcal{A}_n)$ be n relations. Let F be a relational algebra formula composed by join, union and complement operations over the relations R_i, for all $1 \leq i \leq n$, and where no relation appears both complemented and not complemented in F. Each relation R_i can then be represented using $|\mathcal{A}_i| + 2 + o(1)$ words per entry, so that F can be evaluated in worst-case optimal time in data complexity.*

Consider, for example, the query $J' = R(A,B) \bowtie S(B,C) \bowtie \overline{T}(A,C)$, which joins R and S with the complement $\overline{T}$ of T. One could think of two ways to compute this query. The first is just to join R and S and then see which of the resulting tuples are not in T. But if T is dense ($\overline{T}$ is small), it may be more efficient to first compute $\overline{T}$ and then proceed as on the usual triangle query. Seen at a very high level, our algorithm is optimal because it can choose locally between both strategies: by dividing the output into quadrants it finds dense regions of T in which

[2]Our logarithms are to the base 2 by default.

[3]The algorithm Tetris [18] also works with a geometrical representation, but uses it to refine the search for new tuples, not as a way to recursively divide the output space.

computing $\overline{T}$ is cheaper (and it only works towards those few cells that exist in $\overline{T}$), while in sparse regions the algorithm first computes the join of R and S (i.e., it only works towards those few cells that exist in $R \bowtie S$).

We also show that our framework can be extended to handle the full relational algebra, though in this case worst-case optimality is not guaranteed. However, using lqdags to process relational algebra queries has other potential advantages: instead of fully materializing the output, we provide a parameterizable compressed representation, from which the results can be later retrieved with bounded delay. This extends previous results [10] to the full relational algebra.

Our framework is the first in combining worst-case time optimality with the use of compact data structures. The latter can lead to improved performance in practice, because relations can be stored in faster memory, higher in the memory hierarchy [27]. This is especially relevant when the compact representation fits in main memory while a heavily indexed representation requires resorting to the disk, which is orders of magnitude slower. For systems that maintain the database in the aggregate main memory of a cluster, a compact representation leads to using fewer computers, thus reducing hardware, communication, and energy costs, while improving performance.

In practice. To evaluate how our approach fares in practice, we provide a prototype implementation of the join algorithm mentioned in Theorem 1.1. Our implementation stores data as compressed quadtrees, and computes the compressed quadtree representing the result of the joins.

We test our prototype using two different benchmarks for graph databases, namely the wikidata SPARQL benchmark [16] and a set of queries taken from SNAP [35]. Graph databases give us a good way of testing worst-case optimal algorithms in practice, because graph queries usually involve several joins [35]. Further, the selected benchmarks provide a wide range of complex join queries, and have already been used to test other worst-case optimal join implementations.

We compare against other worst-case optimal implementations [1, 16], as well as some leading graph database systems. The advantage of using compact data structures is immediately seen when comparing the size of our representation against all other options: We reduce the storage size used by graph systems by a factor of 10–20, and this factor is around 250 for EmptyHeaded [1].

Our results show that our implementation of the qdag join algorithm is competitive, in terms of performance, with other worst-case optimal implementations, outperforming standard graph systems in many cases. Qdags excel on queries involving up to 4 attributes, outperforming more sharply the non-wco systems on cyclic queries. Considering the amount of storage space that is gained by using quadtrees, we find these results remarkable. On the other hand, we observe that the performance of our implementation quickly degrades as the number of join attributes increases, becoming much slower and with much higher variance than the alternatives. This is somewhat expected, because our query times depend exponentially on the dimension. All of this suggests that qdags excel in applications in which the number of join attributes is low, or perhaps as a component of a more involved algorithm dealing with tree-decomposition of queries, such as EmptyHeaded [1]. We leave these questions, as well as the implementation of lqdags and of dynamic versions supporting insertions and deletions of tuples, for future work.

Organization of the paper. In Section 2 we fix the notation on quadtrees and explain their compressed representation. The algorithm for multiway join queries is introduced together with qdags in Section 3, and our full framework is introduced together with lazy qdags, first in Section 4 for Boolean queries and then in Section 5 for the complete relational algebra. Section 6 describes our implementation of the multiway join algorithm over qdags and Section 7 its experimental comparison with other state-of-the-art systems and prototypes. We conclude in Section 8. Appendix A compares this article with its conference version; the others give more detailed data on implementation and experiments.

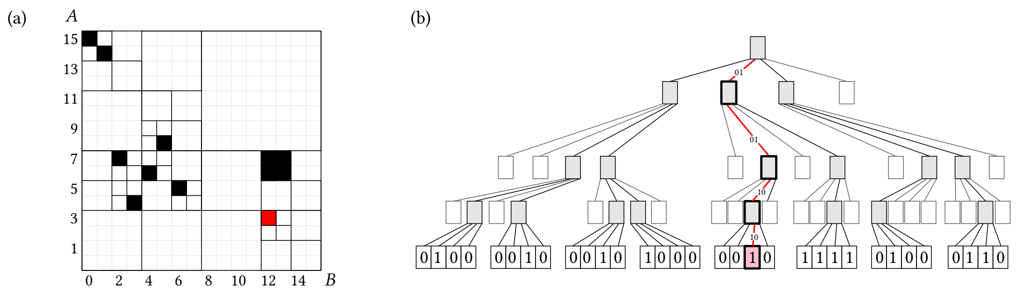


Fig. 1. A quadtree representing $R(A, B) = \{(4, 3), (7, 2), (5, 6), (6, 4), (3, 12), (6, 12), (6, 13), (7, 12), (7, 13), (8, 5), (14, 1), (15, 0)\}$. (a) Representation of $R(A, B)$ in a $2^4 \times 2^4$ grid, with the hierarchical partition defining the quadtree. The black cells correspond to points in R. (b) The quadtree representing R. The integers are in the last level, and the internal nodes are grayed. The shadowed integer 1 of the tree corresponds to the point $(a, b) = (3, 12)$, highlighted in red in the grid. Concatenating the labels in the path down to the integer yields the bit-string '0**1**0**1**1**0**1**0**' which encodes the first (resp., second) coordinate of (a, b) in the bits at odd (resp., even) positions ($a = 3 = 0011, b = 12 = 1100$).

2 QUADTREES

A Region Quadtree [12, 37] is a data structure used to store (pairwise different) points in two-dimensional grids of size $\ell \times \ell$. We focus on the variant called MX-Quadtree [37, 43], which can be described as follows. Assume for simplicity that ℓ is a power of 2. If $\ell = 1$, then the grid has only one cell and the quadtree is an *integer* 1 (if the cell has a point) or 0 (if not). For $\ell > 1$, if the grid has no points, then the quadtree is a *leaf*. Otherwise, the quadtree is an *internal node* with four children, each of which is the quadtree of one of the four $\ell/2 \times \ell/2$ quadrants of the grid.

Assume each data point is described using the binary representation of each of its coordinates (i.e., as a pair of $\log \ell$-bit strings). We order the grid quadrants so that the first contains all points with coordinates of the form $(0 \cdot c_a, 0 \cdot c_b)$, for $\log \ell - 1$ bit vectors c_a and c_b, the second contains points $(0 \cdot c_a, 1 \cdot c_b)$, the third $(1 \cdot c_a, 0 \cdot c_b)$, and the last quadrant stores the points $(1 \cdot c_a, 1 \cdot c_b)$, where '$\cdot$' denotes concatenation of bits. If the corresponding children of internal nodes are labeled 00, 01, 10, and 11, then the concatenation of the labels leading from the root to the node storing the integer 1 of a point (a, b) interlaces the ℓ-bit representations of the coordinates a and b. The path then implicitly represents the point (a, b). Figure 1 shows a grid and its deployment as a quadtree.

Quadtrees can be generalized to higher dimensions. A quadtree of dimension d is a tree representing data points in a d-dimensional grid G of size ℓ^d. In this case, a nonempty grid with side $\ell > 1$ corresponds to an internal node with 2^d children, which represent the 2^d subspaces spanning from combining the first bit of each dimension. Generalizing the case $d = 2$, the children are ordered using the Morton [25] partitioning of the grid: a sequence of 2^d subgrids of size $(\ell/2)^d$ in which the i-th subgrid of the partition, for $0 \le i < 2^d$, labeled with the binary encoding l_i of i, is defined by all the points with coordinates $(b_{c_1}, \ldots, b_{c_d})$ in which the word formed by concatenating the first bit of each string b_{c_j} is precisely the string l_i.

A quadtree with p points has p integers 1 in its last level, and thus at most $p \log \ell$ internal nodes (along the paths of length $\log \ell$ leading from the root to those integers). Since each internal node can be distinguished from its siblings with its label of d bits, $pd \log \ell$ bits are sufficient in principle to encode the quadtree structure. Note that this is also the space needed by a plain representation of p points of d coordinates in $[0, \ell - 1]$.

Indeed, each point is uniquely identified by the sequence of $\log \ell$ d-bit labels in the path leading from the root to its integer 1. For the point $(3, 12)$ in Figure 1, this sequence is 01 01 10 10 = 1122. A trie built on the p resulting sequences then corresponds to the gray and the integer-1 nodes of the right of Figure 1. We store this trie using a compact representation for cardinal trees introduced by Benoit et al. [7, Thm. 4.3], which requires essentially $d + 2 + o(1)$ bits per node and performs the needed tree traversal operations in constant time.

LEMMA 2.1. *(cf. Benoit et al. [7], Thm. 4.3) Let Q be a quadtree storing p points in d dimensions with integer coordinates in the interval $[0, \ell - 1]$. Then, there is a representation of Q that uses $(d + 2 + o(1))p \log \ell + O(\log d)$ bits, can be constructed in linear expected time,[4] and supports constant time parent-children navigation on the tree. More precisely, this representation provides integer identifiers for the quadtree nodes so that, given the identifier of a node and the label of a desired child, it returns in constant time the identifier of that child, if it exists, and a null value otherwise.*

Note that the space overhead of this representation, on top of the $dp \log \ell$ bits needed to represent the raw data, is essentially 2 extra numbers (of $\log \ell$ bits) per point. Further, this structure can *replace* the raw data, because it can recover the points without the need of storing them separately.

From now on, by quadtree we refer to this compact representation. Next, we show how to represent relations using quadtrees and evaluate join queries over this representation.

3 MULTI-WAY JOINS USING QDAGS

We assume for simplicity that the domain $\mathcal{D}(A)$ of an attribute A consists of all binary strings of length $\log \ell$, representing the integers in $[0, \ell - 1]$, and that ℓ is a power of 2.

A relation $R(\mathcal{A})$ with attributes $\mathcal{A} = \{A_1, \ldots, A_d\}$ can be naturally represented as a quadtree: simply interpret each tuple in $R(\mathcal{A})$ as a data point over a d-dimensional grid with ℓ^d cells, and store those points in a d-dimensional quadtree. Thus, using quadtrees one can represent the relations in a database using compact space. The convenience of this representation to handle restricted join queries with naive algorithms has been demonstrated practically on RDF stores [3]. In order to obtain a general algorithm with provable performance, we introduce qdags, an enhanced version of quadtrees, together with a new algorithm to efficiently evaluate join queries over the compressed representations of the relations.

We start with an example to introduce the basics behind our algorithms and argue for the need of qdags. We then formally define qdags and explore their relation with quadtrees. Finally, we provide a complete description of the join algorithm and analyze its running time.

3.1 The triangle query: quadtrees vs qdags

Let $R(A, B)$, $S(B, C)$, $T(A, C)$ be relations over the attributes $\{A, B, C\}$, and consider the triangle query $R(A, B) \bowtie S(B, C) \bowtie T(A, C)$. The basic idea of the algorithm is as follows: we first compute a quadtree Q^*_R that represents the cross product $R(A, B) \times \text{All}(C)$, where $\text{All}(C)$ is a relation with an attribute C storing all elements in the domain $[0, \ell - 1]$. Likewise, we compute Q^*_S representing $S(B, C) \times \text{All}(A)$, and Q^*_T representing $T(A, C) \times \text{All}(B)$. Note that these quadtrees represent points in the three-dimensional grid that has a cell for every possible value in $\mathcal{D}(A) \times \mathcal{D}(B) \times \mathcal{D}(C)$, where we assume that the domains $\mathcal{D}(\cdot)$ of the attributes are all $[0, \ell - 1]$. Finally, we traverse the three quadtrees in synchronization, building a new quadtree that represents the intersection of Q^*_R, Q^*_S, and Q^*_T. This quadtree represents the desired output because

$$R(A, B) \bowtie S(B, C) \bowtie T(A, C) = (R(A, B) \times \text{All}(C)) \cap (S(B, C) \times \text{All}(A)) \cap (T(A, C) \times \text{All}(B)).$$

Though this algorithm is correct, it can perform poorly in terms of space and running time. The size of Q^*_R, for instance, can be considerably bigger than that of R, and even than the size of the output of the query. If, for example, the three relations have N elements each, the size of the output is bounded by $N^{3/2}$ [6], while building

[4]The construction time is expected because it involves perfect hash functions on N elements, providing $O(1)$ time evaluation within $O(N)$ bits. Although they [7] only consider the (easy and practical) randomized construction of such functions, those can be built deterministically in $O(N \log^5 N)$ worst-case time if desired [2].

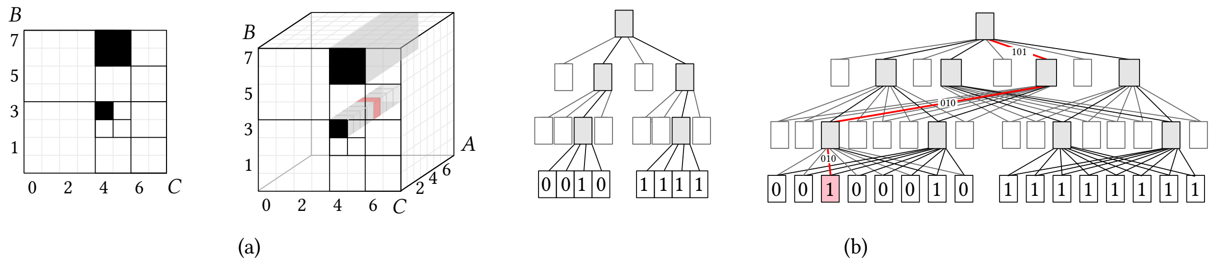


Fig. 2. An illustration of a qdag for $S^*(\{A, B, C\}) = \text{All}(A) \times S(B, C)$, with $S(B, C) = \{(3, 4), (6, 4), (6, 5), (7, 4), (7, 5)\}$. (a) A geometric representation of $S(B, C)$ (left), and $S^*(\{A, B, C\})$ (right). (b) A quadtree Q_S for $S(B, C)$ (left), and the directed acyclic graph induced by the qdag $(Q_S, M = [0, 1, 2, 3, 0, 1, 2, 3])$, which represents $S^*(\{A, B, C\})$. The red cell in (a) corresponds to the point $(4, 3, 4)$. The leaf representing that point in the qdag can be reached following the path highlighted in (b). Note the relation between the binary representation (**1**00,**0**11,**1**00) of $(4, 3, 4)$, and the Morton codes **101**, 011, 010 of the nodes in the path to its integer 1.

Q^*_R costs $\Omega(N\ell)$ time and space. This inefficiency stems from the fact that quadtrees are not powerful enough to represent relations of the form $R^*(\mathcal{A}) = R(\mathcal{A}') \times \text{All}(\mathcal{A} \setminus \mathcal{A}')$, where $\mathcal{A}' \subset \mathcal{A}$, using space close that of a quadtree representing $R(\mathcal{A}')$. Due to its tree nature, a quadtree does not benefit from the regularities that appear in the grid representing $R^*(\mathcal{A})$. To remedy this shortcoming, we introduce qdags, quadtree-based data structures that represent sets of the form $R(\mathcal{A}') \times \text{All}(\mathcal{A} \setminus \mathcal{A}')$ by adding only *constant* additional space to the quadtree representing $R(\mathcal{A}')$, for any $\mathcal{A}' \subseteq \mathcal{A}$.

A qdag is an *implicit* representation of a d-dimensional quadtree Q (that has certain regularities) using only a reference to a d'-dimensional quadtree Q', with $d' \leq d$, and an auxiliary *mapping function* that defines how to use Q' to simulate navigation over Q. Qdags can then represent relations of the form $R(\mathcal{A}') \times \text{All}(\mathcal{A} \setminus \mathcal{A}')$ using only a reference to a quadtree representing $R(\mathcal{A}')$, and a constant-space mapping function (more precisely, with 2^d entries).

To illustrate how a qdag works, consider a relation $S(B, C)$, and let Q^*_S be a quadtree representing $S^*(A, B, C) = \text{All}(A) \times S(B, C)$. Since Q^*_S stores points in the ℓ^3 cube, each node in Q^*_S has 8 children. As $\text{All}(A)$ contains all ℓ elements, for each original point (b, c) in S, S^* contains ℓ points corresponding to elements $(0, b, c), \ldots, (\ell-1, b, c)$. We can think of this as *extending* each point in S to a box of dimension $\ell \times 1 \times 1$. With respect to Q^*_S, this implies that, among the 8 children of a node, the last 4 children will always be identical to the first 4, and their type (leaf, internal, or integer) will be the same as that of the corresponding nodes in the quadtree Q_S representing S. In other words, each of the four subgrids $1a_1a_2$ is identical to the subgrid $0a_1a_2$, and these in turn are identical to the subgrid a_1a_2 in Q_S when projected to dimensions B, C (see Figure 2 for an example). Thus, we can implicitly represent Q^*_S by the pair $(Q_S, M = [0, 1, 2, 3, 0, 1, 2, 3])$ as follows: the types of the roots of Q^*_S and Q_S are the same; if they are integer nodes then both have the same content, and if they are internal nodes then the i-th child of the root of Q^*_S is represented recursively by the pair (Q_i, M) where Q_i is the $M[i]$-th child of the root of Q_S.

3.2 Qdags for relational data

We now introduce a formal definition of the qdags, and describe the algorithms that allow the evaluation of multijoin queries in worst-case optimal time.

Definition 3.1 (Qdag). Let $d > 0$ be an integer. A d-dimensional qdag $\mathcal{Q}$ is a pair (Q', M) in which Q' is a d'-dimensional quadtree, for $d' \leq d$, and $M : [0, 2^d - 1] \to [0, 2^{d'} - 1]$ is a *mapping* function.

A d-dimensional qdag can be used to simulate different d-dimensional quadtrees. In particular, we are interested in the quadtree obtained by the recursive procedure defined below.

Algorithm 1 Value (Q)

```
Require: A qdag $Q = (Q', M)$ with grid side $\ell$.
Ensure: The integer 1 if the grid is a single point, 0 if the grid is empty, and ½ otherwise.
1: if $\ell = 1$ then return the integer $Q'$
2: if $Q'$ is a leaf then return 0
3: return ½
```

Algorithm 2 Child (Q, i)

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

Definition 3.2 (Completion of a qdag). Let d, d' be integers such that $d' \leq d$, let Q' be a d'-dimensional quadtree, and let $Q = (Q', M)$ be a d-dimensional qdag. The *completion* Q^* of Q is the unique d-dimensional quadtree recursively defined as follows:

(1) If Q' represents a single unit-size cell (i.e., the root of Q' is an integer node), then the root of Q^* is also an integer node of the same type (either 0 or 1).
(2) If Q' represents an empty grid of points (i.e., the root of Q' is a leaf), then Q^* has also as root a leaf representing an empty grid.
(3) Otherwise, the root r' of Q' must be an internal node, and Q^* has then as root an internal node r^* such that, for all $0 \leq i < 2^d$, the i-th child of r^* is the completion of the qdag $(Q'[M(i)], M)$, where $Q'[j]$ denotes the j-th child of r'.

Definition 3.3 (Identity Mapping). Let $\mathcal{A}$ be a set of attributes. The*identity mapping* of $\mathcal{A}$, denoted $\mathsf{Id}(\mathcal{A})$, is the mapping function $M : [0, 2^{|\mathcal{A}|} - 1] \to [0, 2^{|\mathcal{A}|} - 1]$ such that $M(i) = i$, for all $0 \leq i < 2^{|\mathcal{A}|}$.

We say that a qdag represents the same relation $R(\mathcal{A})$ represented by its completion. Note that, for any d-dimensional quadtree Q representing $R(\mathcal{A})$, one can generate a qdag whose completion is Q by simply using the pair $(Q, \mathsf{Id}(\mathcal{A}))$. Note also that we can use mappings to represent any desired reordering of the attributes.

In terms of representation, a qdag can abstract from the actual representation used for the respective quadtree, as long as there is a way to refer to any node in the quadtree. For instance, when quadtrees are stored using Lemma 2.1, the references to quadtree nodes consist of a pointer to the data structure representing the quadtree, plus the integer identifier of the corresponding trie node in that structure.

For a qdag $Q = (Q', M)$, we denote by $|Q|$ the number of internal nodes in the base quadtree Q', and by $||Q||$ the number of internal nodes in the completion of Q.

Algorithms 1 and 2, based on Definition 3.2, allow us to simulate the navigation over the completion of a qdag in a way that abstracts from the representation of the inner quadtree. Operation Value yields a 0 if and only if the subgrid represented by the qdag is empty (thus the qdag's root is a leaf or an integer 0), a 1 if the qdag represents a full single cell (i.e., the qdag's root is the integer 1), and ½ if its root is an internal node. Operation Child lets us descend by a given child from internal nodes representing nonempty grids. The operations "integer Q'", "Q is a leaf", and "$Q'[j]$" are implemented in constant time on the compact representation of Q': $Q'[j]$ corresponds to descending to the child of Q' by label j. If there is no such child, the structure of Lemma 2.1 returns a null value, which we interpret as an integer node 0 if its subgrid has a single cell, and as a leaf otherwise (in both cases, as a node with Value 0). Finally, a non-null node representing a single cell corresponds to the integer node 1 (with Value 1).

3.2.1 *Operation* Extend.

We introduce an operation to obtain, from the qdag representing a relation R, a new qdag representing the relation R extended with new attributes.

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

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

```
1: create array $M[0, 2^d - 1]$
2: $d \leftarrow |\mathcal{A}|$, $d' \leftarrow |\mathcal{A}'|$
3: for $i \leftarrow 0, \ldots, 2^d - 1$ do
4:     $m_d \leftarrow$ the $d$-bits binary representation of $i$
5:     $m_{d'} \leftarrow$ the projection of $m_d$ to the positions in which the attributes of $\mathcal{A}'$ appear in $\mathcal{A}$
6:     $i' \leftarrow$ the value in $[0, 2^{d'} - 1]$ corresponding to $m_{d'}$
7:     $M[i] \leftarrow M'[i']$
8: return $(Q', M)$
```

Definition 3.4. Let $\mathcal{A}' \subseteq \mathcal{A}$ be sets of attributes, let $R(\mathcal{A}')$ be a relation over $\mathcal{A}'$, and let $Q_R = (Q, M')$ be a qdag that represents $R(\mathcal{A}')$. The operation Extend$(Q_R, \mathcal{A})$ returns a qdag $Q_R^* = (Q, M)$ that represents the relation $R \times \mathrm{All}(\mathcal{A} \setminus \mathcal{A}')$.

To provide intuition on its implementation, let $\mathcal{A}'$ be the set of attributes $\{A, B, D\}$ and let $\mathcal{A} = \{A, B, C, D\}$, and consider $R(\mathcal{A}')$, Q_R and Q_R^* from Definition 3.4. Each node of (the completion of) Q_R has 8 children, while each node of (the completion of) Q_R^* has 16 children. Consider the child at position $i = 12$ of Q_R^*. This node represents the grid with Morton code m_4='**1100**' (i.e., 12 in binary), and contains the tuples whose coordinates in binary start with **1** in attributes A, B and with **0** in attributes C, D. This child has elements if and only if the child with Morton code m_3='**110**' of Q_R (i.e., its child at position $j = 6$) has elements; this child is in turn the $M'[6]$-th child of Q. Note that m_3 results from projecting m_4 to the positions 0,1,3 in which the attributes A, B, D appear in $\{A, B, C, D\}$. Since the Morton code '**1110**' (i.e., 14 in binary) also projects to m_3, it holds that $M[12] = M[14] = M'[6]$. We provide an implementation of the Extend operation for the general case in Algorithm 3. The following lemma states the time and space complexity of our implementation of Extend. For simplicity, we count the space in terms of computer words used to store references to the quadtrees and values of the mapping function M.

Lemma 3.5. *Let $|\mathcal{A}| = d$ in Definition 3.4. Then, the operation* Extend$(Q_R, \mathcal{A})$ *can be supported in time $O(2^d)$ and its output takes $O(2^d)$ words of space.*

Proof. We show that Algorithm 3 meets the conditions of the lemma. The computations of m_d and i' are immaterial (they just interpret a bitvector as a number or vice versa). The computation of m_d' is done with a constant table (that depends only on the database dimension d) of size $O(2^{2d})$: given the d bits of m_d and other d bits telling which attributes of $\mathcal{A}$ are in $\mathcal{A}'$, the table stores the corresponding bitvector m_d'. A naive algorithm without this table runs in time $O(d2^d)$. □

3.3 Join algorithm

Now that we can efficiently represent relations of the form $R(\mathcal{A}') \times \mathrm{All}(\mathcal{A} \setminus \mathcal{A}')$, for $\mathcal{A}' \subseteq \mathcal{A}$, we describe a worst-case-optimal implementation of joins over the qdag representations of the relations. Our algorithm follows the idea discussed for the triangle query: we first extend every qdag to all the attributes that appear in the query, so that they all have the same dimension and attributes. We then compute their intersection, building a quadtree representing the output of the query; see Figure 3 for an illustration. The implementation of this algorithm is surprisingly simple (see Algorithms 4 and 5), yet worst-case optimal, as we prove later on. Using qdags is essential for this result; this algorithm would not be at all optimal if computed over relational instances stored using standard representations such as B+ trees. First, we describe how to compute the intersection of several qdags, and then analyze the running time of the join.

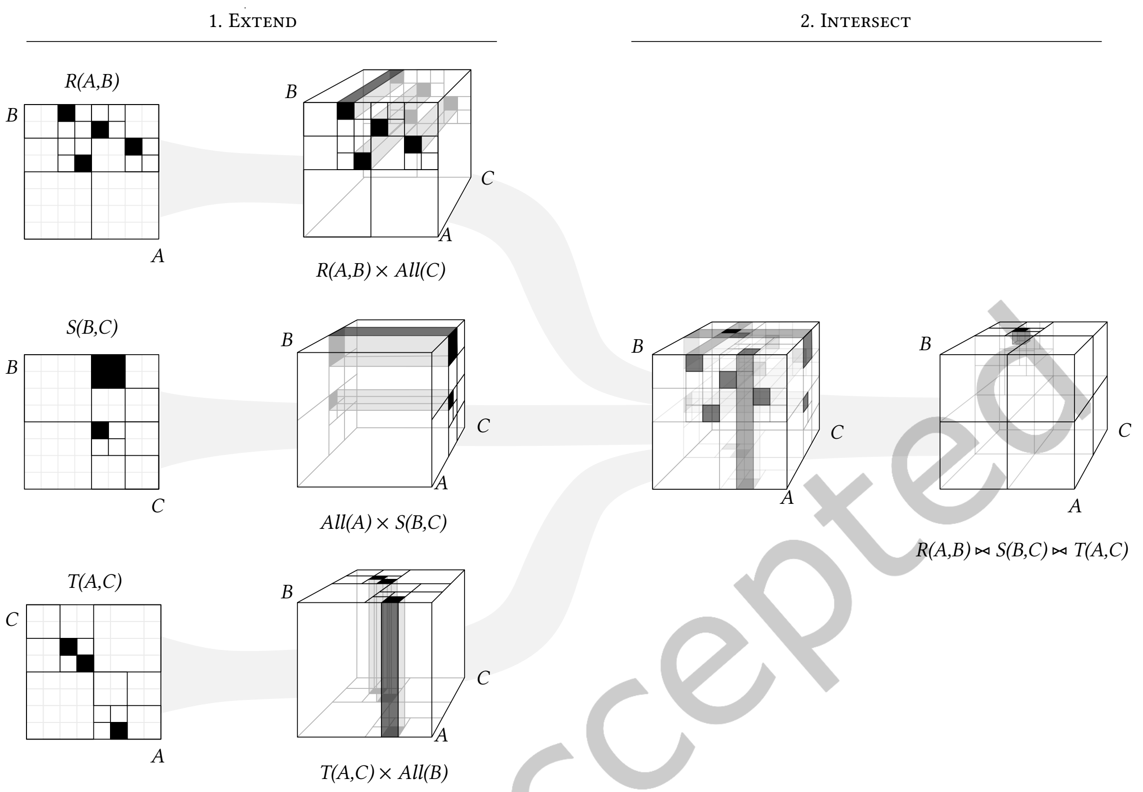


Fig. 3. A graphical representation of the MULTIJOIN algorithm for $R(A, B) \bowtie S(B, C) \bowtie T(A, C)$.

Algorithm 4 MULTIJOIN ($R_1, \ldots, R_n$)

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

Algorithm 5 AND ($Q_1, \ldots, Q_n$)

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

3.3.1 Operation AND. We introduce an operation AND, which computes the intersection of several relations represented as qdags.

Definition 3.6. Let $Q_1, \ldots, Q_n$ be qdags representing relations $R_1, \ldots, R_n$, all over the attribute set $\mathcal{A}$. Operation AND$(Q_1, \ldots, Q_n)$ returns a quadtree Q that represents the relation $R_1 \cap \cdots \cap R_n$.

We implement this operation by simulating a synchronized traversal among the completions $C_1, \ldots, C_n$ of $Q_1, \ldots, Q_n$, respectively, obtaining the quadtree Q that stores the cells that are present in all the quadtrees C_i (see Algorithm 5). We proceed as follows. If $\ell = 1$, then all C_i are integers with values 0 or 1, and Q is an integer equal to the minimum of the n values. Otherwise, if any Q_i represents an empty subgrid, then Q is also a leaf representing an empty subgrid. Otherwise, every C_i is rooted by a node v_i with 2^d children, and so is Q, where the j-th child of its root v is the result of the AND operation of the j-th children of the nodes $v_1, \ldots, v_n$. However, we need a final *pruning* step to restore the quadtree invariants (line 6 of Algorithm 5): if $\textsc{Value}(v_i) = 0$ for all the resulting children of v, then v must become a leaf and the children be discarded. Note that once the quadtree is computed, we can represent it succinctly in linear expected time (Lemma 2.1) so that, for instance, it can be cached for future queries involving the output represented by Q.

3.3.2 Analysis of the algorithm. We compute the output Q of $\textsc{And}(Q_1, \ldots, Q_n)$ in time $O(2^d \cdot (||Q_1|| + \cdots + ||Q_n||))$. More precisely, the time is bounded by $O(2^d n \cdot |Q^+|)$, where Q^+ is the quadtree that would result from Algorithm 5 if we removed the pruning step of line 6. We call this quadtree Q^+ the *non-pruned version* of Q. Although the size of the actual output Q can be much less than that of Q^+, we can still prove that our time is optimal in the worst case. We start with two technical results, the first one (Lemma 3.7) bounding the running time of Algorithm 5 in terms of the maximum number m of internal nodes of some level in Q^+, and the second one (Lemma 3.8) bounding m by the output size of Algorithm 5 over an instance of similar size.

LEMMA 3.7. *Let $Q_1, \ldots, Q_n$ be qdags representing relations $R_1, \ldots, R_n$, all over the attribute set $\mathcal{A}$, and let h be the height of the completions of $Q_1, \ldots, Q_n$. Then the operation $Q = \textsc{And}(Q_1, \ldots, Q_n)$ can be supported in time $O(m \cdot 2^d n \cdot h)$, where m is the maximum number of internal nodes in any level of Q^+, the non-pruned version of Q.*

PROOF. We show that Algorithm 5 meets the conditions of the lemma. Let m_k be the number of nodes of depth k in Q^+, and then $m = \max_{0 \le k < h} m_k$. The number of steps performed by Algorithm 5 is clearly bounded by $n \cdot (\sum_{0 \le k < h} m_k \cdot 2^d) \le n \cdot m \cdot h \cdot 2^d$: at each depth we continue traversing all qdags $Q_1, \ldots, Q_n$ as long as they are all nonempty, and we generate the corresponding nodes in Q^+ (even if at the end some nodes will disappear in Q). The cost incurred in leaves and integer nodes of Q^+ can be charged to their internal parent node. ◻

LEMMA 3.8. *Let $Q_1, \ldots, Q_n$ be n qdags representing relations $R_1(\mathcal{A}_1), \ldots, R_n(\mathcal{A}_n)$, respectively, and let $\mathcal{A} = \bigcup \mathcal{A}_i$. Let $Q = \textsc{And}\Big(\textsc{Extend}(Q_1, \mathcal{A}), \ldots, \textsc{Extend}(Q_n, \mathcal{A})\Big)$, let Q^+ be the non-pruned version of Q, and let m be the maximum number of internal nodes at any level of Q^+. Then there exist relations $R'_1(A_1), \ldots, R'_n(A_n)$ such that:*

- *$|R'_i| \le |R_i|$, for all $1 \le i \le n$;*
- *If the qdags $Q'_1, \ldots, Q'_n$ represent $R'_1(A_1), \ldots, \mathcal{R}'_n(A_n)$, respectively, then the quadtree $Q' = \textsc{And}\Big(\textsc{Extend}(Q'_1, \mathcal{A}), \ldots, \textsc{Extend}(Q'_n$ has at least m integer-1 nodes.*

PROOF. Let the maximum number m of internal nodes be reached at depth $0 \le j < \log \ell$ of Q^+. We construct the relations R'_i, for $1 \le i \le n$, as follows: For a binary string c, let $\mathrm{pre}(c, j)$ denote the first j bits of c. Then, for each relation R_i and each tuple $(c_1, \ldots, c_{d_i})$ in R_i, where $d_i = |\mathcal{A}_i|$, let R'_i contain the tuples $(0^{\log \ell - j}\mathrm{pre}(c_1, j), 0^{\log \ell - j}\mathrm{pre}(c_2, j), \ldots, 0^{\log \ell - j}\mathrm{pre}(c_{d_i}, j))$, corresponding to taking the first j bits of each coordinate and prepending them with a string of $\log \ell - j$ 0s. While this operation may send two tuples in an original relation to a single tuple in the corresponding new one, we still have that each relation R'_i contains at most as many tuples as relation R_i.

Let us show that the quadtree $Q' = \textsc{And}\Big(\textsc{Extend}(Q'_1, \mathcal{A}), \ldots, \textsc{Extend}(Q'_n, \mathcal{A})\Big)$ has at least m integer-1 nodes. Imagine that we represent each R'_i using a qdag Q'_i. Because the first $\log \ell - j$ bits of every tuple component in R'_i are all 0, in the top $\log \ell - j$ levels of (the completion of) Q'_i there will be only one internal node per level, with only the first child of each of these not being a leaf. Moreover, the remaining j levels at the bottom of Q'_i will

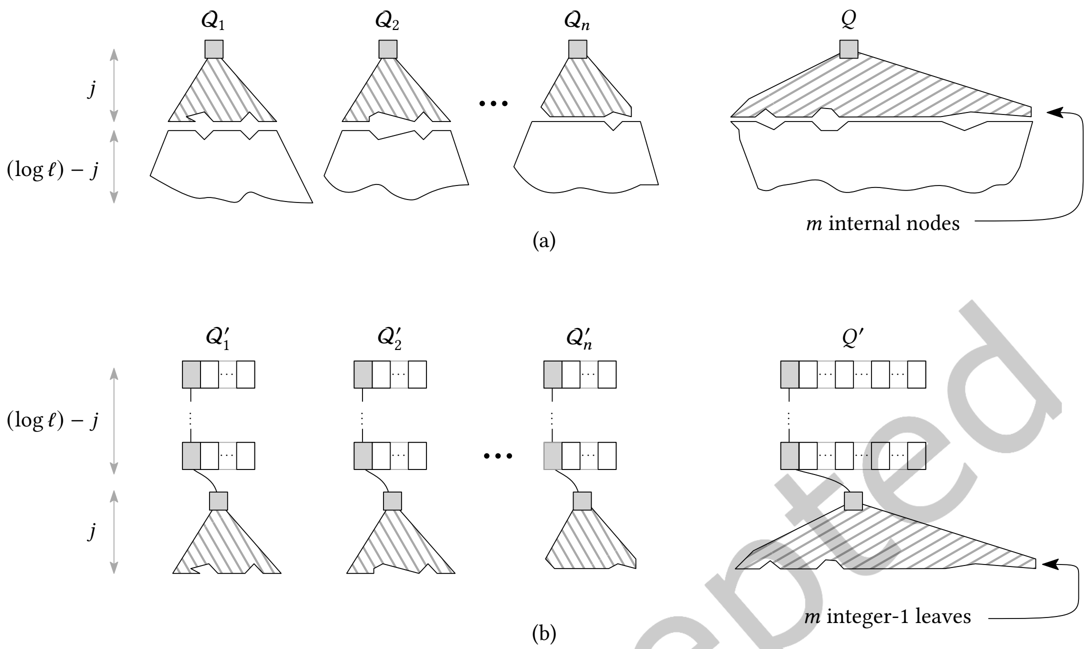


Fig. 4. A graphical representation of the construction for the proof of Lemma 3.8. a) An illustration of the top j-th levels of the qdags $Q_1, \ldots, Q_n$ representing $R_1, \ldots, R_n$, respectively, and the top j-th levels of $Q = \textsc{And}\big(\textsc{Extend}(Q_1, \mathcal{A}), \ldots, \textsc{Extend}(Q_n, \mathcal{A})\big)$. b) The qdags $Q'_1, \ldots, Q'_n$ representing respectively the relations $R'_1, \ldots, R'_n$ generated, and the quadtree $Q' = \textsc{And}\big(\textsc{Extend}(Q'_1, \mathcal{A}), \ldots, \textsc{Extend}(Q'_n, \mathcal{A})\big)$.

be exactly the same as the top j levels of Q_i, except for the last one, in which the internal nodes of Q_i become integer-1 nodes in Q'_i, and the leaves representing empty grids become integer-0 nodes (see Figure 4 for an illustration).

Now suppose that we run Algorithm 5 over $\textsc{Extend}(Q'_1, \mathcal{A}), \ldots, \textsc{Extend}(Q'_n, \mathcal{A})$, and let Q' be its output quadtree. Note that, as before, in the top $\log \ell - j$ levels of Q' there will be only one internal node per level, with only the first child of each of these nodes not being a leaf. Then, since the j levels at the bottom of Q'_i are the same as the top j levels of Q_i, for all $1 \leq i \leq n$, the remaining j levels at the bottom of Q' will be exactly the same as the top j levels of Q, except for the last one, in which the internal nodes of Q at the j-th level become integer-1 nodes in Q'. Therefore, Q' must have at least m integer-1 nodes at the last level. □

Since the running time of Algorithm 4 is dominated by the execution of Algorithm 5, we can use the bounds established in Lemmas 3.7 and 3.8 to prove the worst-case optimality of Algorithm 4. We do this in Theorem 3.9. For a join query J on a database D, we use $2^{\rho^*(J,D)}$ to denote the AGM bound [6] of the query J over D, that is, the maximum size of the output of J over any relational database having the same number of tuples as D in each relation.

Theorem 3.9. *Let $J = R_1 \bowtie \ldots \bowtie R_n$ be a full join query over a database D with schema $\{R_1, \ldots, R_n\}$, let d be the number of different attributes in D, and assume the domains of all the attributes are in $[0, \ell - 1]$. Let $\mathcal{A}_i$ be the set of attributes of R_i, for all $1 \leq i \leq n$, $N = \sum_i |R_i|$ be the total number of tuples in the database, and $S = \sum_i |\mathcal{A}_i| \cdot |R_i|$ its* *total number of tuple components. The relations $R_1, \ldots, R_n$ can then be stored within $S \log \ell + 2N \log \ell + o(S \log \ell) + O(n \log d)$ bits, so that later the output for J can be computed in time $O(2^{\rho^*(J,D)} \cdot 2^d n \log \min(\ell, S)) = \tilde{O}(2^{\rho^*(J,D)})$.*

Proof. First, assume that $\log \ell$ is $O(\log S)$. The space bound can be achieved by storing the relations $R_1, \ldots, R_n$ as quadtrees $Q_1, \ldots, Q_n$, respectively. When these quadtrees are stored using the data structure of Lemma 2.1, the total space used is $\sum_i (|\mathcal{A}_i| + 2 + o(1))|R_i| \log \ell + O(n \log d) = S \log \ell + (2 + o(1))N \log \ell + O(n \log d)$ bits, which is within the claimed space bound. These quadtrees are built during the initialization of the database D, and not as part of the query evaluation algorithm. To solve the join query J we first create the qdags $(Q_1, \mathsf{Id}(\mathcal{A}_1)), \ldots, (Q_n, \mathsf{Id}(A_n))$, which takes constant time per relation. Then we use these qdags as parameters for Algorithm 4 to compute the result of the query.

We now show that Algorithm 4 runs in time within the bound of the theorem. The cost of the Extend operations is only $O(2^d n)$, according to Lemma 3.5, so the main cost of Algorithm 4 owes to the And operation. Let Q be the quadtree resulting from step 4 of Algorithm 4, let Q^+ be its non-pruned version, and let m be the maximum number of internal nodes at any level of Q^+. By Lemma 3.7 we know that the running time of this step is $O(m \cdot 2^d n \cdot \log \ell)$, which is $O(m \cdot 2^d n \cdot \log \min(\ell, S))$ since $\log \ell$ is $O(\log S)$. Furthermore, by Lemma 3.7 we know that there are relations $R_1'(A_1), \ldots, R_n'(A_n)$ with $|R_i'| \leq |R_i|$, for all $1 \leq i \leq n$, such that if the qdags $Q_1', \ldots, Q_n'$ represent $R_1'(A_1), \ldots, R_n'(A_n)$, respectively, then the quadtree $Q' = \textsc{And}\big(\textsc{Extend}(Q_1', \mathcal{A}), \ldots, \textsc{Extend}(Q_n', \mathcal{A})\big)$ has at least m integer-1 nodes. Since running Algorithm 4 over $R_1'(A_1), \ldots, R_n'(A_n)$ returns Q', the output of the query $R_1' \bowtie \ldots \bowtie R_n'$ has at least m tuples. Therefore, $m = O(2^{\rho^*(J,D)})$ and the running time of Algorithm 4 over $R_1, \ldots, R_n$ is $O(2^{\rho^*(J,D)} \cdot 2^d n \log \min(\ell, S))$.

Now, let us consider the case when $\log S$ is $o(\log \ell)$ (e.g., when the attribute values are fixed-length strings). In this case, if we proceed as before we will still meet the space bound, but the height of the quadtrees storing the relations could be $\omega(\log S)$, and then Algorithm 4 would not run in the claimed time because of Lemma 3.7. With a slight variation on how we store the relations, however, we can convert $O(\log \ell)$ to $O(\log S)$ in the time complexity of this algorithm while preserving the space bound. First we store the values of the attributes appearing in any relation in an auxiliary data structure (e.g., an array), and associate an $O(\log S)$-bits identifier to each different value in $[0, \ell - 1]$ that appears in D (e.g., the index of the corresponding value in the array). Then, we represent the relations $R_1, \ldots, R_n$ in quadtrees using the data structure of Lemma 2.1, but this time storing the identifiers of the attribute values instead of the values themselves. This representation requires at most $S \log \ell$ bits for the representation of the distinct attribute values, and $S \log S + (2 + o(1))N \log S + O(n \log d)$ bits for the quadtrees, which is $o(S \log \ell) + O(n \log d)$. The total is then within the claimed space. As already mentioned, this representation is computed during the initialization of the database D, and is not part of the query evaluation algorithm. The analysis of the running time of Algorithm 4 is the same as in the previous case, but this time when reporting the output we must map the $O(\log S)$-bits identifier of an attribute value to its original $O(\log \ell)$-bits value. This can be done by adding only a constant-time overhead per output (e.g., when the auxiliary data structure is an array, the $O(\log S)$-bits identifier is directly the position in this array where its corresponding attribute value is stored). Thus, the space and running time bound claimed also hold for this case, which completes the proof. □

In a practical implementation of Theorem 3.9, given a database D with n relations and d attributes in total, we only store the relations as quadtrees, a lookup table of size $O(2^{2d})$ to support the Extend operation efficiently (as described in the proof of Lemma 3.5), and an array or hash table to store the original values of attributes requiring more than $\omega(\log S)$ bits in their representation (e.g., string-valued attributes). Thus, the initialization of the database consists of building the n quadtrees and computing the lookup table. We assumed in Theorem 3.9 that the domain of all the attributes is the same, but one can overcome this restriction in practice by combining the two different representations described in the proof of the theorem. With respect to the output of the queries note that, although Algorithms 4 and 5 return a quadtree representing the output, they can be easily modified to just report the values found. For instance, in Algorithm 5 it is enough to remove lines 6-7, and replace the **return** statement in line 2 by one reporting the output tuple corresponding to the leaf found. In this case, the working memory used by Algorithms 4 and 5 is $O(\log \min(\ell, S))$, which is the height of the quadtrees representing the relations.

We present and describe in depth a practical implementation in Section 6, and compare its performance with state-of-the-art alternatives in Section 7.

3.4 Better space and time on clustered datasets

As noted in Section 2, storing p points in a quadtree requires at most $p \log \ell$ internal nodes, because each point stored at an integer-1 node induces a path of $\log \ell$ internal nodes leading to it. This is, however, an upper bound, because those paths are not disjoint; see in Figure 1 the paths leading to $(6, 12)$, $(7, 12)$, $(6, 13)$, and $(7, 13)$. It is not hard to show that the quadtree has indeed fewer nodes on clustered data (cf. Gagie et al. [13, Thm. 1] for two dimensions).

Lemma 3.10. *A quadtree in dimension d containing p points distributed along c clusters, with the i-th cluster containing p_i points inside a subgrid of size ${s_i}^d$, has $O(c \cdot 2^d \log \ell + \sum_i p_i \log s_i)$ internal nodes.*

Proof. It is sufficient to count the number of ancestors of the nodes in each cluster separately. Consider the i-th cluster, within a hypercube of size ${s_i}^d$. Since the quadtree nodes at depth $h_i = \lfloor \log(\ell/s_i) \rfloor$ span a subgrid of size at least s_i^d, the cluster hypercube intersects at most two such nodes in each dimension, for a total of 2^d nodes. Adding up their ancestors, it turns out that the cluster has at most $2^d \log(\ell/s_i)$ ancestors of depth h_i or less. In addition, each of the p_i points of the cluster has $p_i(\log \ell - h_i) < p_i(1 + \log s_i)$ ancestors deeper than h_i. Summing up we have the upper bound $O(c \cdot 2^d \log \ell + \sum_i p_i \log s_i)$. □

This reduction directly impacts on the space required to store quadtrees. Indeed, quadtrees have been shown to work well in applications such as RDF stores or web graphs, where data points are distributed in clusters [3, 9]. We can further show that clustering also impacts positively on query times if we represent the data using quadtrees: by combining their space analysis with the technique we used to prove Theorem 3.9, we obtain better time bounds, and a small refinement of the AGM bound itself. Those improvements will show up empirically in Section 7.

Consider again the triangle query $R(A, B) \bowtie S(B, C) \bowtie T(A, C)$, and assume the points in each relation are distributed in c clusters, each of them fitting in a square grid of width at most s and size at most $s \times s$, and with p points in total. Then, at depth $\log(\ell/s)$, the quadtrees of T, R, and S have at most $2^2 = 4$ internal nodes per cluster: at this level one can think of the trimmed quadtree as representing a coarser grid of cells of size $s \times s$, and therefore each cluster can intersect at most two of these coarser cells per dimension. Thus, letting Q'_R, Q'_S, and Q'_T be the quadtrees for R, S and T trimmed up to level $\log(\ell/s)$ (and where internal nodes take value 1), then the proof of Theorem 3.9 yields a bound for the number of internal nodes at level $\log(\ell/s)$ of the non-pruned quadtree Q^+ of the output: this number must be bounded by the AGM bound of the instances given by Q'_R, Q'_S and Q'_T, which is at most $(c \cdot 2^2)^{3/2} = 8c^{3/2}$. Going back to the data for the quadtree Q^+, the bound on the number of internal nodes means that the points of the output are distributed in at most $8c^{3/2}$ clusters of size at most s^3. In turn, the maximal number of 1s in the answer is bounded by the AGM bound itself, which here is $p^{3/2}$. This means, by Lemma 3.10, that the size of Q^+ is bounded by $O(c^{3/2} \log \ell + p^{3/2} \log s)$, and so is also the running time of the algorithm. This is an important reduction from the general bound $O(p^{3/2} \log \ell)$ if the number c of clusters and their width s are small, as we now multiply the number of answers by $\log s$ instead of $\log \ell$.

Corollary 3.11. *If, in the scenario of Theorem 3.9, the points in each relation are distributed in c clusters of width s, then the join algorithm works in time* $O\Big((2^{\rho^*(J,D^s)}\log(\ell/s) + 2^{\rho^*(J,D)}\log s)\cdot 2^d n\Big)$*, where* D^s *denotes the database with all the values trimmed to their highest* $\lfloor\log(\ell/s)\rfloor$ *bits and thus* $2^{\rho^*(J,D^s)} \leq c\cdot 2^{\sum_i |\mathcal{A}_i|} \leq c\cdot 2^{dn}$.

Proof. Assume s is a power of 2, by increasing it to the closest one if needed, and call $d_i = |\mathcal{A}_i|$. Let R_i^s be the relations resulting out of trimming the values in R_i to their highest $h = \log(\ell/s)$ bits, and D^s be the instance given by the relations $\{R_1^s,\ldots,R_n^s\}$. Every empty (resp., nonempty) subgrid of size s^{d_i} of the quadtree of R_i then becomes an integer-0 (resp., integer-1) leaf of the quadtree of R_i^s. Further, every cluster of size s^{d_i} in R_i results in at most 2^{d_i} points in R_i^s, and thus $|R_i^s| \leq c\cdot 2^{d_i}$.

Now consider the non-pruned quadtree Q^+ built during the application of Algorithm 5. The number of nodes at depth h correspond to the last-level nodes of the output of the query J applied on D^s, that is, at most $2^{\rho^*(J,D^s)}$. The total number of internal nodes in Q^+ can then be bounded by splitting them into two parts: (i) the nodes at depth h in Q^+ and their ancestors, and (ii) the deeper nodes in Q^+. While the first part is clearly $O(2^{\rho^*(J,D^s)}\log(\ell/s))$, the second part can be bounded as in Theorem 3.9: take the level of Q^+ with the maximum number m of nodes, which is at most $2^{\rho^*(J,D)}$, and multiply it by the $\log\ell - h = \log s$ levels that are accounted for in (ii). The result then follows by charging the $O(2^d n)$ cost incurred by the intersection algorithm on each node of Q^+. The last inequality follows from the fact that the n restricted relations R_i^s are of maximum cardinality $c\cdot 2^{d_i}$ and $2^{\rho^*(J,D^s)} \leq \Pi_i |R_i^s| \leq c\cdot 2^{\sum_i d_i} \leq c\cdot 2^{dn}$. □

3.4.1 *Geometric interpretation.*

As quadtrees have a direct geometric interpretation, it is natural to compare them to the algorithm based on *gap boxes* proposed by Khamis et al. [18]. In a nutshell, this algorithm uses a data structure that stores relations as a set of multidimensional cubes that contain no data points, which the authors call gap boxes. Under this framework, a data point is in the answer of the join query $R_1 \bowtie \cdots \bowtie R_n$ if the point is not part of a gap box in any of the relations R_i. The authors then compute the answers of these queries using an algorithm that finds and merges appropriate gap boxes covering all cells not in the answer of the query, until no more gap boxes can be found and we are left with a covering that misses exactly those points in the output of the query. Such an algorithm is subject of a finer analysis: the runtime of queries can be shown to be bounded by a function of the size of a *certificate* of the instance (and not its size). The certificate in their case is simply the minimum amount of gap boxes from the input relations that is needed to cover all the gaps in the answer of the query. Finding such a minimal cover is NP-hard, but a slightly restricted notion of gap boxes maintains the bounds within an $O(\log^d \ell)$ approximation factor.

While any index structure can be thought of as providing a set of gap boxes [18], quadtrees provide a particularly natural and compact representation. Each leaf or node valued 0 in a quadtree signals that there are no points in its subgrid, and can therefore be understood as a d-dimensional gap box. Now let $J = R_1 \bowtie \cdots \bowtie R_n$ be a join query over d attributes, and let R_i^* denote the extension of R_i to the attributes of J. As in Khamis et al. [18], a *quadtree certificate* for J is a set of gap boxes (i.e., empty d-dimensional grids obtained from any of the R_i^*s) such that every coordinate not in the answer of J is covered by at least one of these boxes. Let $C_{J,D}$ denote a certificate for J of minimum size; then we can reinterpret the running time of Theorem 3.9 as follows.

Corollary 3.12. *Given multijoin J on a database D, Algorithm 4 runs in time* $O((|C_{J,D}| + |J(D)|)\cdot 2^d n\log\ell)$*, where* $J(D)$ *is the output of the query J over D.*

Now, one can easily construct instances and queries such that the minimal certificate $C_{J,D}$ is comparable to $2^{\rho^*(J,D)}$. So this will not give us instance-optimality results, as discovered [18, 32] for acyclic queries or queries with bounded treewidth. This is a consequence of increasing the dimensionality of the relations. Nevertheless, the bound does yield a good running time when we know that $C_{J,D}$ is small, as evidenced in Corollary 3.11. It is also worth mentioning that our join algorithm directly computes the only possible representation of the output as gap boxes (because its boxes come directly from the representation of the relations). This means that there is a direct connection between instances that give small certificates and instances for which the representation of the output is small.

4 EXTENDING WORST-CASE OPTIMALITY TO BOOLEAN QUERIES

Next we turn to design worst-case optimal algorithms for more general queries of the relational algebra. We start in this section with the Boolean queries: we already studied the *intersection* (which corresponds to operation And over the qdags), and will show that *union* (operation Or) and *complement* (operation Not) can be solved optimally as well. What is most intriguing, however, is whether we can obtain worst-case optimality on combined relational formulas. We introduce a notion of worst-case optimality that generalizes the AGM bound on multijoin queries, and then design a worst-case optimal algorithm (in data complexity) to evaluate formulas that combine join, union, and complement operations. We refer to those formulas as *JUC queries*[5]; note that the intersection is a particular case of join.

Definition 4.1. Let F be a formula in the relational algebra over relations $R_1(\mathcal{A}_1), \dots, R_n(\mathcal{A}_n)$ from a database D. We define $F(D)^*$ as the maximum size of the output of F over instances D' with relations $R'_1, \dots, R'_n$ of respective sizes $|R'_i| = O(|R_i|)$, and their complements of sizes $|\overline{R'_i}| = O(|\overline{R_i}|)$, for all $1 \le i \le n$. An algorithm to evaluate F is said to be *worst-case optimal* if it evaluates F in time $O(F(D)^*)$, and *worst-case optimal in data complexity* if it evaluates F in time $\tilde{O}(F(D)^*)$, which is $O(F(D)^*)$ multiplied by factors that depend only on the size $|F|$ of the formula, the total number $d = |\cup_i \mathcal{A}_i|$ of attributes, and at most polylogarithms of the data and domain sizes.

The guard about the sizes of $\overline{R'_i}$ is important when R_i appears negated in F, to avoid the complement of R'_i to be very large and induce a poor bound $F(D)^*$. For instance, let $F = R_1 \cap \overline{R_2}$ and $|R_2| = \ell^d - O(1)$, so every possible output of F is of size $O(1)$. Yet, if we can choose a relation R'_2 with $|R'_2| = |R_2|/2 = O(|R_2|)$, then the output of F on R'_2 can be of size up to $\ell^d/2 + O(1)$, and hence $F(D)^* = \Theta(\ell^d)$. Our definition avoids this by enforcing that $|\overline{R'_2}| = O(|\overline{R_2}|) = O(1)$. We do not need to enforce this condition on the relations R_i that do not appear negated in F.

In the particular case where F is a multijoin formula J (no negations), we have $F(D)^* = 2^{\rho^*(J,D)}$, and we have achieved the corresponding worst-case optimality (in data complexity) in Theorem 3.9. The key technique to obtain worst-case optimality on more complex queries is to deal with them in a *lazy* form, allowing unknown intermediate results so that all the components of a formula are evaluated simultaneously. To do this we introduce lazy qdags (or lqdags), an alternative to qdags that can navigate over the quadtree representing the output of a formula without the need to entirely evaluate the formula. We then give a worst-case optimal algorithm to compute the *completion* of an lqdag, that is, the quadtree of the grid represented by the lqdag.

4.1 Lqdags for relational formulas

To support worst-case optimal evaluation of relational formulas we introduce two new ideas: we add "full leaves" to the quadtree representation to denote subgrids full of 1s, and we introduce lqdags to represent the result of a formula as an *implicit* quadtree that can be navigated without fully evaluating the formula.

While the last-level nodes of quadtrees represent a single cell and store its value, 0 or 1, quadtree leaves at higher levels always represent subgrids full of 0s. We now generalize the representation, so that those quadtree leaves at higher levels also store an integer, 0 or 1, which is the value of all the cells in the subgrid represented by

[5]Strictly speaking, JUC queries are incomparable to relational algebra because complement operations can only be simulated with difference when the domain of the relations coincides with the active domain. However, to streamline the comparison between boolean queries and relational algebra, we assume that the domain and active domain of relations coincide, and therefore complement and difference are interchangeable.

Algorithm 6 Value on extended qdags

Require: A qdag (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
```

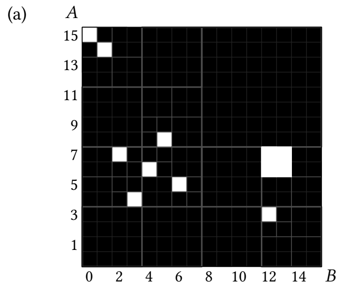
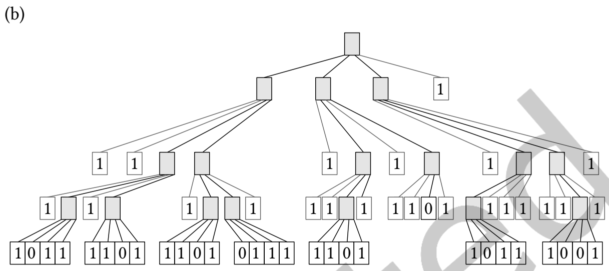



Fig. 5. (a) A grid representing the complement $\overline{R(A,B)}$ of the relation $R(A,B)$ in Fig 1. (b) The quadtree representing $\overline{R}$. The integers can now appear at any level, and they represent either empty or full subgrids. The internal nodes are grayed.

the leaf. This generalization is introduced in order to complement empty subgrids in constant time; see Figure 5. It is now more convenient to call *leaves* all the quadtree nodes, in the last level or higher, whose subgrids are all 0s or all 1s, and thus storing the corresponding integer (see Figure 5). The generalization impacts on the way to compute Value, as depicted in Algorithm 6. We will not use qdags in this section, however; the lqdags build directly on quadtrees.

In terms of the compact representation, this generalization is implemented by resorting to an impossible quadtree configuration: a string shorter than $\log \ell$ in the trie structure of Lemma 2.1 will be used to denote the path of a quadtree node ending at a leaf full of 1s. We emphasize that we are allowed to use leaves with value 1 when possible, but not forced: quadtrees with those nodes partially or fully expanded are also valid and *equivalent*, in the sense that they represent the same set of points. A relation can then be represented by different quadtrees.

The second novelty, the lqdags, are defined as follows.

Definition 4.2 (lqdag). An lqdag $\mathcal{L}$ is a pair (f, o), where f is a *functor* and o is a list of *operands*, recursively built using the following rules. The rules also define the *completion* of $\mathcal{L}$, which is a quadtree Q_R representing a relation $R(\mathcal{A})$; we also say that $\mathcal{L}$ represents R.

(1) $\mathcal{L} = (\mathsf{QTREE}, Q_R)$.
(2) $\mathcal{L} = (\mathsf{NOT}, Q_{\overline{R}})$, where $Q_{\overline{R}}$ is a quadtree representing the complement of R;
(3) $\mathcal{L} = (\mathsf{AND}, \mathcal{L}_1, \mathcal{L}_2)$, where $\mathcal{L}_1$ and $\mathcal{L}_2$ are lqdags, and Q_R represents the intersection of their completions;
(4) $\mathcal{L} = (\mathsf{OR}, \mathcal{L}_1, \mathcal{L}_2)$, where $\mathcal{L}_1$ and $\mathcal{L}_2$ are lqdags, and Q_R represents the union of their completions;
(5) $\mathcal{L} = (\mathsf{EXTEND}, \mathcal{L}_1, \mathcal{A})$, where $\mathcal{L}_1$ is an lqdag representing a relation $R'(\mathcal{A}')$, $\mathcal{A}$ is a set of attributes such that $\mathcal{A}' \subseteq \mathcal{A}$, and $R(\mathcal{A}) = R'(\mathcal{A}') \times \mathrm{All}(\mathcal{A} \setminus \mathcal{A}')$.

To illustrate the definition of lqdags, consider the triangle query $R(A,B) \bowtie S(B,C) \bowtie T(A,C)$, with $\mathcal{A} = \{A, B, C\}$ and the relations represented by quadtrees Q_R, Q_S, and Q_T. This query can then be represented as the lqdag

$$(\mathsf{AND}, (\mathsf{AND}, (\mathsf{EXTEND}, (\mathsf{QTREE}, Q_R), \mathcal{A}), (\mathsf{EXTEND}, (\mathsf{QTREE}, Q_S), \mathcal{A})), (\mathsf{EXTEND}, (\mathsf{QTREE}, Q_T), \mathcal{A})).$$

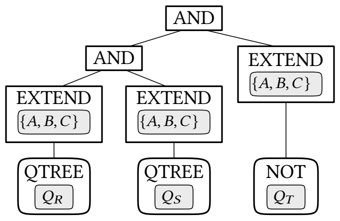


Fig. 6. Illustration of the syntax tree of an lqdag for the formula $(R(A, B) \bowtie S(B, C)) \bowtie \overline{T}(A, C)$. The quadtrees Q_R, Q_S, Q_T represent the relations R, S, T, respectively.

Lqdags as syntax trees. An lqdag $\mathcal{L} = (f, o)$ can be interpreted as a *syntax tree* of the formula F that it represents. The leaves of this tree correspond to the lqdags with functors QTREE and NOT (i.e., those having a quadtree as their operand) present in $\mathcal{L}$, while its internal nodes correspond to lqdags with functors OR, AND, and EXTEND (i.e., those having lqdags as their operands). The root of the syntax tree for $\mathcal{L}$ is a node corresponding to the functor f, and there is an edge from a node for an lqdag $\mathcal{L}_1$ to a node for an lqdag $\mathcal{L}_2$ if and only if $\mathcal{L}_2$ is an operand of $\mathcal{L}_1$ (see Figure 6 for an illustration). The lqdags corresponding to each node of this syntax tree are called *fnodes* of $\mathcal{L}$.

Deriving other functors and the limitations of the NOT *operand.* For a quadtree Q_R representing a relation $R(\mathcal{A}')$, and a set of attributes $\mathcal{A}$ such that $\mathcal{A}' \subseteq \mathcal{A}$, the qdag $(Q_R, M_{\mathcal{A}})$ that represents the relation $R(\mathcal{A}') \times \mathrm{All}(\mathcal{A} \setminus \mathcal{A}')$ can be expressed as the lqdag (EXTEND, (QTREE, Q_R), $\mathcal{A}$). In this sense, lqdags are extensions of qdags. Note also that JUC queries, and even other more general formulas, can be expressed as lqdags. While UNION and COMPLEMENT are equivalent to OR and NOT, respectively, one can define other operations, like JOIN and DIFF, by composing the operations introduced in Definition 4.2:

$$(\texttt{JOIN}, \mathcal{L}_1(\mathcal{A}_1), \mathcal{L}_2(\mathcal{A}_2)) = (\texttt{AND}, (\texttt{EXTEND}, \mathcal{L}_1, \mathcal{A}_1 \cup \mathcal{A}_2), (\texttt{EXTEND}, \mathcal{L}_2, \mathcal{A}_1 \cup \mathcal{A}_2)) \quad (1)$$

$$(\texttt{DIFF}, \mathcal{L}_1(\mathcal{A}), \mathcal{L}_2(\mathcal{A})) = (\texttt{AND}, \mathcal{L}_1, (\texttt{NOT}, \mathcal{L}_2)) \quad (2)$$

Note that in the definition of the lqdag for NOT, the operand is a quadtree instead of an lqdag, and then, for example, $\mathcal{L}_2$ should be a quadtree in the definition of DIFF in Eq. (2), in principle. We can get around that restriction by pushing down the NOT operators until the operand is a quadtree or the NOT is cancelled with another NOT. We formalize this process in Lemma 4.3. Proceeding in this way, however, does limit the types of formulas for which we achieve worst-case optimality, as shown later in Section 4.2.

LEMMA 4.3. *Let $\mathcal{L} = (f, o)$ be an lqdag, and let k be the total number of functors that appear in $\mathcal{L}$, including f and all those present recursively in o. Let $\texttt{NOT}^*(\mathcal{L})$ denote the quadtree representing the complement of the completion of $\mathcal{L}$. Then there is an lqdag $\overline{\mathcal{L}} = (f', o')$ whose completion is $\texttt{NOT}^*(\mathcal{L})$, and contains k functors in total (i.e., including f' and those present recursively in o').*

PROOF. We prove the lemma by induction on k. When $k = 1$, $\mathcal{L}$ must be either (QTREE, o) or (NOT, o), with o being a quadtree. In this case $\overline{\mathcal{L}'}$ = (NOT, o), or $\overline{\mathcal{L}'}$ = (QTREE, o), respectively.

When $k > 1$, f must be either AND, or OR, or EXTEND. For $\mathcal{L}$ = (AND, $\mathcal{L}_1, \mathcal{L}_2$), the number of functors k_1, k_2 present in $\mathcal{L}_1, \mathcal{L}_2$, respectively, must be less than k. Thus, by induction there are lqdags $\overline{\mathcal{L}_1}, \overline{\mathcal{L}_2}$, containing exactly k_1, k_2 functors, respectively, and whose completions are $\texttt{NOT}^*(\mathcal{L}_1)$, $\texttt{NOT}^*(\mathcal{L}_2)$. By the de Morgan law, the completion of the lqdag $\overline{\mathcal{L}}$ = (OR, $\overline{\mathcal{L}_1}, \overline{\mathcal{L}_2}$) is $\texttt{NOT}^*(\mathcal{L})$. Moreover it contains exactly $1 + k_1 + k_2 = k$ functors in it, and thus the statement is true for this case. Analogously, when $\mathcal{L}$ = (OR, $\mathcal{L}_1, \mathcal{L}_2$), $\overline{\mathcal{L}}$ = (AND, $\overline{\mathcal{L}_1}, \overline{\mathcal{L}_2}$) contains k functors and has $\texttt{NOT}^*(\mathcal{L})$ as its completion.

We are left with the case when $\mathcal{L} = (\texttt{EXTEND}, \mathcal{L}_1, \mathcal{A})$. Since $\mathcal{L}_1$ must contain $k-1$ functors, there is by induction an lqdag $\overline{\mathcal{L}_1}$ containing exactly k_1 functors and whose completion is $\texttt{NOT}^*(\mathcal{L}_1)$. We can show that the completion of $\overline{\mathcal{L}} = (\texttt{EXTEND}, \overline{\mathcal{L}_1}, \mathcal{A})$ is precisely $\texttt{NOT}^*(\mathcal{L})$. By Definition 4.2, the completion of $\mathcal{L}_1$ must represent a relation $R'(\mathcal{A}')$ with $\mathcal{A}' \subseteq \mathcal{A}$, while the completion of $\mathcal{L}$ must represent a relation $R(\mathcal{A}) = R'(\mathcal{A}') \times \text{All}(\mathcal{A} \setminus \mathcal{A}')$. Let $\overline{R(\mathcal{A})}, \overline{R'(\mathcal{A}')}$ denote the complements of $R(\mathcal{A}), R'(\mathcal{A}')$, respectively. Let t_X denote the projection of a tuple t to a set of attributes X. Note that $\overline{R(\mathcal{A})}$ can be written as

$$\overline{R(\mathcal{A})} = \{t \in \text{All}(\mathcal{A}) \mid t_{\mathcal{A}'} \notin R'(\mathcal{A}') \text{ or } t_{\mathcal{A}\setminus\mathcal{A}'} \notin \text{All}(\mathcal{A} \setminus \mathcal{A}')\}.$$

Since the complement of $\text{All}(\mathcal{A} \setminus \mathcal{A}')$ is empty, $\overline{R(\mathcal{A})} = \{t \in \text{All}(\mathcal{A}) \mid t_{\mathcal{A}'} \in \overline{R'(\mathcal{A}')}\}$, which is the same as $\overline{R(\mathcal{A})} = \overline{R'(\mathcal{A}')} \times \text{All}(\mathcal{A} \setminus \mathcal{A}')$. By Definition 4.2, this is the relation represented by the completion of the lqdag $\overline{\mathcal{L}} = (\texttt{EXTEND}, \overline{\mathcal{L}_1}, \mathcal{A})$, thus completing the proof. ☐

Laziness of lqdags. To understand why we call lqdags lazy, consider the operation Q_1 AND Q_2 over quadtrees Q_1, Q_2. If either VALUE at the roots of Q_1 or Q_2 is 0, then the result of the operation is for sure a leaf with VALUE 0. If either VALUE is 1, then the result of the operation is the other quadtree. However, if both roots have VALUE ½, one cannot be sure of the VALUE of the resulting root until the AND between the children of Q_1 and Q_2 has been computed. Solving this dependency eagerly would go against worst-case optimality: it forces us to fully evaluate parts of the formula without considering it as a whole. To avoid this, we allow the VALUE of a node represented by an lqdag to be, apart from 0, 1, and ½, the special value $\diamond$. This indicates that one cannot determine the value of the node without computing the values of its children.

As we did for qdags, in order to simulate the navigation over the completion Q of an lqdag $\mathcal{L}$ we need to describe how to obtain the value of the root of Q, and how to obtain an lqdag whose completion is the i-th child of Q, for any given i. We implement those operations in Algorithms 7 and 8, both constant-time. Note that CHILD can only be invoked when VALUE = ½ or $\diamond$. The base case occurs when $\mathcal{L} = (\texttt{QTREE}, Q)$ or $\mathcal{L} = (\texttt{NOT}, Q)$, where we enter the quadtree and resort to the algorithms based on the compact representation of Q. As per Algorithm 6, VALUE(Q) returns ½ for internal nodes, and thus the implementation of VALUE for EXTEND is trivial in Algorithm 7.

Note that the recursive calls of Algorithms 7 and 8 traverse the fnodes of the lqdag, and terminate immediately upon reaching an fnode of the form $(\texttt{QTREE}, Q)$ or $(\texttt{NOT}, Q)$. Therefore, their time complexity depends only on the size of the formula represented by the lqdag. We show next how, using these implementations of VALUE and CHILD, one can efficiently evaluate a relational formula using lqdags.

4.2 Evaluating JUC queries

To evaluate a formula F represented as an lqdag $\mathcal{L}_F$, we compute the completion Q_F of $\mathcal{L}_F$, that is, the quadtree Q_F representing the output of F (as detailed in Algorithm 9).

To implement this we introduce the idea of *non-pruned completion* of an lqdag. The non-pruned completion Q_F^+ of $\mathcal{L}_F$ is the quadtree induced by navigating $\mathcal{L}_F$, and interpreting the values $\diamond$ as ½ (as in Algorithm 9 omitting lines 4 and 5). Note that, by interpreting values $\diamond$ as ½, we are disregarding the possibility of pruning resulting subgrids full of 0s or 1s and replacing them by single leaves with values 0 or 1 in Q_F. Therefore, Q_F^+ is a *non-pruned* quadtree (just as Q^+ in Section 3.3) that nevertheless represents the same points of Q_F. Moreover, Q_F^+ shares with Q_F a key property: all its nodes with value 1, including the last-level leaves representing individual cells, correspond to actual tuples in the output of F.

To see how lqdags are evaluated, let us consider the query $F = R(A, B) \bowtie S(B, C) \bowtie \overline{T}(A, C)$. This corresponds to an lqdag $\mathcal{L}_F$:

$$(\texttt{AND}, (\texttt{AND}, (\texttt{EXTEND}, (\texttt{QTREE}, Q_R), \mathcal{A}), (\texttt{EXTEND}, (\texttt{QTREE}, Q_S), \mathcal{A})), (\texttt{EXTEND}, (\texttt{NOT}, Q_T), \mathcal{A})).$$

Algorithm 7 Value ($\mathcal{L}$)

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

Algorithm 8 Child ($\mathcal{L}, i$)

```
Require: An lqdag $\mathcal{L}(\mathcal{A})$ and an integer $0 \le i < 2^{|\mathcal{A}|}$.
Ensure: An lqdag for the $i$-th child of $\mathcal{L}$.
1: if $\mathcal{L}$ = (QTREE, $Q$) then
2:     return (QTREE, Child($Q$, $i$))
3: if $\mathcal{L}$ = (NOT, $Q$) then
4:     return (NOT, Child($Q$, $i$))
5: if $\mathcal{L}$ = (AND, $\mathcal{L}_1$, $\mathcal{L}_2$) then
6:     if Value($\mathcal{L}_1$) = 1 then return Child($\mathcal{L}_2$, $i$)
7:     if Value($\mathcal{L}_2$) = 1 then return Child($\mathcal{L}_1$, $i$)
8:     return (AND, Child($\mathcal{L}_1$, $i$), Child($\mathcal{L}_2$, $i$))
9: if $\mathcal{L}$ = (OR, $\mathcal{L}_1$, $\mathcal{L}_2$) then
10:     if Value($\mathcal{L}_1$) = 0 then return Child($\mathcal{L}_2$, $i$)
11:     if Value($\mathcal{L}_2$) = 0 then return Child($\mathcal{L}_1$, $i$)
12:     return (OR, Child($\mathcal{L}_1$, $i$), Child($\mathcal{L}_2$, $i$))
13: if $\mathcal{L}$ = (EXTEND, $\mathcal{L}_1(\mathcal{A}')$, $\mathcal{A}$) then
14:     $d \leftarrow |\mathcal{A}|$, $d' \leftarrow |\mathcal{A}'|$
15:     $m_d$ ← the $d$-bits binary representation of $i$
16:     $m_{d'}$ ← the projection of $m_d$ to the positions in which the
        attributes of $\mathcal{A}'$ appear in $\mathcal{A}$
17:     $i'$ ← the value in $[0, 2^{d'} - 1]$ corresponding to $m_{d'}$
18:     return (EXTEND, Child($\mathcal{L}_1$, $i'$), $\mathcal{A}$)
```

Assuming that some of the quadtrees involved in $\mathcal{L}_F$ have internal nodes, the non-pruned completion Q_F^+ first produces 8 children. Suppose the grid of T is full of 1s in the first quadrant (00). Then the first child (00) of Q_T has value 1, which becomes value 0 in (NOT, Q_T). This implies that (EXTEND, (NOT, Q_T)) also yields value 0 in octants 000 and 010. Thus, when function Child is called on child 000 of Q_F, our 0 is immediately propagated and Child returns 0, meaning that there are no answers for F on this octant, without ever consulting the quadtrees Q_R and Q_S (see Figure 7 for an illustration). On the other hand, if the value of the child 11 of T is 0, then (EXTEND, (NOT, Q_T)) will return value 1 in octants 101 and 111. This means that the result on this octant corresponds to the result of joining R and S; indeed Child towards 101 in Q_F returns

$$(\text{AND}, \textsc{Child}((\text{EXTEND}, (\text{QTREE}, Q_R), \mathcal{A}), 101), \textsc{Child}((\text{EXTEND}, (\text{QTREE}, Q_S), \mathcal{A}), 101)).$$

If Child((EXTEND, (QTREE, Q_R), $\mathcal{A}$), 101) and Child((EXTEND, (QTREE, Q_S), $\mathcal{A}$), 101) are trees with internal nodes, the resulting AND can be either an internal node or a leaf with value 0 (if the intersection is empty), though not a leaf with value 1. Thus, for now, the Value of this node is unknown, a ◇. See Figure 7 for an illustration.

Note that the running time of Algorithm 9 is $O(|Q_F^+|)$. Lines 4 and 5 compact Q_F^+ to obtain Q_F, without affecting the complexity of traversing Q_F^+. Thus, bounding $|Q_F^+|$ yields a bound for the running time of evaluating F. While $|Q_F^+|$ can be considerably larger than the actual size $|Q_F|$ of the output, we show that $|Q_F^+|$ is bounded by the worst-case output size of formula F for a database with relations of approximately the same size.

We will not use the ◇ values in the proof, because these are not needed when all we want is to materialize the output of the query; we can just replace them all by ½ in that case. The ◇ values are useful, instead, to evaluate the formula progressively, for example to provide an iterator over the output.

4.2.1 Analysis of the algorithm. Let $\mathcal{L}_F$ be an lqdag for a formula F, and consider the syntax tree corresponding to $\mathcal{L}_F$. We call *atomic expressions of* $\mathcal{L}_F$ the lqdags associated with the leaves of this tree (i.e., the fnodes with

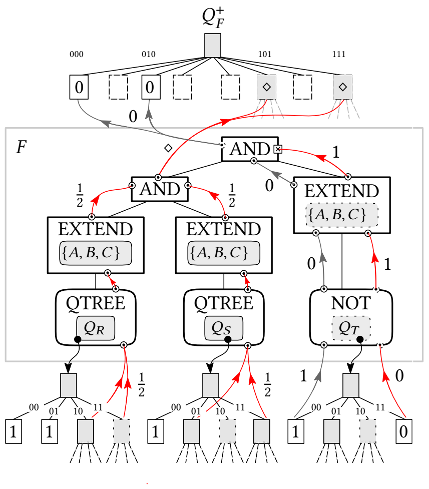


Fig. 7. Illustration of the evaluation of an lqdag for the formula $(R(A,B) \bowtie S(B,C)) \bowtie \overline{T}(A,C)$. The quadtrees Q_R, Q_S, Q_T represent the relations R, S, T, respectively. We show the top values of Q_F^+ on top and of Q_T on the bottom. The gray upward arrows show how the value 1 in the quadrant 00 of Q_T becomes 0s in octants 000 and 010 of Q_F^+ without accessing Q_R or Q_S. The red upward arrows show how the value 0 in the quadrant 11 of Q_T makes the quadrants 101 and 111 of Q_F^+ depend only on their left child (and, assuming their value is ½, becomes a ◇ in Q_F^+).

Algorithm 9 COMPLETION ($\mathcal{L}_F$)

Require: An lqdag $\mathcal{L}_F$ whose completion represents a formula F over relations with d attributes.
Ensure: The completion Q_F of $\mathcal{L}_F$.

```
1: if VALUE($\mathcal{L}_F$) ∈ {0, 1} then return a leaf with value VALUE($\mathcal{L}_F$)
2: for $i \leftarrow 0, \ldots, 2^d - 1$ do
3:     $C_i \leftarrow$ COMPLETION(CHILD($\mathcal{L}_F, i$))
4: if max{VALUE($C_0$), ..., VALUE($C_{2^d-1}$)} = 0 then return a leaf with value 0
5: if min{VALUE($C_0$), ..., VALUE($C_{2^d-1}$)} = 1 then return a leaf with value 1
6: return a quadtree with value ½ and children $C_0, \ldots, C_{2^d-1}$
```

functors QTREE and NOT, see Figure 6 again). We say that two atomic expressions $\mathcal{L}_1$ and $\mathcal{L}_2$ are equal if both their functors and operands are equal. For example, in the formula

$$\mathcal{F} = (\texttt{OR}, (\texttt{AND}, (\texttt{QTREE}, Q_R), (\texttt{QTREE}, Q_S)), (\texttt{AND}, (\texttt{QTREE}, Q_R), (\texttt{QTREE}, Q_T)))$$

there are three different atomic expressions, $(\texttt{QTREE}, Q_R)$, $(\texttt{QTREE}, Q_S)$, and $(\texttt{QTREE}, Q_T)$, while in $\mathcal{F}' = (\texttt{AND}, (\texttt{QTREE}, Q_R), (\texttt{NOT}, Q_R))$ there are two atomic expressions. Notice that in formulas like $\mathcal{F}'$, where a relation appears both negated and not negated, the two occurrences are seen as different atomic expressions. We return later to the consequences of this definition.

The following lemma is key to bound the running time of Algorithm 9 for evaluating a formula.

LEMMA 4.4. *Let F be a JUC query represented by an lqdag $\mathcal{L}_F$ in dimension d, and let Q_F^+ be the non-pruned completion of $\mathcal{L}_F$. Let $R_1(\mathcal{A}_1), \ldots, R_n(\mathcal{A}_n)$ be the relations that appear in F, and assume F does not contain subexpressions* $(\texttt{QTREE}, Q)$ *and* $(\texttt{NOT}, Q)$ *for a same quadtree Q. Let m be the maximum number of internal nodes in any level of Q_F^+. Then, there is a database D' with relations $R'_1(\mathcal{A}_1), \ldots, R'_n(\mathcal{A}_n)$ of sizes $|R'_i| = |R_i|$ for all $1 \le i \le n$, such that the output of F evaluated over D' has size $\Omega(m)$.*

PROOF. Let m_k be the number of internal nodes in level k of Q_F^+ and j be a level where $m = m_j$ is maximum. Let $A_1, \ldots, A_n$ be the distinct atomic expressions of $\mathcal{L}_F$ and $Q_1, \ldots, Q_n$ their corresponding quadtrees (note that this is the number of relations intervening in F, because both subexpressions $(\texttt{QTREE}, Q)$ and $(\texttt{NOT}, Q)$ cannot appear for the same Q). Let $Q'_1, \ldots, Q'_n$ be the quadtrees that result from trimming the levels at depths higher than $j-1$ from $Q_1, \ldots, Q_n$, respectively, and replacing each node with value ½ in level j by a leaf valued 0 or 1 depending on whether the functor of A_i is NOT or QTREE, respectively. We first show that $Q'_1, \ldots, Q'_n$ represent relations $R'_1, \ldots, R'_n$ with $|R'_i| \leq |R_i|$ such that, when F is evaluated over these relations, the output of the formula is at least m. Then we complete the proof by showing how to augment R'_i so that $|R'_i| = |R_i|$, while preserving an output size of at least m when F is evaluated over the new relations.

From internal nodes to output tuples. Let $Q_F^{+\prime}$ be the non-pruned completion of $\mathcal{L}_F$ when evaluated over $Q'_1, \ldots, Q'_n$. We will show that $Q_F^{+\prime}$ has at least m nodes with value 1 at the last level (the j-th), and thus the output of F over the relations represented by $Q'_1, \ldots, Q'_n$ is at least m.

First, consider the completion A_i^* of A_i evaluated over Q_i, and the completion $A_i^{*\prime}$ of A_i evaluated over Q'_i, for all $1 \leq i \leq n$. We say that $A_i^* \leq A_i^{*\prime}$ if both have the same topology up to the last level of $A_i^{*\prime}$, and the value of each node in $A_i^{*\prime}$ is not smaller than that of its homologous node in A_i^*.

The value of a node in the quadtree resulting from an AND or OR operand is *monotonic* on the values of the homologous nodes of the operands (i.e., when a node value in an operand is increased, the value in the homologous node of the result never decreases). A simple inductive argument on the syntax tree of $\mathcal{L}_F$ shows that the value of each node in Q_F^+ is also monotonic on the values of the homologous nodes from the quadtrees A_i^* involved in its computation. Thus, if we show that $A_i^* \leq A_i^{*\prime}$ for all i, we prove that $Q_F^+ \leq Q_F^{+\prime}$. This implies that $Q_F^{+\prime}$ has at least m nodes with value 1 in its last level: in Q_F^+ there are at least m internal nodes at level j that become leaves in the last level of $Q_F^{+\prime}$, and since their values cannot be 0 due to monotonicity, they must be 1.

If A_i is of the form $(\texttt{QTREE}, Q_i)$, showing that $A_i^* \leq A_i^{*\prime}$ is easy: the values in the first $j-1$ levels of A_i^* are respectively equal to those in the $j-1$ levels of $A_i^{*\prime}$, and the only change in the j-th level is that the values ½ were increased to 1 when converting every Q_i to Q'_i.

When $A_i = (\texttt{NOT}, Q_i)$, we show instead that $A_i^{*\prime}$ is equivalent to a quadtree $\overline{Q'_i}$ (i.e., both represent the negation of Q'_i) such that $A_i^* \leq \overline{Q'_i}$. When we convert the nodes at level j with value ½ in Q_i to leaves with value 0 in Q'_i, we may create nodes v in Q'_i whose descendant leaves are all of value 0, and thus those nodes v would become leaves with value 0 themselves. This changes the topology of $A_i^{*\prime}$ with respect to A_i^*, and it is the reason why we instead compare A_i^* with a quadtree $\overline{Q'_i}$ equivalent to $A_i^{*\prime}$. We construct $\overline{Q'_i}$ by using the same topology of Q'_i, and complementing the values 0 and 1 of every leaf (see Figure 8). In addition, we expand up to level j all those new leaves v of Q'_i that were created via pruning (which is valid because the leaves we expand in $\overline{Q'_i}$ are of value 1). After this expansion, $\overline{Q'_i}$ has the same topology of A_i^* up to level j, its node values are never smaller than those of the corresponding nodes in A_i^*, and thus $A_i^* \leq \overline{Q'_i}$.

So far, we showed how to obtain n relations over the same set of attributes as the original ones such that, when F is evaluated over them, the output size is $\Omega(m)$. Moreover, since each quadtree Q'_i is obtained by trimming Q_i at some level, we know these quadtrees represent relations R'_i such that $|R'_i| \leq |R_i|$. The values in $R'_1, \ldots, R'_n$ belong, however, to a smaller universe of j-bit values. This is remedied by simply appending $(\log \ell - j)$ 0s at the beginning of the binary representation of these values. Restoring the size of the universe also gives space to augment the generated relations so that their cardinalities coincide with the original ones. We show next how to achieve this.

Augmenting the relations. After appending the $(\log \ell - j)$ 0s at the beginning of the binary representation of each value in R'_i, the points represented by each Q'_i are distributed in a subgrid of size 2^{dj} from the total space of size $2^{d \log \ell} = \ell^d$. There are $2^{d \log \ell}/2^{dj} = 2^{dc}$ such subgrids in the domain of R_i, for $c = \log \ell - j > 0$. Thus, we have $2^{dc} - 1 > 0$ empty subgrids where we can add spurious points in order to increase the size of R'_i. We show that the amount of points within these subgrids is enough to augment R'_i so that $|R'_i| = |R_i|$.

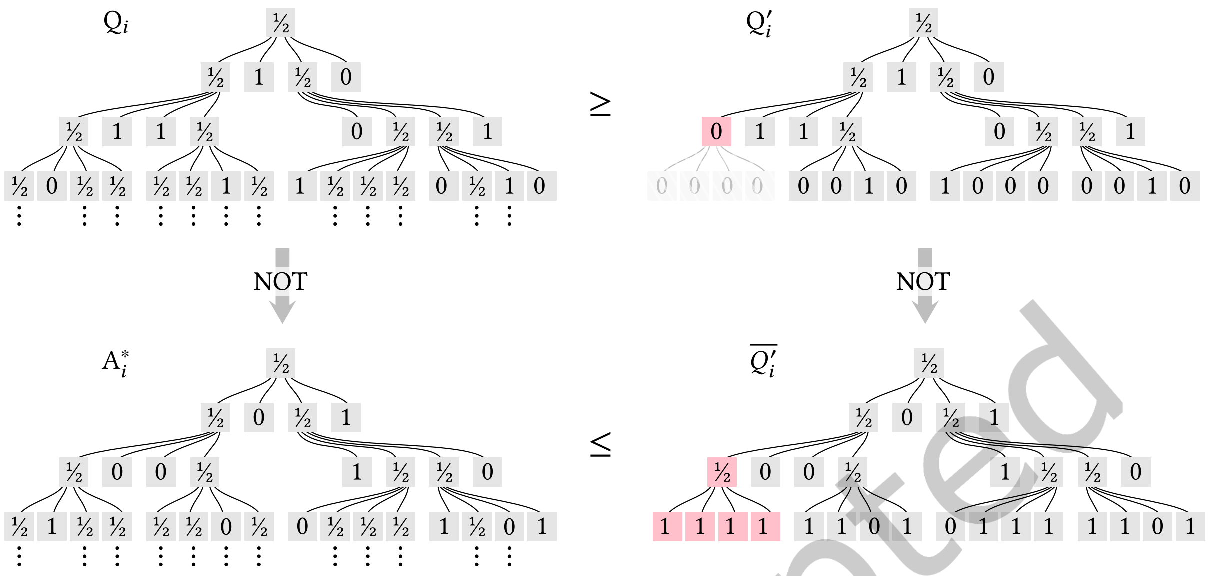


Fig. 8. An illustration of the construction of Q'_i and $\overline{Q'_i}$ in the proof of Lemma 4.4 for an atomic expression $A_i = (\texttt{NOT}, Q_i)$. The number inside each node represents its value, and the relations ≥ and ≤ drawn between trees apply to the value of each pair of corresponding nodes. The figure assumes that the maximum number of internal nodes in Q_i occurs at the fourth level. The red node in Q_i has been compacted into a leaf with value 0 since their children had all value 0 after the transformation of values ½ to 0. These nodes are un-compacted back in $\overline{Q'_i}$ to ensure Q_i and $\overline{Q'_i}$ have the same topology up to level four.

We first consider the case when $A_i = (\texttt{NOT}, Q_i)$. For the purpose of the proof, assume Q_i has leaves with value 1 whenever possible.[6] The points of R_i can be classified into the points O_i represented by a node with value 1 at level $k \leq j$ in Q_i, and the points I_i represented by a node that descends from a ½-valued node at level j of Q_i. Thus $|R_i| = |O_i| + |I_i|$, and:

- Each node valued 1 at level $k \leq j$ in Q_i induces $2^{d(j-k)}$ points in R'_i and $2^{d(\log \ell - k)}$ points in R_i; therefore there are 2^{dc} points in O_i per point in R'_i. Those are all the points in R'_i, so to match $|O_i|$ we must insert $2^{dc} - 1$ further points for each point in R'_i.
- For every empty cell in R'_i there can be up to $2^{dc} - 1$ points in I_i, because if the 2^{dc} points existed in the subgrid of R_i corresponding to the cell of R'_i, then they would have induced a 1 at level j of Q_i, not a ½. Therefore, to match $|I_i|$ we must insert up to $2^{dc} - 1$ further points for each empty cell in R'_i.

Summing up both cases, it suffices to add $2^{dc} - 1$ points for each of the 2^{dj} cells of R'_i, full or empty. Then, we have sufficient space in all the other $2^{dc} - 1 > 0$ subgrids, each of 2^{dj} cells, to add the spurious points needed to make R'_i of size $|R_i|$.

The case of $A_i = (\texttt{QTREE}, Q_i)$ follows easily from the previous case. Let R''_i be a relation represented by the quadtree Q''_i generated via trimming at the j-th level for the atomic expression $(\texttt{NOT}, Q_i)$. Then, $|R'_i| = |R''_i| + m$ because in Q'_i the m internal nodes in the j-th level are converted into nodes with value 1, while in Q''_i they become leaves with value 0. We already know that there is enough room in the new subgrids to augment R''_i so that $|R_i| = |R''_i|$, and therefore the same can be achieved for R'_i, which requires fewer points. □

[6]Alternatively, when generating Q'_i, we convert to 1, not to 0, the values ½ whose descendant leaves are all 1s.

For a formula F represented as a d-dimensional lqdag $\mathcal{L}_F$ that involves relations $R_1, \ldots, R_n$, we can bound the time needed to compute the non-pruned completion Q_F^+ of $\mathcal{L}_F$ using the same reasoning as in Section 3.3. Since m is the maximum number of internal nodes in a level of Q_F^+, the number of internal nodes in Q_F^+ is at most $m \log \ell$. Now, each internal node in $Q_{\mathcal{L}_F}^+$ results from the application of $|F|$ operations on each of the 2^d children being generated, all of which take constant time. Thus the non-pruned completion can be computed in time $O(m \cdot 2^d |F| \log \ell)$. On the other hand, by Definition 4.1, it holds that $F(D)^* = \Omega(m)$, and therefore the query F can be computed in time $O(F(D)^* \cdot 2^d |F| \log \ell)$. This means that the algorithm is worst-case optimal in data complexity. By using the same technique of Theorem 3.9 to convert $\log \ell$ into $\log \min(\ell, S)$, we obtain the result.

Theorem 4.5. *Let F be a JUC query on the relations $\{R_1, \ldots, R_n\}$ of a database D, with d attributes in total, and where the domains of the relations are in $[0, \ell - 1]$. Let $\mathcal{A}_i$ be the set of attributes of R_i, for all $1 \leq i \leq n$, $N = \sum_i |R_i|$ be the total number of tuples in the database, and $S = \sum_i |\mathcal{A}_i| \cdot |R_i|$ its total number of entries. The relations $R_1, \ldots, R_n$ can then be stored within $S \log \ell + 2N \log \ell + o(S \log \ell) + O(n \log d)$ bits so that, if the lqdag of F does not contain both subexpressions $(\texttt{QTREE}, Q)$ and $(\texttt{NOT}, Q)$ for some Q, its output can be computed in time $O(F(D)^* \cdot 2^d |F| \log \min(\ell, S)) = \tilde{O}(F(D)^*)$.*

This result generalizes Theorem 3.9, where $F(D)^* = 2^{\rho^*(F,D)}$ and $|F| = \Omega(n)$. Moreover, it does not matter how we write our formula F to achieve worst-case optimal evaluation. For example, our algorithms behave identically on $((R \bowtie S) \bowtie T)$ and on $(R \bowtie (S \bowtie T))$.

Lemma 4.4 requires that no single quadtree Q_i appears in both forms, $(\texttt{QTREE}, Q_i)$ and $(\texttt{NOT}, Q_i)$, in the formula, because for each of those atoms we create different versions of Q_i' to construct the worst-case database D'. This restriction carries over to Theorem 4.5. To see why it is necessary, consider again our example formula $\mathcal{F}' = (\texttt{AND}, (\texttt{QTREE}, Q_R), (\texttt{NOT}, Q_R))$. While the answer of this query is always empty, and therefore $|Q_{\mathcal{F}'}| = 0$, it holds $|Q_{\mathcal{F}'}^+| = \Omega(|R|/2^d)$ for every R, and our algorithm will take time $\Omega(|R|)$. Our algorithm is then worst-case optimal only if we consider the possible output size of a more general formula, $\mathcal{F}'' = (\texttt{AND}, (\texttt{QTREE}, Q_R), (\texttt{NOT}, Q_R'))$. This impacts in other operations of the relational algebra: we can write them all as lqdags, but for some of them we will not ensure their optimal evaluation.

In case the formula $\mathcal{F}$ contains (after pushing down the NOT operators) both atomic subexpressions $(\texttt{QTREE}, Q)$ and $(\texttt{NOT}, Q)$ for the same Q, Theorem 4.5 can still yield an upper bound on a more relaxed formula $\mathcal{F}'$, where the occurrences of $(\texttt{NOT}, Q)$ are replaced by $(\texttt{NOT}, Q')$ and Q' is a copy of Q. Note, however, that the optimality in the evaluation of $\mathcal{F}'$ does not imply the optimality on F, because a worst-case instance of $\mathcal{F}'$ may choose different instances for Q and Q'.

5 FULL RELATIONAL ALGEBRA ON LQDAGS

Lqdags can be elegantly extended to handle the full relational algebra, even if in general we cannot provide optimality guarantees when the remaining operations are included. We consider in the sequel the operations not covered in Section 4 (recall that set difference is handled as in Eq. (2), and its optimality holds under the same conditions of Theorem 4.5).

First, note that attribute renaming, $\rho_{A_i/A_j}(\mathcal{L})$, requires no computation in our framework: we retain the same lqdag of $\mathcal{L}(\mathcal{A})$ but interpret it as $\mathcal{L}(\mathcal{A}')$, where $\mathcal{A}' = \mathcal{A} \setminus \{A_i\} \cup \{A_j\}$, assuming that the dimension of $\mathcal{L}$ that represented A_i now represents A_j. Relation renaming is also immaterial. Note that renaming can be used to alter the order of the attributes in a relation, and to include it with various orders in the same formula. This is supported in our framework with various qdags (Q, M_i) referring to the same quadtree and with different permutations M_i of its attributes.

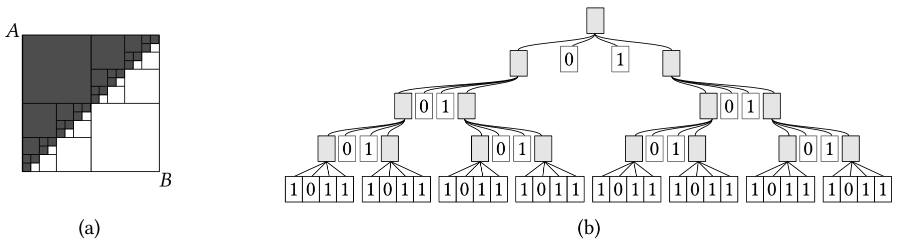


Fig. 9. An illustration of Q_θ for the predicate $\theta : A < B$, with A, B attributes. a) A representation in a grid of the tuples matching θ when $\ell = 2^4$. b) The quadtree that represents the tuples matching θ.

5.1 Selection and θ-join

The selection operation in relational algebra, $\sigma_\theta(F)$, takes a subexpression F and a *predicate* θ, which is a logical expression on the attributes of F. This can be written with our lqdags as

$$(\texttt{SELECT}, \mathcal{L}_F(\mathcal{A}), \theta) \ = \ (\texttt{AND}, \mathcal{L}_F(\mathcal{A}), pred(\theta, \mathcal{A})), \tag{3}$$

where $\mathcal{L}_F$ is an lqdag representing F, and $pred(\theta, \mathcal{A})$ is a virtual lqdag on $\mathcal{A}$ whose cells are exactly those that satisfy the predicate θ. For example, a simple predicate like $\theta \equiv (A_i = a)$, which fixes the value of the attribute A_i, can be written as

$$pred((A_i = a), \mathcal{A}) = (\texttt{EXTEND}, (\texttt{QTREE}, \langle a \rangle), \mathcal{A} \setminus \{A_i\}),$$

where $\langle a \rangle$ is a quadtree in one dimension, on attribute A_i, with a single cell with value a. Logical disjunctions, conjunctions, and negations in θ can be handled by using the lqdag operations OR, AND, and NOT, respectively. All those $pred(\theta, \mathcal{A})$ lqdags can then be easily formed from very small quadtrees and the Boolean lqdag operations.

However, θ can be in general a logical formula over more complex conditions than an attribute being equal to a constant: two attributes, or an attribute and a constant, can be compared with any operator in $\{=, \neq, \leq, \geq, <, >\}$. A general way to handle any such predicate is to set $pred(\theta, \mathcal{A}) = (\texttt{QTREE}, Q_\theta)$ without building an actual quadtree Q_θ on $\mathcal{A}$ (see Figure 9 for an example of Q_θ). Instead, we simulate the navigation through Q_θ, keeping track of the subgrid boundaries as we descend, and returning Value for the current node as follows:

- 0 if θ does not hold for any cell in the current subgrid;
- 1 if θ holds for every cell in the current subgrid;
- ½ otherwise.

Whether θ holds for some or for all cells in a given subgrid of the output can be easily determined in time $O(|\theta|)$ for the logical formulas comparing attributes and constants we have discussed above (e.g., as in Figure 9, $A < B$ holds somewhere iff $s_a < e_b$ and it holds everywhere if $e_a < s_b$, where $[s_a, e_a]$ and $[e_b, s_b]$ are the ranges of values of the attributes A and B, respectively, in the subgrid). For a formula F represented by the lqdag $\mathcal{L}_F = (\texttt{SELECT}, \mathcal{L}, \theta)$, if we count $|\theta|$ as a part of $|F|$, then we can compute the output of $\mathcal{L}_F$ in time $O(|Q_F^+| \cdot 2^d |F|)$, where Q_F^+ is the super-completion of $\mathcal{L}_F$. This complexity is indeed worst-case optimal when θ selects tuples from a single relation R: the AGM bound for $\mathcal{L}_F = (\texttt{AND}, (\texttt{QTREE}, R), (\texttt{QTREE}, Q_\theta))$ is $F(D)^* = \min(|R|, |Q_\theta|)$ because there exists a relation R' with $|R|$ points placed at the coordinates that satisfy θ, unless of course there are fewer of the latter. The quadtree Q_F^+ we virtually traverse, on the other hand, contains exactly the ancestors of the leaves of the quadtree of R that also belong to Q_θ, of which there are at most $O(F(D)^* \log \min(\ell, S))$. We then obtain the same bounds of Theorem 4.5.

The complexity of evaluating our formula might not be worst-case optimal, however, when θ combines attributes from two or more tables, because $|Q_F^+|$ may no longer be bounded by $F(D)^*$ in that case. To see this, let

$\mathcal{L}_R, \mathcal{L}_S$ be lqdags representing relations $R(A,B), S(B,C)$, respectively, and consider the lqdag

$$\mathcal{L}_F \;=\; (\mathtt{SELECT}, (\mathtt{JOIN}, \mathcal{L}_R, \mathcal{L}_S), A = C).$$

Because each value of A in R can now match only one value of C in S, $F(D)^*$ is linear in the size of R and S. In terms of lqdags, however, computing $Q^+_{\mathcal{L}_F}$ amounts to intersecting the lqdag $(\mathtt{JOIN}, \mathcal{L}_R, \mathcal{L}_S)$ with a virtual tree $Q_{A=C}$ of dimensions $\{A, C\}$ and whose points lie on the diagonal where $A = C$. The problem is that checking whether a point in $(\mathtt{JOIN}, \mathcal{L}_R, \mathcal{L}_S)$ intersects with this diagonal may require inspecting all the way down to the leaves of this lqdag. This implies that the size of the non-pruned completion Q^+_F is actually bounded by $|R \bowtie S|$.

In terms of our results, if we apply Theorem 4.5 on the formula of Eq. (3), then $F(D)^*$ is allowed to replace Q_θ by any other relation of the same cardinality on $\mathcal{A}$. In the example above, the running time for $\sigma_{A=C}(R \bowtie S)$ is bounded by the worst case running time of the join of R, S and any other binary relation with ℓ tuples.

Therefore, we can retain worst-case optimality upon selections whenever the predicates can be pushed down to the leaves of the syntax tree, that is, to the individual relations. Otherwise, our technique still handles them correctly, though not optimally. The same happens if θ is an arbitrary formula whose validity can only be tested for individual cells; in this case we can only evaluate F and discard one by one the cells that do not satisfy θ.

The Cartesian product is a variant of the join where the relations have no common attributes; our join formula in Eq. (1) actually computes the Cartesian product in this case. The θ-join selects the tuples from the Cartesian product that satisfy the predicate θ. We can then define

$$(\mathtt{THETAJOIN}, \mathcal{L}_1, \mathcal{L}_2, \theta) \;=\; (\mathtt{SELECT}, (\mathtt{JOIN}, \mathcal{L}_1, \mathcal{L}_2), \theta),$$

which also allows the relations to share attributes. Once again, our lqdag evaluation of this expression is correct but not worst-case optimal.

5.2 Projection and derivatives

The projection, $\pi_{\mathcal{A}'}(F)$, takes a subexpression F on attributes $\mathcal{A}$ and projects it onto $\mathcal{A}' \subseteq \mathcal{A}$. We define the corresponding lqdag $\mathcal{L} = (\mathtt{PROJECT}, \mathcal{L}_F(\mathcal{A}), \mathcal{A}')$ as follows. Let $|\mathcal{A}| = d$, $|\mathcal{A}'| = d'$, and $\mathcal{L}_F$ be an lqdag for F. If $\textsc{Value}(\mathcal{L}_F)$ is 0 or 1 then $\textsc{Value}(\mathcal{L}) = \textsc{Value}(\mathcal{L}_F)$, otherwise $\textsc{Value}(\mathcal{L}) = \diamond$. In the latter case, the root of the non-pruned completion $Q^+_{\mathcal{L}_F}$ of $\mathcal{L}_F$ has 2^d children, and the root of the non-pruned completion of $Q^+_{\mathcal{L}}$ of $\mathcal{L}$ has $2^{d'}$ children. The i-th child of $Q^+_{\mathcal{L}}$ is computed as the $\mathtt{OR}$ of all children j of $Q^+_{\mathcal{L}_F}$ such that the projection of the d-bit representation of j to the positions in which attributes in $\mathcal{A}'$ appear in $\mathcal{A}$ is precisely the d'-bit representation of i (see Figure 10 for an illustration). For example, if $\textsc{Value}(\mathcal{L}_F(\{A,B,C\}))$ is not 0 or 1 and $\mathcal{L} = (\mathtt{PROJECT}, \mathcal{L}_F, \{A,B\})$, we have that

$$\begin{aligned}
\textsc{Child}(\mathcal{L}, 0) &= (\mathtt{PROJECT}, (\mathtt{OR}, \textsc{Child}(\mathcal{L}_F, 0), \textsc{Child}(\mathcal{L}_F, 1)), \{A, B\});\\
\textsc{Child}(\mathcal{L}, 1) &= (\mathtt{PROJECT}, (\mathtt{OR}, \textsc{Child}(\mathcal{L}_F, 2), \textsc{Child}(\mathcal{L}_F, 3)), \{A, B\});\\
\textsc{Child}(\mathcal{L}, 2) &= (\mathtt{PROJECT}, (\mathtt{OR}, \textsc{Child}(\mathcal{L}_F, 4), \textsc{Child}(\mathcal{L}_F, 5)), \{A, B\});\\
\textsc{Child}(\mathcal{L}, 3) &= (\mathtt{PROJECT}, (\mathtt{OR}, \textsc{Child}(\mathcal{L}_F, 6), \textsc{Child}(\mathcal{L}_F, 7)), \{A, B\}).
\end{aligned}$$

Supporting projection on lqdags then enables the remaining operations of the relational algebra:

$$\begin{aligned}
(\mathtt{SEMIJOIN}, \mathcal{L}_1(\mathcal{A}_1), \mathcal{L}_2(\mathcal{A}_2)) &= (\mathtt{PROJECT}, (\mathtt{JOIN}, \mathcal{L}_1, \mathcal{L}_2), \mathcal{A}_1)\\
(\mathtt{ANTIJOIN}, \mathcal{L}_1(\mathcal{A}_1), \mathcal{L}_2(\mathcal{A}_2)) &= (\mathtt{PROJECT}, (\mathtt{JOIN}, \mathcal{L}_1, (\mathtt{NOT}, \mathcal{L}_2)), \mathcal{A}_1)\\
(\mathtt{DIVISION}, \mathcal{L}_1(\mathcal{A}_1), \mathcal{L}_2(\mathcal{A}_2)) &= (\mathtt{DIFF}, \mathcal{L}'_1, (\mathtt{PROJECT}, (\mathtt{DIFF}, (\mathtt{JOIN}, (\mathcal{L}'_1, \mathcal{L}_2), \mathcal{L}_1), \mathcal{A}_1 \setminus \mathcal{A}_2))\\
&\ \text{where } \mathcal{L}'_1 = (\mathtt{PROJECT}, \mathcal{L}_1, \mathcal{A}_1 \setminus \mathcal{A}_2) \text{ and } \mathcal{A}_2 \subseteq \mathcal{A}_1
\end{aligned}$$

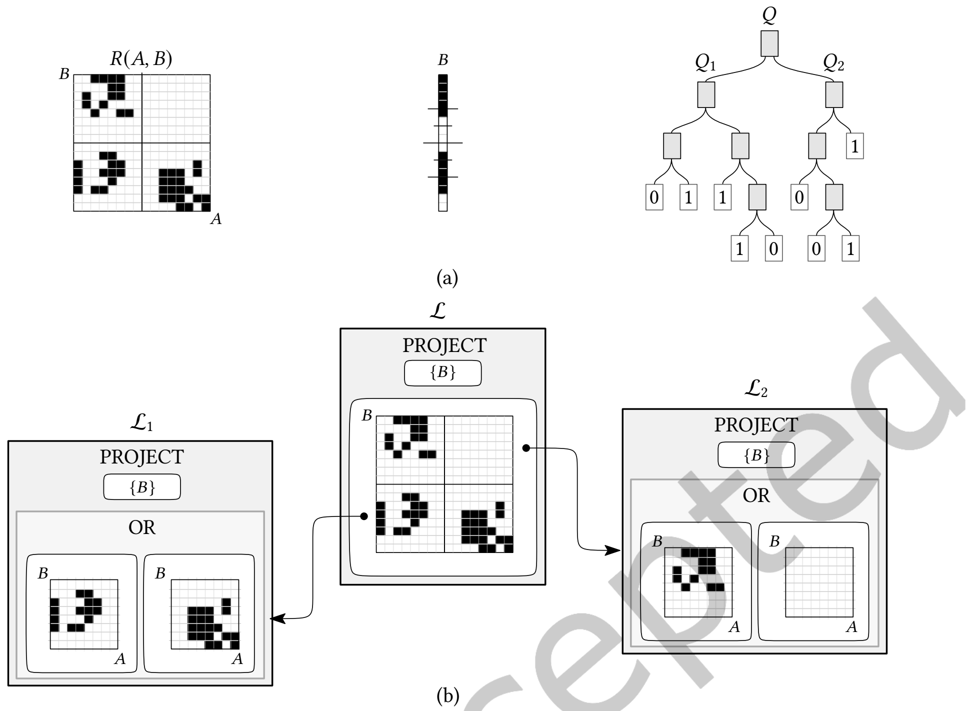


Fig. 10. An illustration of the Child operation for a 2-dimensional lqdag with functor PROJECT. a) A relation $R(A, B)$ in a two dimensional grid (left), its projection to B in a 1-dimensional grid (center), and the quadtree Q_F that represents the projection (right). b) An lqdag $\mathcal{L}$ = PROJECT((QTREE, Q_R), $\{B\}$) whose completion is Q, with Q_R denoting the quadtree that represents $R(A, B)$. Note that Q_1, Q_2 in a) are the completions of $\mathcal{L}_1, \mathcal{L}_2$ in b), respectively. Thus, Child$(\mathcal{L}, 0) = \mathcal{L}_1$, and Child$(\mathcal{L}, 1) = \mathcal{L}_2$.

Just as in the case of selection, our strategy to handle projections does not ensure worst-case optimality either. Consider, for example, the projection F represented by the lqdag

$$\mathcal{L}_F \ = \ (\texttt{PROJECT}, (\texttt{JOIN}, \mathcal{L}_R, \mathcal{L}_S), \{A\}),$$

where $\mathcal{L}_R, \mathcal{L}_S$ are lqdags representing relations $R(A, B), S(B, C)$, respectively. Since its output is of dimension 1, $F(D)^* \leq \ell$. We have, essentially, the same problem encountered for the case of selection: As we navigate $\mathcal{L}_F$ to project out values in the lqdag (JOIN, $\mathcal{L}_R, \mathcal{L}_S$), we may have to inspect all the way down to the leaves. In total, the non-pruned completion $Q^+_{\mathcal{L}_F}$ of $\mathcal{L}_F$ may be of size $|R \bowtie S|$.

A projection $\mathcal{L}$ = (PROJECT, $\mathcal{L}_F(\mathcal{A}), \mathcal{A}'$) from a d-dimensional lqdag $\mathcal{L}_F$ to d' dimensions has one further problem: its Child translation makes the resulting formula to grow by $2^{d-d'}$ elements every time we move to a child in $Q^+_{\mathcal{L}}$. Thus, during a traversal of $Q^+_{\mathcal{L}}$, the lqdag $\mathcal{L}$ may grow up to size $|\mathcal{L}| \cdot 2^{c(d-d')}$ for nodes at level c, possibly becoming as large as $Q_{\mathcal{L}_F}$. Therefore, the size $|\mathcal{L}|$ cannot be anymore considered constant in data complexity when projections are considered.

A consequence of the non-optimality of our strategy in this section is that, unlike in Section 4, the way in which the formula is written does matter. For example, if we rewrite $\mathcal{L}_F$ in the preceding example as

$$(\texttt{PROJECT},\ (\texttt{JOIN}, \mathcal{L}_R, (\texttt{AND}, (\texttt{PROJECT}, \mathcal{L}_R, \{B\}), (\texttt{PROJECT}, \mathcal{L}_S, \{B\}))),\ \{A\}),$$

then we have $|Q^+_{\mathcal{L}_F}| \leq |R| + |S|$. On the other hand, as noted, evaluating each node of $Q^+_{\mathcal{L}_F}$ can be more expensive now because the formula may grow due to the internal projections.

5.2.1 Yannakakis' algorithm. Having defined the projection, a natural question is whether one can use it to obtain finer bounds for acyclic queries or for queries with bounded treewidth. For example, even though the AGM bound for $R(A,B) \bowtie S(B,C)$ is quadratic, one can use Yannakakis' algorithm [44] to compute it in time $O(|R| + |S| + |R \bowtie S|)$. This is commonly achieved by first computing $\pi_B(R)$ and $\pi_B(S)$, intersecting them, and then using this join to filter out R and S. Unfortunately, expressing this strategy directly in our lqdag framework, even if we push down the projections,

$$\begin{aligned}&(\texttt{JOIN}, (\texttt{JOIN}, \mathcal{L}_R, \mathcal{I}), (\texttt{JOIN}, \mathcal{L}_S, \mathcal{I}))\\ &\text{where } \mathcal{I} \;=\; (\texttt{AND}, (\texttt{PROJECT}, \mathcal{L}_R, \{B\}), (\texttt{PROJECT}, \mathcal{L}_S, \{B\})),\end{aligned}$$

would still give us a quadratic algorithm, even for queries with small output, because after extending $\mathcal{I}$ it may take quadratic time to compute the join.

More generally, this also rules out the possibility to achieve optimal bounds for queries with bounded treewidth or similar measures. We return to this point in the Conclusions.

5.3 Partial materialization of query results

The algorithms we have described can generate the output Q of the formula F explicitly, in the same compressed quadtree format of the input relations, and as such they can be used compositionally. It is worth reminding that the output of a join query can be considerably larger than the input relations, which is particularly relevant in the context of using little space for query processing. In this section we show another advantage of using quadtrees for representing the query results: Because the output is a function of the input relations, it can be not only represented as a compressed quadtree, but also *materialized only partially*, as long as one is willing to pay extra time to navigate such output. This parameterizable materialization of the query output, where we can trade space for traversal time, is inspired by a similar result by Deep and Koutris [10].

We compute a partial quadtree representation $\tilde{Q}$ that contains $|\tilde{Q}| \leq \min(|Q|, |Q^+|/\tau)$ nodes, for any parameter $\tau \geq 1$ fixed at query time, and allows us to traverse the result with delay $O(\tau \cdot 2^d V(F) \log \min(\ell, S))$, where $V(F)$ is the time to compute VALUE on F using Algorithm 7. The time to produce $\tilde{Q}$ is the same as for generating Q using Algorithm 9, that is, $O(|Q^+| \cdot 2^d V(F))$.

Our partial result $\tilde{Q}$ is a version of Q where some subtrees are pruned. The quadtree representation we use for $\tilde{Q}$ is a slight extension of the one we have been using. In this case, leaves may have VALUE 0, 1, or ½. A leaf node with value ½ means that the result Q has elements below that node but these are not stored in $\tilde{Q}$, and thus must be recomputed (see Figure 11 for an illustration). In terms of the representation of Lemma 2.1, a leaf with value ½ is seen as an internal node with no children, a configuration that cannot occur in the basic quadtrees but that the compact representation allows.

The key is that leaves v with value ½ in $\tilde{Q}$ will be produced only when, in a DFS traversal of Q^+ below v, (1) the number of internal nodes of Q^+ visited between any two consecutive internal nodes of Q is at most τ, and (2) the number of internal nodes of Q^+ visited before the first and after the last internal node of Q, are at most $\tau/2$. This then guarantees the promised delay when rerunning the intersection algorithm to recover the values omitted below v in $\tilde{Q}$.

To produce $\tilde{Q}$ instead of Q, we modify Algorithm 9 so that every recursive call may, after computing the subtree of Q below a node v in line 6, choose to return instead a leaf with value ½. In case it returns a leaf, the algorithm accompanies it with the following numbers:

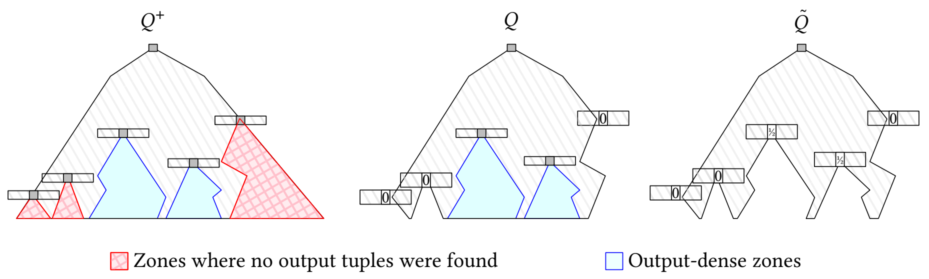


Fig. 11. A comparison between Q, Q^+, and $\tilde{Q}$. The quadtree Q is obtained by pruning from Q^+ the subtrees explored by Algorithm 9 that did not produce an output tuple. Similarly, $\tilde{Q}$ is obtained from Q by pruning the subtrees of Q for which, to produce an output tuple, Algorithm 9 does not traverse more than τ internal nodes before finding it. We call the subtrees meeting this property output-dense zones.

- For a leaf with Value = 0, the number c of internal nodes of Q^+ visited below v (this is the number of times it reached line 2 in further recursive calls).
- For a leaf with Value = ½, the numbers l and r of internal nodes of Q^+ visited before the first internal node of Q and after the last internal node of Q, respectively.
- For a leaf with Value = 1, the numbers $l = r = 0$.

The algorithm decides as follows whether to return a quadtree with children $C_0, \ldots, C_{2^d-1}$ or to prune them and return instead a leaf with value ½. If its children are integer nodes (i.e., the last level), it always prunes them because it can be recomputed in time $O(2^d V(F))$, which is within our time budget. Otherwise, if some child is not a leaf, the algorithm never prunes the node: if the child was not converted into a leaf, it is because the delay τ cannot be guaranteed below it. Otherwise, if all the children C_i are leaves, the algorithm may choose to prune the node, as follows. Let C_{i_j} be the children leaves with Value = ½ or 1, for $0 \leq i_1 < \cdots < i_k < 2^d$. Then, if

$$l \;=\; \left(\sum_{s=0}^{i_1-1} c(C_s)\right) + l(C_{i_1}) \;\leq\; \tau/2, \tag{4}$$

$$r \;=\; r(C_{i_k}) + \left(\sum_{s=i_k+1}^{2^d-1} c(C_s)\right) \;\leq\; \tau/2, \text{ and} \tag{5}$$

$$r(C_{i_j}) + \left(\sum_{s=i_j+1}^{i_{j+1}-1} c(C_s)\right) + l(C_{i_{j+1}}) \;\leq\; \tau \text{ for all } j = 1, \ldots, k-1,$$

the algorithm prunes the node and returns instead a leaf with value ½, accompanied with the values l and r computed in Eqs. (4) and (5).

The way to traverse the full result Q from $\tilde{Q}$ is to execute Algorithm 9 as an iterator, which also traverses $\tilde{Q}$ in synchronization. Every time we reach a leaf v with value 0 or 1 in $\tilde{Q}$, we skip the recursion of Algorithm 9 at v, because we know that Q has no results (0) or is full of results (1) in the subgrid of v. If, instead, we arrive at a leaf v of $\tilde{Q}$ with value ½, we traverse all the subtree of v in Q using Algorithm 9 as an iterator, knowing that the promised delay is attained below v: we obtain a true internal node of Q every, at most, τ internal nodes we traverse in Q^+.

The space guarantee is obtained by considering that, on the one hand, the nodes of $\tilde{Q}$ are a subset of the internal nodes of Q. On the other hand, if we regard the internal nodes of Q in DFS order, then for each internal node of Q we include in $\tilde{Q}$ there are more than $\tau/2$ internal nodes of Q^+ and not in $\tilde{Q}$ preceding or following it, therefore $\tilde{Q}$ contains less than $4|Q^+|/\tau$ internal nodes of Q (so we can build our structures with 4τ instead of τ to obtain our precise promised bounds).[7]

By representing $\tilde{Q}$ with the data structure of Lemma 2.1, and since we may have to traverse up to $2^d \log \min(\ell, S)$ nodes of Q between two consecutive leaves with value 1 (i.e., two consecutive output elements), we obtain the following result.

Corollary 5.1. *Assume lqdags can compute the output of a relational formula F in time $V(F)$ per node of Q_F^+. Then, given a parameter τ chosen at query time, we can compute a partial representation of the output in time $O(|Q_F^+| \cdot 2^d V(F))$, which takes $O(|Q_F^+| \cdot d/\tau)$ bits using the encoding of Lemma 2.1, and retrieves the successive elements of the output of F with a delay of $O(\tau \cdot 2^d V(F) \log \min(\ell, S))$.*

This corollary then extends to the full relational algebra the result Deep and Koutris [10] had obtained for join queries. We remark that $V(F) = O(|F|)$ if F does not contain projections, and that $|Q_F^+| \leq F(D)^* \log \min(\ell, S)$ for JUC queries.

6 ENGINEERING AND IMPLEMENTATION

We implemented our representation and multijoin algorithm described in Section 3, plus the simplest selection described in the beginning of Section 5.1. This section describes the algorithm engineering and implementation, and the next one its experimental evaluation.

6.1 Space-efficient qdags

We do not use the tries [7] of the theoretical proposal (Lemma 2.1) to implement the quadtrees, but the simpler k^d-trees of Brisaboa et al. [9] with parameter $k = 2$. Such a k^d-tree represents each internal quadtree node as the 2^d bits telling which of its quadrants is empty (0) or nonempty (1). Note that, for the deepest internal nodes, this coincides with the values corresponding to its children, which are integer nodes. Leaves and integer nodes are then not represented, because their data is deduced from the corresponding bit of their parent. We exploit the simplicity of k^d-trees to enable speedups and direct compressed construction of the output, as described in the sequel.

The k^d-tree is just a bitvector V concatenating the 2^d bits that represent every internal node in levelwise order. Each node is identified with the first position of its 2^d bits in this concatenation, the first position being 1. Bitvector V supports navigating towards a node's children using a succinct data structure to support rank operation: $\mathsf{rank}(V, i)$ counts the number of 1s in $V[1..i]$. This operation can be computed in $O(1)$ time using $o(|V|)$ additional bits on top of V [26]. With the rank operation at hand we can then simulate the traversal of the k^d-tree: For an internal node x (i.e., whose description starts at $V[x]$), its j-th child ($0 \leq j < 2^d$) is (i.e., its description starts in V at position) $\mathsf{child}(x, j) = 2^d \cdot \mathsf{rank}(V, x + j) + 1$; this is computed in $O(1)$ time.

In practice, the bitvector V is cut in two parts, $V = T \cdot L$, where L concatenates the 2^d-bit descriptions of the deepest-level internal nodes, and T those of the higher nodes. That is, L stores the content of all the integer nodes (see Figure 12 for an example). This is advantageous because the rank operation must only be supported on T, whereas for L we only need to support access.

On a quadtree in dimension d storing p points, the length of the bitvector V is $|V| \leq 2^d p \log \ell$, which increases exponentially with d. In high dimensions, this is much more than the $(d + 2 + o(1))p \log \ell$ bits used in Lemma 2.1. A practical alternative to reduce the space in higher dimensions is to exploit the fact that V has at most $p \log \ell$ 1s, one per ancestor of each integer-1 node. A sparse bitvector representation [36] then also uses $(d + 2 + o(1))p \log \ell$

[7]Every node in $\tilde{Q}$ has $\tau/2$ nodes not in $\tilde{Q}$ preceding or following it, but those can be the same $\tau/2$ nodes that follow the previous node of $\tilde{Q}$ or that precede the next node of $\tilde{Q}$, respectively. Thus we can exclusively assign only $\tau/4$ nodes not in $\tilde{Q}$ to each node in $\tilde{Q}$. Therefore, $\tilde{Q} \leq |Q^+|/(\tau/4)$.

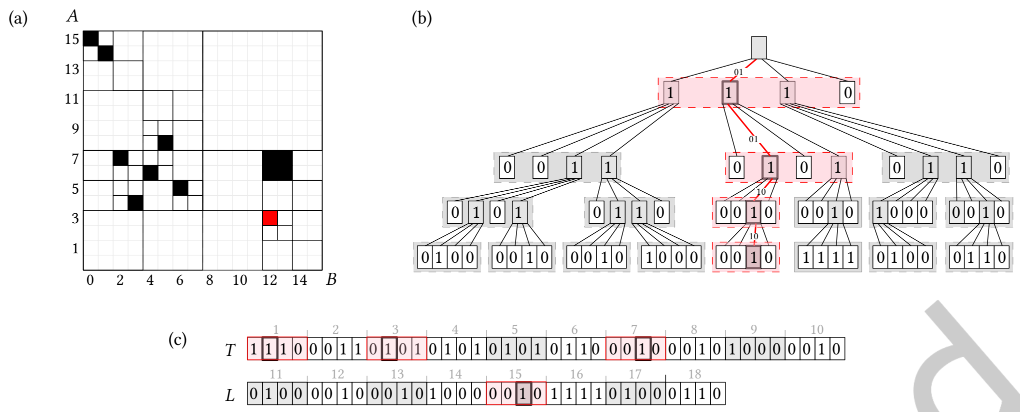


Fig. 12. A k^2-tree representing the same relation $R(A, B)$ of Figure 1. (a) Representation of $R(A, B)$ in a $2^4 \times 2^4$ grid. (b) A quadtree representing R, and the respective k^2-tree nodes. A 1-bit is assigned to every internal and integer-1 node of the quadtree, and a 0-bit to every leaf and integer-0 node. The k^2-tree combines the 2^d children of a node of the quadtree into one node. The k^2-tree nodes are highlighted as shaded rectangles with dashed borders. The integer-1 node of the quadtree in bold correspond to $(a, b) = (3, 12)$, highlighted in red in the grid. (c) The k^2-tree represents R using only the bitvector $V = T \cdot L$, where T represents the bits of the nodes at all levels but the last, in level-wise order, and L stores the bits of the bottom level. The identifier of each node is represented as a small gray number on top of each node. The nodes highlighted in red, and the bolder bits within, correspond to the root-to-leaf path encoding the tuple $(3, 12)$.

bits, though it supports rank in time $O(d)$. We choose not to implement this option, as we were focusing only on relatively low dimensions. However, this is an interesting direction for future work.

In practice, not all the domains are of the same size ℓ. We extend them all to the next common power of 2. This does not pose a significant overhead because the points are clustered within a subgrid of the actual size.

6.1.1 Our k^d-tree Implementation. Contrary to the typical k^d-tree representation just described (for $k = 2$), we use a set of $b = \log \ell$ bit vectors $B_0, B_1, \ldots, B_{b-1}$ to store each level of the quadtree separately. This will facilitate appending new nodes as we compute the join, so that we can produce the output directly in compressed form. Each bitvector B_i stores the nodes at depth i, using the 2^d-bit encoding explained above. It is easy to see that the j-th child of a node x at depth $i < \log \ell$ starts at position $2^d \cdot \mathrm{rank}(B_i, x + j) + 1$ in B_{i+1}. Again, bitvector B_{b-1} needs not support rank.

Although there are highly-optimized and practical approaches to support operation rank [14], we obtain good results with the following ad-hoc scheme, optimized for current processors of $w = 64$ bit words. For a bitvector $B[1..N]$, we store a precomputed table P such that $P[j] = \mathrm{rank}(B, w \cdot j)$, for $j = 0, \ldots, \lfloor N/w \rfloor$. If 32-bit integers are used for the cells of P, then it takes $N/2$ bits on top of B. Operation $\mathrm{rank}(B, x)$ is computed as $P[\lfloor x/w \rfloor]$ plus the number of 1s within $B[1 + \lfloor x/w \rfloor \cdot w, x]$. The latter is computed using a *popcount* operation within the w-bit word storing $B[x]$. In our particular implementation, with $w = 64$, we use the special popcount operation from the SSE 4.2 instruction set. This supports rank efficiently: almost twice as fast as the highly-optimized implementation from the `sdsl` library [14]. The precomputation of P takes $O(N/w)$ time.

6.2 Implementing extensions

To compute the join $R_1(\mathcal{A}_1) \bowtie \cdots \bowtie R_n(\mathcal{A}_n)$, each $R_r(\mathcal{A}_r)$ represented with a qdag $Q_r = (T_r, I_r)$, we must first compute operation $\textsc{Extend}(Q_r, \mathcal{A} \setminus \mathcal{A}_r)$, for $\mathcal{A} = \cup_{r=1}^{n} \mathcal{A}_r$. We regard the attributes A_j as integer identifiers in $0, \ldots, |\mathcal{A}| - 1$, and use them to give an order in $\mathcal{A}$. Along with Q_r, we represent the attribute set $\mathcal{A}_r$ using an integer a_r with the corresponding attribute identifiers: We precompute a_r starting with $a_r \leftarrow 0$ and then, for each $A_j \in \mathcal{A}_r$, do $a_r \leftarrow a_r \mid (1 \ll (A_j - 1))$, where operators $\mid$ and $\ll$ denote bitwise-or and shift-left, respectively. Then, line 6 of Algorithm 3, which is run for $i = 0, \ldots, 2^{|\mathcal{A}|} - 1$, is implemented as $i' \leftarrow i \,\&\, a'$, where a' represents $\mathcal{A}'$ and & denotes bitwise-and. This encodes in i' the projection of i to the positions in which the attributes of $\mathcal{A}'$ appear in $\mathcal{A}$, which can then be used in line 7 of Algorithm 3. The time complexity of this implementation is $O(2^{|\mathcal{A}|})$ under the assumption that $|\mathcal{A}| = O(w)$; otherwise we can obtain $O(2^{|\mathcal{A}|} \cdot |\mathcal{A}|/w)$ time by representing the numbers using multiple computer words.

6.3 Implementing multiway intersections

Once we have extended every qdaq $Q_r = (T_r, I_r)$ representing R_r to $Q_r^* = (T_r, M_r) = \textsc{Extend}(Q_r, \mathcal{A} \setminus \mathcal{A}_r)$ in order to compute the join $R_1(\mathcal{A}_1) \bowtie \cdots \bowtie R_n(\mathcal{A}_n)$, we proceed to intersect the qdags Q_r^* by traversing them all in synchronization, as explained.

The recursive traversal algorithm takes the current node x_r in each qdag Q_r^*, as well as the current level (or depth) v they have in the tree. It is invoked with $v = 0$ and $x_1 = \cdots = x_n = 1$, the qdag roots. At each step, for $j = 0, \ldots, 2^d - 1$, it recurses on the j-th child of every x_r if the n children are not null; otherwise the traversal is pruned and a leaf in the resulting quadtree is written. Formally, we recurse on the j-th child of $x_1, \ldots, x_n$ if it holds that $T_r.B_v[x_r + M_r[j]] = 1$ for all $1 \leq i \leq n$. Checking this naively takes $O(2^d n)$ time per node (as it shows up in Theorem 3.9), which is inefficient when M_r maps to the same child in T_r for many different values of j.

6.3.1 *Materializing nodes.*

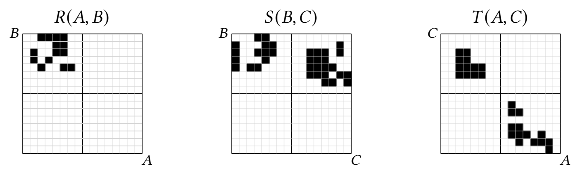
We now introduce a more efficient approach to reduce this $O(2^d n)$ cost. The main idea is to quickly *materialize* the 2^d children of every node $x_r \in Q_r^*$, and then intersect the n sets of children using bit-parallel operations, instead of computing the intersection child-wise.

Recall that, for each qdag Q_r^*, we only store explicitly the k^{d_r}-tree of the base quadtree T_r; the traversal on the virtual k^d-tree representing Q_r^* is simulated through mapping M_r. Therefore, the sequence of 2^d bits that describes the children of x_r in Q_r^* is a function of the 2^{d_r} children of $x_r \in T_r$ (which are contiguous in its k^{d_r}-tree representation) and of the mapping M_r.

To quickly materialize the 2^d children of $x_r \in Q_r^*$, we precompute a table C_r on which the 2^{d_r}-bit encoding e of the children of $x_r \in T_r$ is used as an index to obtain in $C_r[e]$ the corresponding 2^d-bit encoding of the children of $x_r \in Q_r^*$. Table C_r has $2^{2^{d_r}}$ entries, the number of possible encodings of 2^{d_r} bits (see Figure 13 for an example). Each entry stores the corresponding 2^d-bit encoding, so the overall size of C_r is $2^{2^{d_r}+d}$ bits. We build all the tables C_r, for $r = 1, \ldots, n$, on the fly, before running the intersection, in time $O(2^{2^{d_r}+d})$, by obtaining each bit of each entry using M_r. The extra time and space, $\sum_{r=1}^{n} O(2^{2^{d_r}+d})$, is $O(2^{2^d+d} n)$ in the worst case, yet in general it is considerably smaller. These tables can be discarded once the intersection is completed.

Once the tables C_r are built, the traversal takes only $O(\lceil 2^d/w \rceil n)$ time per recursion step, where w is the number of bits in the computer word. We then obtain a w-fold speedup compared to the naive algorithm (typically $w = 64$, but it can be larger if we use broadword operations). Being at nodes $x_1, \ldots, x_n$ during the traversal, we extract in constant time the 2^{d_r} contiguous bits encoding the children of each $x_r \in T_r$, $e_r = B_v[x_r..x_r + 2^{d_r} - 1]$, and lookup in C_r its extended version of 2^d bits, $e_r^* = C_r[e_r]$. We then intersect the bitvectors e_r^* using bitwise-and operations (&), which operate w bits in $O(1)$ time, $e^* = e_1^* \,\&\, \cdots \,\&\, e_n^*$. This takes $O(\lceil 2^d/w \rceil n)$ time (we stop earlier if the result is all zeros before operating all the e_r^*s).

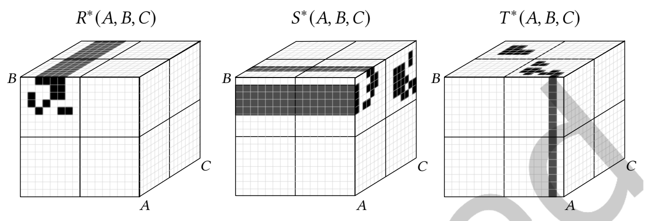
Finally, we must recurse on the children j of all $x_1, \ldots, x_n$ whenever $e^*[j] = 1$. To obtain each such 1 from e^* in constant time, we use the `__builtin_clz` built-in function of `gcc` compilers to count the number of leading zeroes in e^*, hence finding its most-significant bit set. This built-in function is implemented by hardware in most modern CPU architectures, yet it can also be easily implemented [21] if not provided by the processor. This constant time can then be charged to each child we visit.



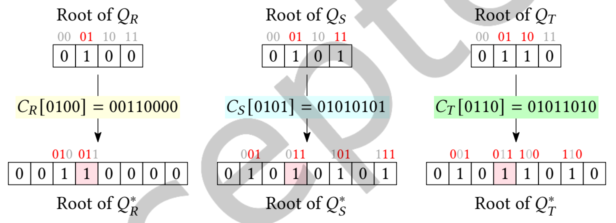
(a)

	i	$C_{R(A,B)}$	$C_{S(B,C)}$	$C_{T(A,C)}$
0	0000	00000000	00000000	00000000
1	0001	00000011	00010001	00000101
2	0010	00001100	00100010	00001010
3	0011	00001111	00110011	00001111
4	0100	00110000	01000100	01010000
5	0101	00110011	01010101	01010101
6	0110	00111100	01100110	01011010
7	0111	00111111	01110111	01011111
8	1000	11000000	10001000	10100000
9	1001	11000011	10011001	10100101
10	1010	11001100	10101010	10101010
11	1011	11001111	10111011	10101111
12	1100	11110000	11001100	11110000
13	1101	11110011	11011101	11110101
14	1110	11111100	11101110	11111010
15	1111	11111111	11111111	11111111

(b)



(c)



(d)

Fig. 13. Materialization of the k^d-tree nodes of the extended relations using lookup tables. (a) Three relations $R(A,B)$, $S(B,C)$ and $T(A,C)$. We denote by Q_R, Q_S and Q_T the k^2-trees representing R, S, T, respectively. (b) The lookup tables C_R, C_S, C_T for the materialization of extended nodes when computing $R(A,B) \bowtie S(B,C) \bowtie T(A,C)$. (c) The relations $R^*(A,B,C) = R(A,B) \times \mathsf{All}(C)$, $S^*(A,B,C) = S(B,C) \times \mathsf{All}(A)$, $T^*(A,B,C) = T(A,C) \times \mathsf{All}(B)$. We denote by Q_R^*, Q_S^* and Q_T^* the (virtual) k^3-trees representing R^*, S^*, T^*, respectively. (d) The k^3-tree nodes corresponding to the roots of Q_R^*, Q_S^*, Q_T^* are looked up in C_R, C_S, C_T, respectively, using the root node of Q_R, Q_S, Q_T as the index in the table. The bits highlighted in red belong to the intersection of the root nodes of Q_R^*, Q_S^*, Q_T^* since $00110000 \,\&\, 01010101 \& \, 01011010 \;=\; 00010000$.

In our experiments it holds that $2^{d_r} \leq w$, and therefore we can handle the indices in C_r in constant time as if they were numbers. The tables C_r are also reasonably small and fast to build. For larger d_r values, we can split the bitvectors e_r into chunks of a maximum allowed size $c \leq w$, $e_r = e_r^1 \cdots e_r^s$ for $s = \lceil 2^{d_r}/c \rceil$, and create tables $C_r^1, \ldots, C_r^s$ for each chunk, such that each C_r^t has 2^c entries storing 2^d bits, where only those activated by the t-th chunk of e_r are set. We then simulate $C_r[e] = C_r^1[e_r^1] \mid \cdots \mid C_r^s[e_r^s]$ in time $O(\lceil 2^d/w \rceil s)$, using bitwise-or operations ($\mid$). We now spend $O(\sum_{r=1}^{n} \lceil 2^{d_r}/c \rceil 2^{c+d}) \subseteq O((2^{c+2d}/c)n)$ time and bits to build the tables. Note that the superexponential growth is now controlled using c. In exchange, we spend $O(\sum_{r=1}^{n} \lceil 2^{d_r}/c \rceil \lceil 2^d/w \rceil) \subseteq O(2^{2d}/(cw)\, n)$ time per recursion step, a slowdown factor of $O(2^d/c)$.

6.3.2 Writing the output. We generate the output of the intersection directly in the form of a k^d-tree T. Because we traverse the n quadtrees Q_r^* in depth-first and left-to-right form, we can append the 2^d bits of each new node of level v to the corresponding bitvector $T.B_v$ of the output.

We maintain one cursor per output bitvector $T.B_v$. Once we compute the bitvector e^*, we recurse on the 1s of e^* as explained. For each $e^*[j] = 1$, we recursively enter into the j-th children of $x_1, \ldots, x_n$. If that recursive call returns that the subgrid is empty, we set $e^*[j] = 0$. If, after all the recursive calls to children j, the bitvector is $e^* = 0$, we return to the recursion parent indicating that the whole subgrid is empty, and do not modify $T.B_v$. Otherwise, we append e^* to $T.B_v$, increase its cursor by 2^d positions, and return to the recursion parent indicating that the subgrid is nonempty.

In this way, although we can work more than what is needed to produce the output T, we do not allocate more space than what is needed to represent T, because we only write the bits when we know that the subgrid is nonempty. The recursion stack is of maximum depth $\log \ell$, so it requires at most $O(2^d n \log \ell)$ bits of space.

Once T is built, if we want to further operate on it, we must (1) prepare its bitvectors $T.B_v$ for fast rank operations in $O(|T|/w)$ time, and (2) convert it into a qdag $\mathcal{Q} = (T, \mathsf{Id})$, Id being the identity mapping on d attributes (Def. 3.3), in $O(2^d)$ time.

6.4 Some special cases

Our implementation also handles some specific extensions in order to support the queries in the chosen benchmarks; we describe those in this section.

6.4.1 Simple selections. Our implementation supports simple selections $\sigma_\theta(L)$, in particular the following generalization of predicate $\theta \equiv (A_i = a)$ discussed in Section 5.1. For an attribute A_i and a set of constant values V, we support predicates of the form $\theta \equiv (A_i \in V)$, which can be expressed using lqdag notation as

$$pred(A_i \in V, \mathcal{A}) = (\mathtt{EXTEND}, (\mathtt{QTREE}, \langle V \rangle), \mathcal{A} \setminus \{A_i\}),$$

where $\mathcal{A}$ is the attribute set of subexpression L and $\langle V \rangle$ is the quadtree in one dimension with $|V|$ cells and values in V. Our implementation uses, however, qdags instead of lqdags. Thus, we build a compressed quadtree Q_V for V, to then form the corresponding qdag $\mathcal{Q}_V = (Q_V, \mathsf{Id})$ by just adding the identity mapping Id. Then, we extend it to $\mathcal{Q}_V^* = \textsc{Extend}(\mathcal{Q}_V, \mathcal{A} \setminus \{A_i\})$ using Algorithm 3, to finally use Algorithm 5 to compute the AND of Eq. (3), which will yield the desired selection. The construction of $pred(A_i \in V, \mathcal{A})$ then takes time $O(|V| \log \ell + 2^{|\mathcal{A}|})$.

6.4.2 Self-joins. Graph patterns usually involve several self-joins. Our framework can be extended to efficiently compute that kind of queries without any overhead.

Consider, for example, a graph whose edges are represented as one relation $R(S, O)$, and a query looking to retrieve all tuples (a, b, c) of elements forming a triangle in the graph. In order to express this query in relational algebra, we create three different renamings of R: $R_1(A, B)$, $R_2(B, C)$ and $R_3(C, A)$. Then, the triangle query is expressed as $J = R_1(A, B) \bowtie R_2(B, C) \bowtie R_3(C, A)$. We can do this without actually creating any physical copy of R, relying instead on mappings. More precisely, we intersect three qdags: (Q_R, M_{AB}), (Q_R, M_{BC}), and (Q_R, M_{CA}), where Q_R is the quadtree representing $R(A, B)$ and the mappings extend it to three dimensions (A, B, C) in different ways:[8]

$$M_{AB} = [0, 0, 1, 1, 2, 2, 3, 3],\ M_{BC} = [0, 1, 2, 3, 0, 1, 2, 3],\ M_{CA} = [0, 2, 0, 2, 1, 3, 1, 3].$$

By Theorem 3.9, this strategy is worst-case-optimal, as the running time matches $2^{\rho^*(J,D)}$, the worst-case size of the triangle query when we regard the join as operating on different relations $R_1(A, B)$, $R_2(B, C)$, and $R_3(C, A)$. This bound may overestimate the actual worst-case output size of a self-join, though in the case of the triangle query, they differ only by a constant factor that depends on the query [15] (and this is the best known bound for queries with self-joins).

[8]The mappings are obtained by listing the identifiers of the octants $a_0b_0c_0$, $a_0b_0c_1$, $a_0b_1c_0$, $a_0b_1c_1$, $a_1b_0c_0$, $a_1b_0c_1$, $a_1b_1c_0$, $a_1b_1c_1$, where the identifiers are 0, 1, 2, or 3 depending on the quadrant of R referred by the octant, for example in $R_2(B, C)$ the quadrants are $0 = b_0c_0$, $1 = b_0c_1$, $2 = b_1c_0$, $3 = b_1c_1$ and thus the sequence is as shown in M_{BC}.

6.5 A parallel version

Algorithm 5 is easily parallelizable, because it traverses the independent subgrids of the output. As a proof of concept, we explore a simple way to parallelize its computation: given p processors available, we find the first level ℓ' such that Q^+ has at least p nonempty nodes, and from there an independent thread computes the points in the subgrid of each nonempty node at that level.

Note that, because of our mechanism to lift the dimension of the input quadtrees, many sub-trees of the extended qdags share the same subtrees of the input quadtrees, and then those may be accessed simultaneously by various processes when running the intersection. The corresponding parts of the output relation, instead, are written independently by each process. At the end, their output bitvectors $T.B_v$ must be concatenated for each v, to form the final result.

This strategy does not ensure a theoretical speedup, because all the nodes in Q^+ may be concentrated in one of the subgrids; a more robust parallelization is a matter of future work. Even so, in practice this simple technique produced a significant speedup, as we show in Section 7.

7 EXPERIMENTAL RESULTS

In this section we report the results of the experimental comparison of our implementation of qdags with various other prototypes and systems, both in terms of index space and query time.

All our experiments ran on an Intel(R) Xeon(R) CPU E5-2630 at 2.30GHz, with 6 cores and 12 hyperthreads, 15 MB of cache, and 96 GB of RAM. Our source codes were compiled using g++ with flags `-std=c++11`, `-O3`, and `-msse4.2`. Our parallel versions then use $p = 12$ threads.

7.1 The Wikidata SPARQL Benchmark

We test first the Wikidata Graph Pattern Benchmark (WGPB) introduced by Hogan et al. [16]. This corresponds to a Wikidata [42] sub-graph with 81,426,573 RDF triples (subject,predicate,object) featuring 2,101 different predicates. We store the triples as binary relations, according to the predicates: for each triple (s, p, o), the pair (s, o) is stored in the binary relation corresponding to predicate p. This benchmark provides 17 query patterns of different widths and shapes, including acyclic and cyclic queries, as shown in Figure 14. Each pattern is instantiated with 50 different random queries[9] involving random Wikidata predicates, so that the query results are nonempty. We execute the 850 resulting queries in random order.

We compare our Qdags with the following prototypes and database systems:

EmptyHeaded: An implementation [1] of NPRR [33]. Each predicate is stored as a separate relation. For each, EmtpyHeaded creates two different tries, which must be maintained in main memory during query evaluation.

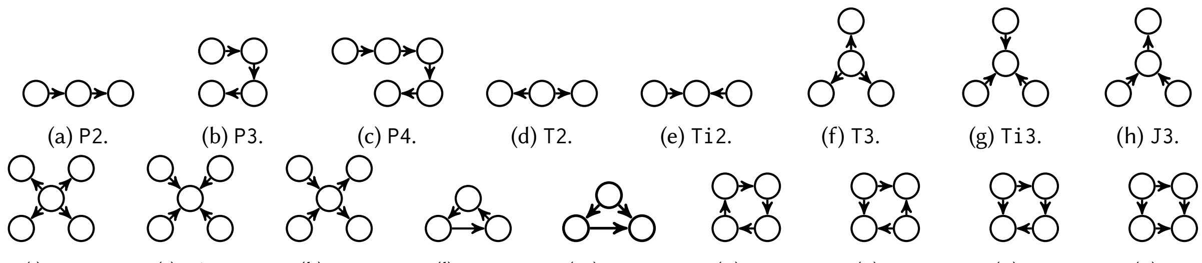
Apache Jena: A reference implementation of the SPARQL standard. We use TDB, with B+-trees indexes in three orders: *spo*, *pos*, and *osp*. For a fair comparison, and since predicates are constant in our queries, we disregard the *osp* order.

Jena LTJ: An implementation [16] of the wco leapfrog trie join algorithm (LTJ) on top of Jena TDB. All six different orders are indexed in B+-trees, though we again disregard the space usage of orders *osp* and *sop*.

RDF-3X: The reference scheme [30] that indexes a single table of triples in a compressed clustered B+-tree. The triples are sorted and those in each B+-tree leaf are differentially encoded.

Virtuoso: A widely used graph database hosting the public DBpedia endpoint, among others [11]. It provides a column-wise index of quads with an additional graph (g) attribute, with two full orders (*psog*, *posg*) and three partial indexes (*so*, *op*, *gs*) optimized for patterns with constant predicates (like the ones in our queries).

[9]https://github.com/GQgH5wFgzT/benchmark-leapfrog/tree/gh-pages/benchmark/queries/bgps



P2	P3	P4	T2	Ti2	T3	Ti3	J3	
353.60	114.50	111.34	175.50	236.92	29.60	127.74	142.18	
T4	Ti4	J4	Tr1	Tr2	S1	S2	S3	S4
63.08	81.88	110.00	11.62	14.90	10.60	28.32	38.34	4.50

Fig. 14. The 17 query patterns for the Wikidata Graph Pattern Benchmark and, below, their average number of results per query.

Blazegraph: The graph database system [40], hosting the official Wikidata Query Service [23]. We run the system in triples mode wherein B+-trees index three orders: *spo*, *pos*, and *osp*.

7.1.1 Sorting out the database. Since our index assumes that the attribute values are integers in a range $[0, \ell - 1]$, we must assign integer identifiers to the distinct strings appearing in the triples.

We can do this assignment in any convenient way. As already mentioned in Section 3.4, the clustering of the points across the underlying hypercube is crucial for the efficiency of quadtrees, both in terms of space usage and of running time. An offline optimization we attempt here is to assign the integer identifiers so as to improve the clustering of the resulting points within the hypercube. Similar strategies have been used on binary matrices [17], graphs [1], and inverted indexes [5]. In the particular case of Wikidata triples, we enumerate the subject s and object o of each RDF triple (s, p, o) of Wikidata, thus generating the 2-dimensional points that are further indexed using qdags. We try the following strategies for enumerating subjects and objects:

Original Order (Qdag Original)**:** Subjects and objects are enumerated using the original order of the triples in the Wikidata dataset.

Lexicographic Order of Predicates (Qdag Lex)**:** We first sort the triples lexicographically by predicates, and then carry out the enumeration of subjects and objects using this order. The rationale is that, if each predicate tends to connect a small subset of subjects and objects, then the resulting matrices will be clustered.

BFS Order (Qdag BFS)**:** Consider the graph where subjects and objects are the nodes, and predicates are directed edges from subjects to objects. We then number the nodes of this graph by running several BFS traversals until enumerating all the nodes. The traversals are started according to an initial lexicographic order of the nodes. The resulting node ordering is the integer assignment to subjects and objects; the aim is to give consecutive identifiers to the objects assigned to the same subjects, thereby inducing clusters.

Appendix C illustrates the effect of those node orderings on the point distribution of grids. As it can be seen, Lex and BFS yield a less scattered distribution (e.g., in BFS, points tend to lie on the diagonal of the grid). As anticipated in Section 3.4, this clustering will impact positively on the space of the index and the time performance of the multijoin queries.

Table 1. Index space, in bytes per tuple, and construction time, in elapsed minutes, on the Wikidata benchmark. Qdags and EmptyHeaded build the indexes from the numeric tuples, so we have added to those the 13.12 minutes it took us to parse the strings and convert them to integers. Further, we added to the Lex and BFS versions the time it took us to reorder the identifiers.

System (Data + Indexes)	Space	Build time
Qdag Original	6.76	18.90
Qdag Lex	4.75	17.86
Qdag BFS	4.90	25.47
Qdag BFS, non-compact	205.71	30.31
EmptyHeaded	1292.28	61.39
Jena	48.42	101.47
Jena LTJ	96.83	107.43
RDF-3X	107.65	16.41
Virtuoso	104.89	14.42
Blazegraph	99.86	58.28

7.1.2 Space usage. Table 1 shows the space usage (in bytes per tuple) for the different systems and algorithms we tried, as well as their construction time, in elapsed minutes. As it can be noted, Lex and BFS orders yield considerably improved space usage when compared to the original order. Qdag Lex and Qdag BFS use about 1/10 of the space of Jena, the one that uses the least space among the competing schemes. Recall that within this space usage we include data and indexes, which means that all the compact Qdag versions use less than the 8 bytes per tuple needed to just represent the dataset if we separate the triples by predicate and represent each pair (s, o) with two 32-bit integers. The Lex and BFS orders not only reduce the index space, but also the construction time of Qdag variants: once the integers of the triples are generated in the appropriate order, Original takes 4.26 microsecs/tuple, Lex takes 3.49, and BFS 3.67. Note this is 5–6 minutes total construction time from the sorted integer triples. The qdags are also built within space close to their final size, 5.51 bytes per tuple for Lex and 5.66 bytes per tuple for BFS.

7.1.3 Comparing Qdag variants. We first compare the different Qdag variants, testing the parallel implementation from Section 6.5, a non-compact Qdag baseline we describe in Appendix B, and the clustering approaches proposed in Section 7.1.1. Figure 15 summarizes the query times for the 17 query patterns of Figure 14.

As it can be seen, the orders Lex and BFS yield improved query time, as expected from the discussion in Section 3.4. In general, BFS outperforms Lex and Original, both in the sequential and parallel versions.[10] Parallelism yields important speedups, except where the sequential version is already very efficient and thus the overhead introduced by parallelism does not pay off. The average speedup is 3.24. Finally, the non-compact version (of which we show only the BFS variant) is not only 40 times larger than the serial compact BFS version, but also about 130 times slower on average, which shows the beneficial effects of compact representations in reducing the cache faults in main memory.

7.1.4 Comparison against others. Figure 16 compares the query times of Qdag BFS and Qdag Par BFS, the best performing variants in Section 7.1.3, with the prototypes and systems listed at the beginning of Section 7.1, on the 17 query patterns of Figure 14 (see Appendix D for the full detail). We note that EmptyHeaded, Jena, Jena

[10]A disadvantage of those convenient orders is that, in the more general setting that allows selections, predicates with inequalities make no sense if one reorders the identifiers. This has no effect on the equijoin queries we study in these experiments.

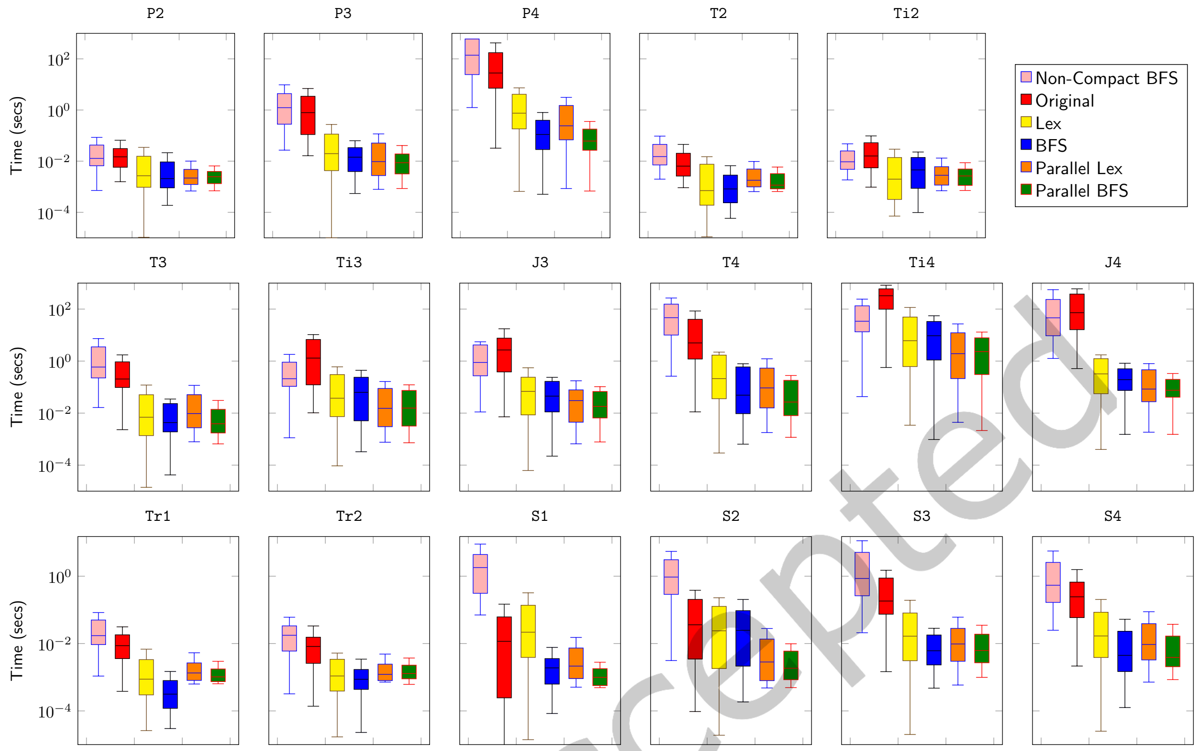


Fig. 15. Query times (in seconds and logscale) for the different variants of Qdags on the Wikidata benchmark.

LTJ, and Blazegraph use several processors, like Qdag Par BFS, while RDF-3X and Virtuoso are sequential. We observe the following facts:

- Qdags are the fastest alternative on almost all patterns with three nodes: P2, T2, Tr1, and Tr2, and is only slighlty outperformed by EmptyHeaded on Ti2, the remaining pattern with three nodes in the benchmark. This is in line with the theoretical bounds presented in Section 3: patterns with three nodes implies qdags with only three dimensions, and since the running time is exponential in the dimension, it is expected that our algorithm excels in low-dimensional queries. To put times in context with respect to the space usage, the next-best alternative, EmptyHeaded, uses 264 times the space of Qdag BFS, as noted in Table 1. The same table shows that Jena, the next least-memory contender, uses 10 times the space of our approach. Note that Qdags do not need to load a different index order from secondary storage upon query optimization, because they have only one index.
- Let us consider next the patterns P3 and P4, representing paths. For the case of P3, a pattern with 4 nodes, Qdags are close to the best time, given by EmptyHeaded. On the other hand, neither Qdags nor EmptyHeaded are competitive for P4, but the best algorithms are Jena LTJ, the other worst-case optimal algorithm we tested against, and the standard engine of Jena. We speculate that systems based on Jena have an edge because they are able to consistently find a better variable ordering for this type of queries (see Hogan et al. [16] for a discussion on orderings). On the other hand, Qdags are not dependent on variable orderings, but as we mentioned our index becomes less competitive as the dimension of queries increases.

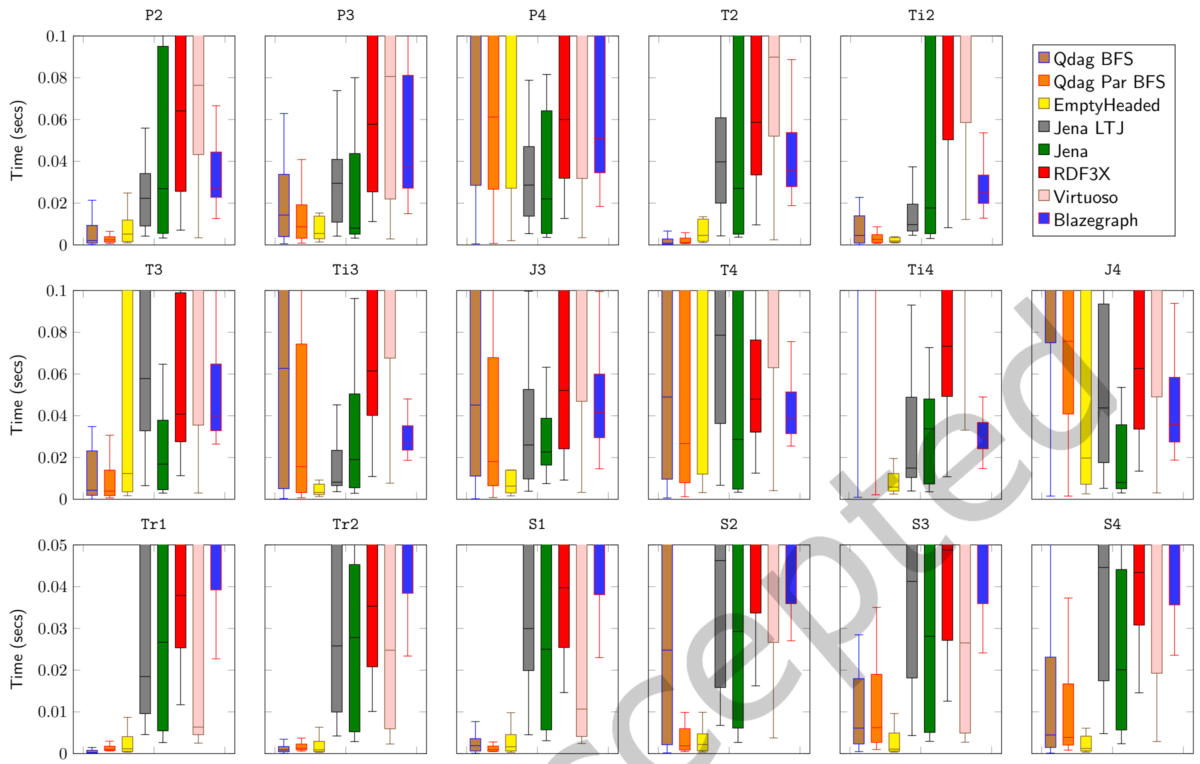


Fig. 16. Query times (in seconds) of the best Qdag variants and state-of-the-art prototypes and database systems, on the Wikidata benchmark.

- The next queries we discuss are the so-called *star* queries, with T3, Ti3 and J3 having one central node connected to three external nodes, and T4, Ti4, and J4 having a central node connected to four external nodes. The behaviour we see is consistent with the other set of queries. For queries with four nodes Qdags remain competitive, being the fastest for T3, but outperformed for Ti3 and J3. Interestingly, EmptyHeaded, the usual contender and the best alternative for Ti3 and J3, does not perform well on T3. On the other hand, Qdags do not fare well on the star queries with 5 nodes. It turns out that, in these types of star queries, worst case algorithms in general are outperformed by traditional graph database systems, although EmptyHeaded is still the best alternative for Ti4.[11]
- Finally, we discuss the square queries S1, S2, S3, and S4. Once again, worst-case optimal algorithms take the lead, with Qdags and EmptyHeaded earning the best time in two queries each. Note the large difference between the non-parallel and parallel versions in S2. Upon inspecting the intermediate results of these queries, we found that they were demanding Qdags to focus on poorly-clustered parts of the graph, and therefore parallelism was more important since more distant portions of the qdags needed to be analyzed.

[11]Since the benchmark is constructed out of random walks, the probability of finding a star query with 3 or more leaves in which all joins are equally selective is quite low. Thus, we conjecture that traditional systems are able to achieve good times simply by a careful planning of the join order, beating wco algorithms and Yannakakis, which may end up doing more excess work if a bad order is chosen.

One can notice how BFS do not take advantage of clustering when processing queries in S2 by looking again at Figure 15: the times for Original and BFS are virtually the same.

7.1.5 String identifiers. Our implementation of Qdags works directly on datasets and queries where the identifiers are already mapped to integers. This can be seen as unfair to the database systems managing strings, because reporting strings may induce additional time overhead and their space can be significant compared to the integer triples. In our Wikidata subgraph, for example, the strings occupy 1,427 MB, nearly 150% of the 932 MB used by the integer triples in plain form. Blazegraph, Jena, Jena-LTJ, RDF-3X, and Virtuoso include this overhead, while EmptyHeaded and Qdags do not.

We can largely diminish the impact of the strings by storing them in succinct dictionaries [24], which map from strings to integers (to translate the queries) and back (to translate the solutions) in a few microseconds per string. For example, using the variant `HTFC-rp` with sampling 64 [24], the strings in our benchmark are compressed to 17%, or 235 MB (3 additional bytes per triple) and an identifier is translated to its string in about 3 microseconds. The total impact of translating the identifiers back to strings adds at most 0.003 seconds in Figure 16.

7.2 Benchmark on SNAP Graphs

Our second test uses the benchmark by Nguyen et al. [35], which includes several patterns on SNAP graphs [22]. The benchmark includes both cyclic and acyclic queries, and allows one to regulate the selectivity of each pattern. The detailed experimental setup and results are given in Appendix E. Table 2 summarizes the results by showing, for each kind of query and each competing implementation, the proportion of queries where Qdags perform better or worse than the competitor, as follows:

- Qdag << X: Qdags are 10 times, or more, faster than the other, or only the other gives time-out or out-of-memory.
- Qdag < X: Qdags are more than twice, but less than 10 times, faster than the other.
- Qdag = X: both times are within a factor of two of each other.
- Qdag > X: Qdags are more than twice, but less than 10 times, slower than the other.
- Qdags >> X: Qdags are 10 times, or more, slower than the other, or only Qdags give time-out.

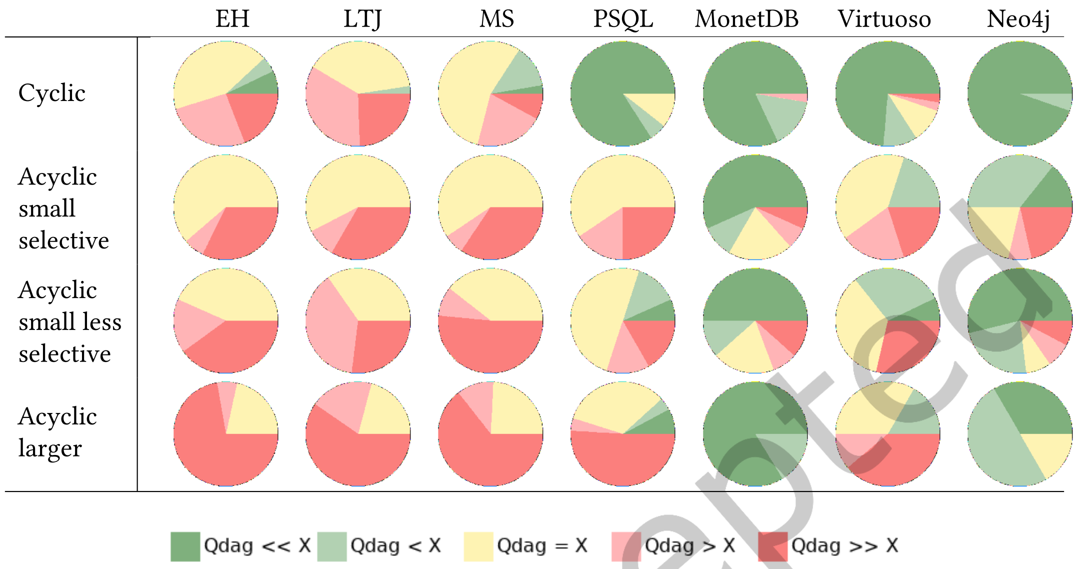
For the case where a given time is zero, we round it up to 1 in order to consider proportions. The cases where both indexes give time-out or out-of-memory are disregarded.

We separate the queries into cyclic, acyclic on small graphs (with more and less selectivity), and acyclic on large graphs (here, selectivity did not make an important difference). We can see that:

- In general, Qdags are outperformed by wco schemes like EmptyHeaded (EH), LTJ, and MS, and tend to outperform those that are not: MonetDB, Virtuoso, and Neo4j. The situation is mixed for PSQL.
- On cyclic queries, where being wco is most relevant, Qdags perform similarly to the other wco approaches in about half of the cases, and sharply outperform the non-wco schemes in most queries.
- On acyclic queries, Qdags perform similarly to the other wco approaches in half to a quarter of the queries, worsening as the output size increases. Qdags still outperform MonetDB and Neo4j most of the time.

We remind that Qdags obtain these mixed results using one or two orders of magnitude less space than the competing schemes (in particular, they use 6–200 times less space than EmptyHeaded). This can make the Qdag attractive, for example, as a companion index to non-wco systems where, for a very modest increase in space, a Qdag may speed up cyclic queries considerably. Qdags can also be a relevant solution when the extra space of other wco data structures cannot be afforded.

Table 2. Summary of the results on querying the SNAP benchmark, depending on the fraction of queries where Qdags performed much better, better, similarly, worse, or much worse than each other index.



8 CONCLUSIONS

We have introduced the first index for multijoin queries that is simultaneously time-optimal and nearly space-optimal. More precisely, it achieves worst-case-optimal time in data complexity while storing only $2 + o(1)$ extra words per tuple. Our index regards relations on d attributes as point sets in a d-dimensional hypercube, which are represented with compressed quadtrees. The join algorithm uses a new structure we dub qdag, which simulates a dimensionality-lifted quadtree, and then generates the result in the form of a compressed quadtree by virtually traversing the output space. We prove that such a simple traversal does reach the AGM bound. A lazy version of qdags, dubbed lqdags, support the full relational algebra, and retain for Boolean queries an extended notion of worst-case optimality, while requiring no space for intermediate results.

The evaluation of join queries using qdags provides a competitive alternative to current worst-case optimal algorithms [18, 20, 32, 33, 41]. Regarding space, qdags require only a few extra words per tuple in the worst case, and in practice they even manage to compress the database representation, as shown in our experiments. This is generally much less than what standard database indexes require, and definitely less than the space required by current worst-case optimal algorithms (e.g., [18, 33, 41]). Moreover, in both NPRR [33] and leapfrog [41], the required index structure only works for a specific ordering of the attributes. Thus, in order to efficiently evaluate any possible query using these two algorithms, a separate index is required for every possible attribute order. In contrast, all we need to store is one quadtree per relation, and that works for any query. In our experiments, which include only binary relations (i.e., alternative indexes need to store just two attribute orders), the least space-consuming index, Apache Jena, uses 10 times more space than our Qdag index, whereas EmptyHeaded uses about 250 times more space.

Time complexity. Regarding time, the first comparison that stands aside is the $\log N$ factor, present in our solution (as well as in, e.g., Hogan et al. [16]) but not in others like NPRR [33] and leapfrog [41]. Note, however, that NPRR assumes to be able to compute a join of two relations R and S in time $O(|R| + |S| + |R \bowtie S|)$, which is only possible when using a hash table and when time is computed in an amortized way or in expectation [33, footnote 3]. This was also noted for leapfrog [41, Section 5], where they state that their own $\log N$ factor can be avoided by using hashes instead of tries, but they leave open whether this is actually better in practice. More involved algorithms such as Panda [20] build upon algorithms to compute joins of two relations, and therefore the same $\log N$ factor appears if one avoids hashes or amortized running time bounds.

Our algorithm also incurs in an additional 2^d factor in time when compared to NPRR or leapfrog, similarly to other worst-case optimal solutions based on geometric data structures [18, 32]. While this factor does not depend on the data, it is relevant in practice, as shown in our experiments: Qdags excel in queries yielding relations of 3 attributes and fare well on 4, but cease being competitive on 5 or more attributes. This slowdown, the price we pay for using so little space, is partly compensated by the fact that our indexes are compressed, and thus might fit in faster memory: our experiments show that a non-compact version of our index is 40 times larger and 2–1000 times slower.

Usage scenarios. One important benefit of our framework is that the answers to queries can be built directly in their compressed representation. As such, we can iterate over them, or store them, or use them as materialized views, either built eagerly as quadtrees, or in lazy form as lqdags, or in partially materialized form with bounded delay, as incomplete quadtrees.

Aside from their standalone use, one could take advantage of the low storage cost of quadtrees and add them as a companion to a more traditional database setting. Simple queries could be handled by the database, while multijoins could be processed faster over the quadtrees.

Using a simple proof-of-concept, we also show that our multijoin algorithm is easily parallelizable to obtain, without sophisticated techniques, average speedups over 3 with 6 cores. This opens the door to a deeper study of quadtrees in the setting of parallel computation (see, e.g., Suciu [39]).

Dynamism. We have only discussed the static scenario in this article, where tuples cannot be added or removed from the relations. In order to enable dynamism, we only need to replace the quadtree representation of Lemma 2.1 by a data structure that supports insertions and deletions of points. The qdags and lqdags then stay automatically up to date upon changes on their underlying quadtrees. A dynamic quadtree can be obtained via its k^d-tree representation described in Section 6.1. For the dynamic case, we rather represent the bitvector V, which is of length $|V| \leq 2^d p \log \ell$ and has $v \leq p \log \ell$ 1s, as the sequence of v differences $x_j = p_j - p_{j-1}$ between the consecutive positions $p_1, \ldots, p_v$ of the 1s, with $p_0 = 0$. Operation $\mathsf{rank}(V, i)$ is then equivalent to operation $\mathsf{search}(j) = \max\{j, p_j \leq i\} = \max\{j, \sum_{t=1}^{j} x_t \leq i\}$, and $V[j] = \mathsf{search}(j) - \mathsf{search}(j-1)$. A dynamic representation of partial sums [29, Lem. 1.4] using, say, δ-codes [38] to encode the values x_j, takes $dp \log \ell(1 + o(1))$ bits of space and implements search in time $O(\log |V| / \log\log |V|) \subseteq o(d + \log p + \log\log \ell)$. Within the same time, it can insert and remove values x_i, which suffices to emulate insertion and deletion of points in the k^d-tree (using $O(\log \ell)$ such operations) [8]. Therefore, we can insert or delete tuples in time $o((d + \log p + \log\log \ell) \log \ell)$. The price is an $o(d + \log p + \log\log \ell)$ slowdown factor with respect to the times given in Lemma 2.1, Theorem 3.9 (where the times are still in $\tilde{O}(2^{\rho^*(J,D)})$), and Theorem 4.5 (where the times are still in $\tilde{O}(F(D)^*)$).

Beyond wco. Finally, an important direction for future work is to go beyond worst-case optimality. EmptyHeaded [1] may outperform the AGM bound for multijoin queries and approach their fractional hypertreewidth. It structures the query graph, which is cyclic in general, as a tree where each node is a cyclic subquery. Each node is then solved within the AGM bound, and the resulting materialized relations are finally processed with Yannakakis' algorithm. While, as discussed in Section 5.2.1, a direct application of lqdags would not obtain the same complexity, we can use qdags to process each tree node in worst-case optimal time and materialize the resulting relation, so as to apply Yannakakis' algorithm on those. A clear advantage of using our representation is that those intermediate materialized relations, which can be large, are represented in compact form as quadtrees, thereby making this algorithm much more practical. Further, the qdags would typically have to produce relations of lower dimension, a case where they perform significantly better. The Panda algorithm [20] takes this further, and can process queries in time bounded by the submodular width bound. Again, an interesting direction for future work is to understand how to use Qdags in combination with Panda.

Epilogue. We continued pursuing the line initiated in this article, about compact data structures able to implement wco joins. After the conference version of this paper, we developed with other coauthors a new compact data structure called the *ring* [4], which supports wco joins for the specific case of labelled graphs (or, equivalently, a relation formed by triples). The ring is not compositional: the outcome of a multijoin query is not anymore a ring. In exchange, it offers more stable times than qdags, though qdags still outperform the ring in queries with few nodes. The qdags we implement here are more space-efficient than the most compressed ring representation.

What is most interesting is how both structures evolve as the dimension d of the relations grow. Our qdags always need one copy of the database, though their query time grows as $O(2^d)$. The ring, instead, requires one copy only in the particular case of $d = 3$. Multidimensional rings, though not yet implemented, are shown to require $\Omega(2^d d^{-1/2})$ copies of the database, though in exchange their query time grows as $O(d^2)$. Space translates into update time in the dynamic scenario, however: the time to insert/delete tuples in qdags grows as $O(d)$, as we have seen, whereas it is $\Omega(2^d d^{1/2})$ on rings. A very interesting question is how to develop compact data structures that can more gracefully trade space for time in this exponential dependence on d, or that can eliminate it completely.

ACKNOWLEDGMENTS

We thank Aidan Hogan for his help with the construction time of alternative indexes, Matilde Rivas and Fabrizio Barisione for spotting mistakes in two figures, and the reviewers for their insightful comments. This work was funded by ANID – Millennium Science Initiative Program – Code ICN17_002 (all authors) and Fondecyt Grant 1-200038 (second author).

REFERENCES

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

[2] N. Alon and M. Naor. 1996. Derandomization, witnesses for Boolean matrix multiplication and construction of perfect hash functions. *Algorithmica* 16 (1996), 434–449.

[3] S. Álvarez-García, N. Brisaboa, J. Fernández, M. Martínez-Prieto, and G. Navarro. 2015. Compressed vertical partitioning for efficient RDF management. *Knowledge and Information Systems* 44, 2 (2015), 439–474.

[4] D. Arroyuelo, A. Hogan, G. Navarro, J. Reutter, J. Rojas-Ledesma, and A. Soto. 2021. Worst-case optimal graph joins in almost no space. In *Proc. 47th ACM International Conference on Management of Data (SIGMOD)*. 102–114.

[5] D. Arroyuelo, M. Oyarzún, S. González, and V. Sepúlveda. 2018. Hybrid compression of inverted lists for reordered document collections. *Information Processing Management* 54, 6 (2018), 1308–1324.

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

[7] D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. 2005. Representing trees of higher degree. *Algorithmica* 43, 4 (2005), 275–292.

[8] N. Brisaboa, A. Cerdeira-Pena, G. de Bernardo, and G. Navarro. 2017. Compressed representation of dynamic binary relations with applications. *Information Systems* 69 (2017), 106–123.

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

[10] S. Deep and P. Koutris. 2018. Compressed representations of conjunctive query results. In *Proc. 37th ACM Symposium on Principles of Database Systems (PODS)*. 307–322.

[11] O. Erling and I. Mikhailov. 2009. RDF support in the Virtuoso DBMS. In *Networked Knowledge – Networked Media*. Springer.

[12] R. A. Finkel and J. L. Bentley. 1974. Quad Trees: A data structure for retrieval on composite keys. *Acta Informatica* 4 (1974), 1–9.

[13] T. Gagie, J. González-Nova, S. Ladra, G. Navarro, and D. Seco. 2015. Faster compressed quadtrees. In *Proc. 25th Data Compression Conference (DCC)*. 93–102.

[14] S. Gog and M. Petri. 2014. Optimized succinct data structures for massive data. *Software: Practrice and Experience* 44, 11 (2014), 1287–1314.

[15] G. Gottlob, S. T. Lee, G. Valiant, and P. Valiant. 2012. Size and treewidth bounds for conjunctive queries. *Journal of the ACM* 59, 3 (2012), 16.

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

[17] D. S. Johnson, S. Krishnan, J. Chhugani, S. Kumar, and S. Venkatasubramanian. 2004. Compressing large Boolean matrices using reordering techniques. In *Proc. 30th International Conference on Very Large Data Bases (VLDB)*. 13–23.

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

[19] M. A. Khamis, H. Q. Ngo, and A. Rudra. 2016. FAQ: Questions asked frequently. In *Proc. 35th ACM Symposium on Principles of Database Systems (PODS)*. 13–28.

[20] M. A. Khamis, H. Q. Ngo, and D. Suciu. 2017. What do Shannon-type Inequalities, Submodular Width, and Disjunctive Datalog have to do with one another?. In *Proc. 36th ACM Symposium on Principles of Database Systems (PODS)*. 429–444.

[21] D. E. Knuth. 2009. *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams* (12th ed.). Addison-Wesley Professional.

[22] J. Leskovec and A. Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

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

[24] M. A. Martínez-Prieto, N. Brisaboa, R. Cánovas, F. Claude, and G. Navarro. 2016. Practical compressed string dictionaries. *Information Systems* 56 (2016), 73–108.

[25] G. M. Morton. 1966. *A computer oriented geodetic data base; and a new technique in file sequencing*. Technical Report. IBM Ltd.

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

[27] G. Navarro. 2016. *Compact Data Structures – A practical approach*. Cambridge University Press.

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

[29] G. Navarro and K. Sadakane. 2014. Fully-functional static and dynamic succinct trees. *ACM Transactions on Algorithms* 10, 3 (2014), article 16.

[30] T. Neumann and G. Weikum. 2010. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal* 19 (2010), 91–113.

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

[32] H. Q. Ngo, D. T. Nguyen, C. Re, and A. Rudra. 2014. Beyond worst-case analysis for joins with Minesweeper. In *Proc. 33rd ACM Symposium on Principles of Database Systems (PODS)*. 234–245.

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

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

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

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

[37] H. Samet. 2006. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann.

[38] D. Solomon. 2007. *Variable-Length Codes for Data Compression*. Springer-Verlag.

[39] D. Suciu. 2017. Communication cost in parallel query evaluation: A tutorial. In *Proc. 36th ACM Symposium on Principles of Database Systems (PODS)*. 319–319.

[40] B. B. Thompson, M. Personick, and M. Cutcher. 2014. The Bigdata®RDF Graph Database. In *Linked Data Management*. 193–237.

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

[42] D. Vrandecic and M. Krötzsch. 2014. Wikidata: a free collaborative knowledgebase. *Communications of the ACM* 57, 10 (2014), 78–85.

[43] D. S. Wise and J. Franco. 1990. Costs of quadtree representation of nondense matrices. *Journal of Parallel and Distributed Computing* 9, 3 (1990), 282–296.

[44] M. Yannakakis. 1981. Algorithms for acyclic database schemes. In *Proc. 7th International Conference on Very Large Databases (VLDB)*. 82–94.

A APPENDIX: COMPARISON WITH THE CONFERENCE VERSION

This article is the extended version of the ICDT'20 paper "Optimal Joins Using Compact Data Structures" [28]. We have revised and extended the original paper as summarized below:

- We implemented a prototype of our index, and compared it experimentally with state-of-the-art alternatives. We added a new section on the choices we made during the implementation, and on the variants of our index that were implemented. We also added a section on the experimental analysis of these variants, and more importantly on the comparison between our solution and other state-of-the-art systems on two different benchmarks.
- We improved, proved theoretically, and validated in practice, the bounds on the space and running time of our solution on clustered datasets. These results were added as an extension to the section on multi-join queries.
- We refined the definition of worst-case optimality for formulas composed of join, union and complement (JUC) operations so that upper bounds for formulas containing complement operations are more meaningful. As a consequence, we had to revise and update the proof of the worst-case optimality of lazy qdags for JUC queries so that it remained valid. Our main result on JUC queries is now stronger than in its conference version.
- We extended our framework of lazy qdags (lqdags) to evaluate more expressive queries from relational algebra. We added a new section showing how to extend lqdags to allow renaming the attributes of a relation, and to support selection and projection operations. By combining these three with the join, union, and complement operations, lqdags can now evaluate any formula of the relational algebra. Although for these extended formulas we can no longer guarantee worst-case optimality, we introduce a partial materialization scheme that subsumes and extends results from Deep and Koutris regarding compressed representation of query results with enumeration delay guarantees.
- Finally, we improved the completeness and readability of the paper by adding every non-obvious full proof to theorems and lemmas of the conference paper, and including new figures and examples to illustrate the description of our solution.

B APPENDIX: A NON-COMPACT QDAG BASELINE

Using a compact representation of quadtrees increases the constants in the running time of accessing the value of a node, or its children: while in a non-compact representation these can be stored directly within the node, the compact representation resorts to compact data structures to compute them. Using a compact representation, however, also increases the probability of having the memory required by the CPU to perform an operation already in the processor's cache. Given that cache memory is considerably faster than main memory, it should mitigate or eliminate the effect of higher constants in accessing the value or children of a compact quadtree node. To investigate this trade-off, we also implemented a non-compact version of our index. We then compared the running times of join queries over these two different representations, changing only the storage scheme of the quadtrees, and keeping exactly the same algorithm.

The non-compact quadtree representation consists of just two arrays, F and C. Each node of the tree is identified by an id, the root's being 0. For a node x with id i, $F[i]$ stores the id of its first child, while the id of the j-th non-empty child of x is $F[i] + j$. Moreover, the k-th least significant bit of $C[i]$ is 1 iff the k-th child of x represents a non-empty subgrid. Each value of F is a 64-bit integer; the length of each value of C depends on the number of attributes in the relation being represented by the quadtree (e.g., for two and three attributes with use 8-bit integers). Thus, for a quadtree with N nodes this representation uses at least $(64 + 8)N$ bits. For two-dimensional quadtrees, for instance, this wastes at least 4 bits for each node because the smallest basic type in the language we used (C++) is the byte. Moreover, if the relation is small, using 64 bits to identify each node also incurs waste.

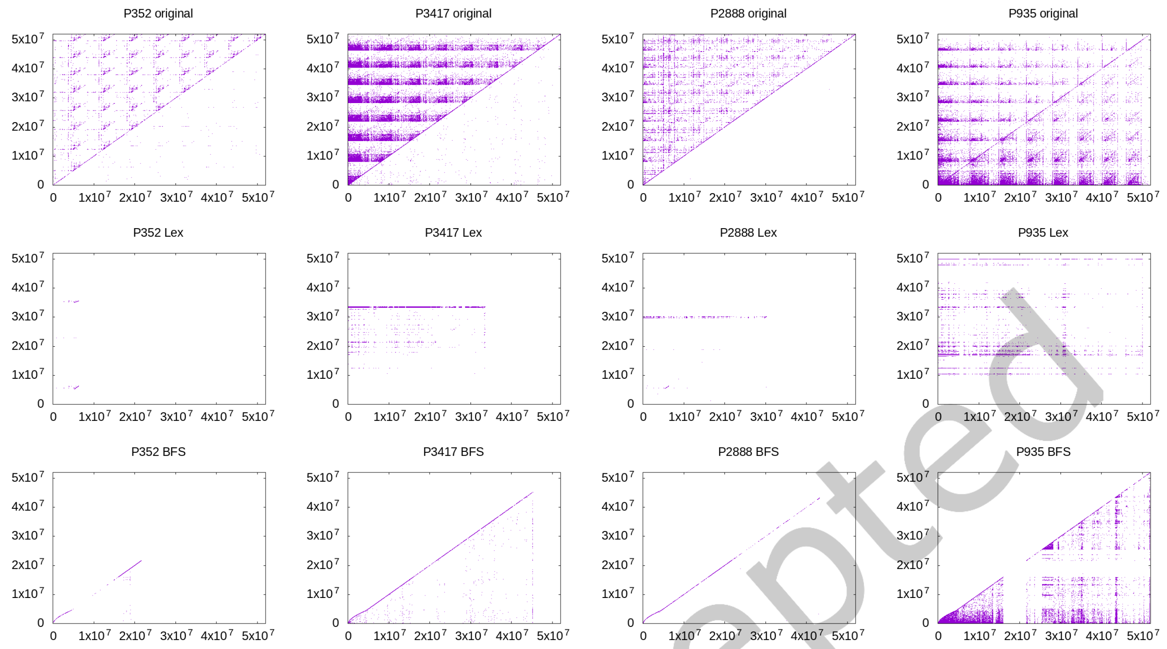


Fig. 17. Point distribution of the grids corresponding to predicates P352, P3417, P2888, and P935 of the Wikidata dataset. The figure shows the original order (top), lexicographic order (middle), and BFS order (bottom). The images corresponding to the distributions with high clusterization seem to have fewer points because multiple ones are rendered in the same pixel due to lack of resolution.

C APPENDIX: EFFECT OF NODE ORDERING ON THE POINT DISTRIBUTIONS

Figure 17 shows the point distribution of the grids corresponding to predicates P352, P3417, P2888, and P935 of the Wikidata dataset[12], using the original order (top), lexicographic order (middle), and BFS order (bottom). These relations correspond to the join P352(A, B) ⋈ P3417(A, C) ⋈ P2888(D, B) ⋈ P935(D, C) from query pattern S2. For this pattern, this particular query is the one that yields the maximum relative difference between Original and BFS order.

D APPENDIX: RESULTS ON WIKIDATA WITH COMPLETE TIME INTERVAL

Figure 18 shows the complete time intervals for the experimental results of Figure 16, on the Wikidata benchmark.

E APPENDIX: FULL DETAILS ON THE SNAP BENCHMARK

We describe the setup and detailed results on the benchmark by Nguyen et al. [35], which includes several patterns on SNAP graphs [22].

[12]Available at https://www.wikidata.org/wiki/Property:X, where $X \in \{$P352, P3417, P2888, P935$\}$.

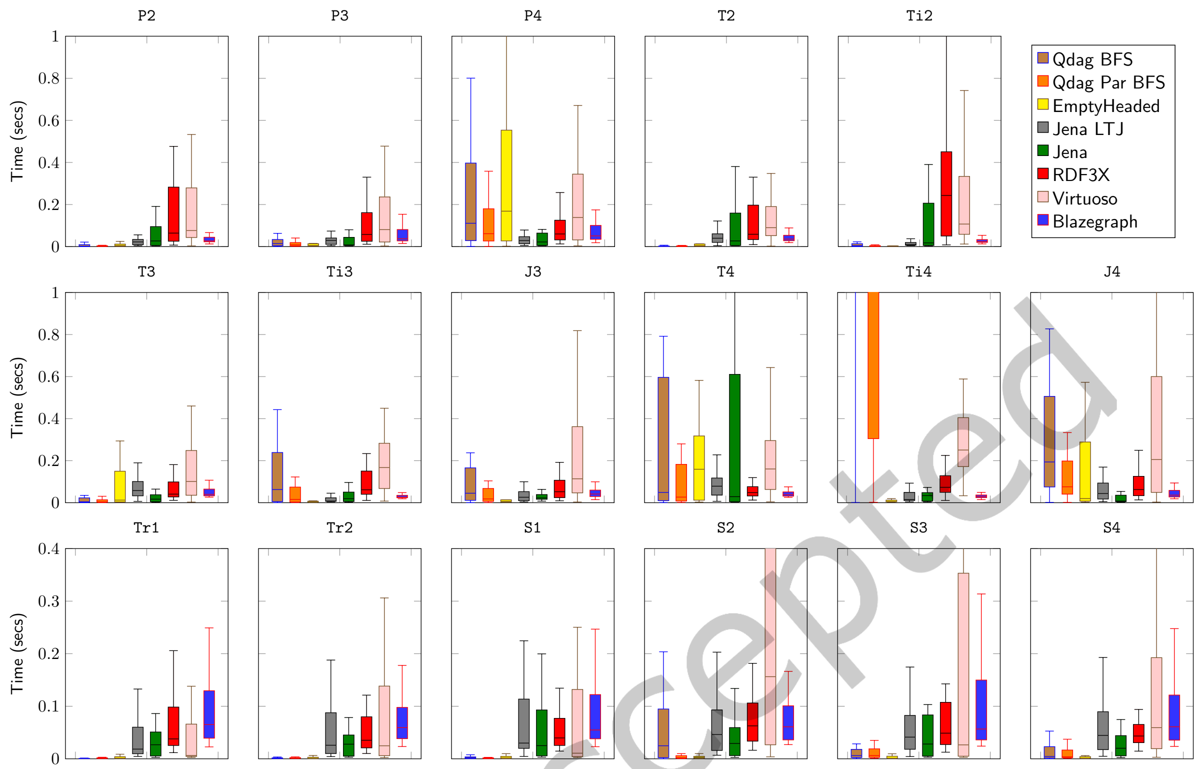


Fig. 18. Query times (in seconds) of the best qdag variants and state-of-the-art prototypes and database systems, on the Wikidata benchmark.

E.1 Experimental setup

Table 3 shows the graphs tested, along with the number of nodes, edges, and triangles of each graph, and the space used by Qdags BFS. According to the number of nodes, the graphs are classified as small (top ones in the table), medium-size (middle), and large (bottom) [35].

We run only Qdags Par BFS and EmptyHeaded in this comparison. Let edge denote the binary relation representing the graph edges. Note then that all the queries will be self-joins, which we handle as described in Section 6.4.2.

For this benchmark, we test the following cyclic queries:

3-Clique (Triangle): $\mathsf{edge}(A,B) \bowtie \mathsf{edge}(B,C) \bowtie \mathsf{edge}(C,A)$;
4-Clique: $\mathsf{edge}(A,B) \bowtie \mathsf{edge}(B,C) \bowtie \mathsf{edge}(C,D) \bowtie \mathsf{edge}(D,A) \bowtie \mathsf{edge}(A,C) \bowtie \mathsf{edge}(B,D)$;
4-Cycle: $\mathsf{edge}(A,B) \bowtie \mathsf{edge}(B,C) \bowtie \mathsf{edge}(C,D) \bowtie \mathsf{edge}(D,A)$.

For queries 3-Clique and 4-Clique, graphs are regarded as undirected.

We also consider the following acyclic queries:

3-Path: $\mathsf{V}_1(A) \bowtie \mathsf{edge}(A,B) \bowtie \mathsf{edge}(B,C) \bowtie \mathsf{edge}(C,D) \bowtie \mathsf{V}_1(D)$;
4-Path: $\mathsf{V}_1(A) \bowtie \mathsf{edge}(A,B) \bowtie \mathsf{edge}(B,C) \bowtie \mathsf{edge}(C,D) \bowtie \mathsf{edge}(D,E) \bowtie \mathsf{V}_2(E)$;
1-Tree: $\mathsf{V}_1(A) \bowtie \mathsf{edge}(A,B) \bowtie \mathsf{edge}(A,C) \bowtie \mathsf{V}_2(C)$;
2-comb: $\mathsf{edge}(A,B) \bowtie \mathsf{edge}(A,C) \bowtie \mathsf{edge}(B,D) \bowtie \mathsf{V}_1(C) \bowtie \mathsf{V}_2(D)$.

Table 3. Graphs used in the SNAP benchmark: small (top), medium-size (middle), and large (bottom) graphs. We also include the bytes per tuple used by Qdag BFS and EmptyHeaded (EH).

Graph	Nodes	Edges	Triangles	Space	Space EH
wiki-vote	7,115	103,689	608,389	1.68	43.48
p2p-Gnutella31	62,586	147,892	2,024	3.37	57.36
p2p-Gnutella04	10,876	39,994	934	0.29	60.53
loc-Brightkite	58,228	428,156	494,728	1.28	38.29
ego-Facebook	4,039	88,234	1,612,010	1.27	32.59
email-Enron	36,692	367,662	727,044	0.97	41.98
ca-GrQc	5,242	28,980	48,260	0.64	60.49
ca-CondMat	23,133	186,936	173,361	0.91	41.37
ego-Twitter	81,306	2,420,766	13,082,506	1.19	21.85
soc-Slashdot0902	82,168	948,464	602,592	1.65	45.34
soc-Slashdot0811	77,360	905,468	551,724	1.61	38.30
soc-Epinions1	75,879	508,837	1,624,481	1.95	42.71
soc-Pokec	1,632,803	30,622,564	32,557,458	4.42	25.72
soc-LiveJournal1	4,847,571	68,993,773	285,730,264	3.61	29.57
com-Orkut	3,072,441	117,185,083	627,584,181	3.96	23.66

Here, unary relations V_1 and V_2 represent subsets of the graph nodes. These are used to select a particular subset of nodes on which we want to query the corresponding pattern. For instance, for 3-Path, we query for all directed paths of length 3 that start in some node $a \in \mathsf{V}_1$, and end in some node $b \in \mathsf{V}_2$. This kind of query allows us to test in practice our implementation of the select operator from Section 6.4.1. By choosing the size of V_1 and V_2, we can also regulate the query selectivity. For instance, let us consider query 3-Path. For a given target selectivity s, we build $\mathsf{V}_1(A)$ by choosing every node a in $\pi_A(\mathsf{edge}(A, B))$ with probability $1/s$ [35]. We proceed similarly for $\mathsf{V}_2(D)$. After building $\mathsf{V}_1(A)$ and $\mathsf{V}_2(D)$, we proceed as explained in Section 6.4.1 to obtain the corresponding extended qdag needed to carry out the intersection. In our experiments, for small graphs we test with selectivity 8 and 80, which select (at random) 12.5% and 1.25%, respectively, of the graph nodes. For medium-size and large graphs, we use selectivity 10, 100, and 1000.

E.2 Results

Table 4 shows the times for cyclic queries, Table 5 for small acyclic queries, and Table 6 for large acyclic queries. We use a timeout of 1800 seconds and carry out count queries (i.e., we do not materialize the output; we just compute its size). We also include the times obtained by Nguyen et al. [35], as a comparison point. In particular, they compare LeapFrog Trie Join (LTJ), Minesweeper (MS), postgresql (PSQL), MonetDB, Virtuoso, and Neo4J. Their processor is similar to ours, though slightly faster: an Intel(R) Xeon(R) E5-2670 at 2.60GHz, with 8 hyperthreads (recall ours is an E5-2630 at 2.30GHz). We put in gray the times obtained directly from their paper.

Table 4. Query times (in seconds, rounded to the nearest integer) for cyclic queries on SNAP. TO stands for timeout (>1800 secs), whereas OM indicates an out-of-memory crash. Times in gray are obtained directly from Nguyen et al. [35].

		wiki-vote	Gnutella31	Gnutella04	Brightkite	Facebook	Enron	GrQc	CondMat	Twitter	Slashdot0902	Slashdot0811	Epinions1	Pokec	LiveJournal1	Orkut
3-Clique	Qdag	0	0	0	0	0	0	0	0	3	4	3	3	596	956	TO
	EmptyHeaded	0	0	0	0	0	0	0	0	2	0	0	0	5	12	73
	LTJ	0	0	0	0	0	0	0	0	5	1	1	1	75	165	742
	MS	1	1	0	2	1	3	0	1	23	7	5	6	282	TO	TO
	PSQL	1446	6	2	TO	575	TO	10	348	TO	TO	TO	TO	TO	TO	TO
	MonetDB	TO	3	3	945	947	TO	22	98	TO	TO	TO	TO	TO	TO	TO
	Virtuoso	18	2	1	17	23	46	1	4	296	75	68	158	TO	TO	TO
	Neo4j	348	19	6	212	250	418	4	32	TO	1441	1308	1745	TO	TO	TO
4-Clique	Qdag	26	47	5	37	20	29	0	2	351	698	616	450	TO	TO	TO
	EmptyHeaded	6	0	0	4	12	5	0	1	29	2	2	5	61	OM	TO
	LTJ	3	0	0	11	9	4	0	1	427	4	4	13	644	TO	TO
	MS	11	1	0	10	31	25	1	2	288	39	32	96	TO	TO	TO
	PSQL	TO	52	10	TO	TO	TO	1021	TO	TO	TO	TO	TO	TO	TO	TO
	MonetDB	TO	17	15	TO	TO	TO	1219	TO	TO	TO	TO	TO	TO	TO	TO
	Virtuoso	447	2	0	364	1240	968	2	38	TO	1247	1273	TO	TO	TO	TO
	Neo4j	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
4-Cycle	Qdag	3	1	0	70	21	59	0	4	136	1430	923	221	TO	TO	TO
	EmptyHeaded	22	1	0	14	11	29	0	2	350	OM	OM	OM	OM	OM	OM
	LTJ	11	1	0	4	8	7	0	1	171	31	29	34	1416	TO	TO
	MS	24	3	1	17	23	59	0	3	587	183	156	268	TO	TO	TO
	PSQL	309	4	1	1394	539	TO	47	112	TO	TO	TO	TO	TO	TO	TO
	MonetDB	502	1	1	657	347	TO	19	60	TO	TO	TO	TO	TO	TO	TO
	Virtuoso	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
	Neo4J	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO

Table 5. Query times (in seconds, rounded to the nearest integer) for acyclic queries on small SNAP graphs, using selectivity 8 and 80. TO stands for timeout (>1800 secs), whereas OM indicates an out-of-memory crash. Times in gray are obtained directly from Nguyen et al. [35].

		wiki-vote		Gnutella31		Gnutella04		Brightkite		Facebook		Enron		GrQc		CondMat	
		80	8	80	8	80	8	80	8	80	8	80	8	80	8	80	8
1-Tree	Qdag	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0
	EmptyHeaded	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	LTJ	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	MS	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
	PSQL	0	1	0	0	0	0	0	1	0	0	0	2	0	0	0	0
	MonetDB	4	5	1	1	0	0	TO	TO	TO	TO	TO	TO	0	0	1	1
2-Comb	Qdag	0	8	19	29	1	4	44	206	0	1	12	115	0	0	3	12
	EmptyHeaded	0	0	0	0	0	0	0	1	0	0	0	2	0	0	0	0
	LTJ	0	6	0	0	0	0	1	20	0	3	1	50	0	0	0	2
	MS	0	0	0	1	0	0	1	3	0	0	1	2	0	0	1	1
	PSQL	0	51	0	0	0	0	2	206	0	29	3	553	0	0	0	6
	MonetDB	388	478	3	3	1	1	TO	TO	TO	TO	TO	TO	5	5	53	62
3-Path	Qdag	0	6	5	17	0	2	39	198	0	0	10	114	0	0	2	12
	EmptyHeaded	0	0	0	0	0	0	0	1	0	0	0	2	0	0	0	0
	LTJ	0	2	0	0	0	0	1	20	0	0	1	40	0	0	0	2
	MS	0	1	0	0	0	0	1	4	0	0	1	3	0	0	1	1
	PSQL	0	12	0	0	0	0	2	203	0	3	3	556	0	0	0	7
	MonetDB	128	131	1	1	0	0	993	1036	45	56	TO	TO	6	5	57	68
	Virtuoso	1	16	0	0	0	0	18	319	0	4	37	719	0	1	1	10
	Neo4j	4	71	1	2	0	1	82	633	4	19	163	1584	1	4	6	42
4-Path	Qdag	21	690	513	1725	19	154	TO	TO	0	10	TO	TO	1	4	278	TO
	EmptyHeaded	1	2	0	0	0	0	7	OM	1	4	OM	OM	0	1	1	3
	LTJ	4	193	0	0	0	0	44	1155	1	9	75	TO	1	5	6	59
	MS	1	1	0	1	0	0	4	9	0	1	4	7	0	0	2	4
	PSQL	3	1099	0	1	0	0	299	TO	0	102	914	TO	0	39	4	437
	MonetDB	TO	TO	3	4	1	2	TO	TO	TO	TO	TO	TO	230	321	TO	TO
	Virtuoso	30	1363	0	1	0	0	1664	TO	5	189	TO	TO	4	29	37	577
	Neo4j	161	TO	1	7	0	3	TO	TO	105	437	TO	TO	23	109	201	1309

Table 6. Query times (in seconds, rounded to the nearest integer) for acyclic queries on medium-size and large SNAP graphs, using selectivity 10, 100, and 1000. TO stands for timeout (>1800 secs), whereas OM indicates an out-of-memory crash. Times in gray are obtained directly from Nguyen et al. [35].

		Twitter			Slashdot0902			Slashdot0811			Epinions1			Pokec			LiveJournal1			Orkut		
		1K	100	10	1K	100	10	1K	100	10	1K	100	10	1K	100	10	1K	100	10	1K	100	10
1-Tree	Qdag	0	0	2	0	1	7	0	1	5	0	0	1	49	530	TO	201	TO	TO	TO	TO	TO
	EmptyHeaded	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	1	35
	LTJ	0	0	2	0	0	1	0	0	1	0	0	0	1	3	30	1	7	82	2	32	443
	MS	1	2	2	1	1	1	1	1	1	0	1	1	28	32	46	55	64	97	79	100	152
	PSQL	0	1	44	0	0	4	0	0	1	0	0	2	1	17	160	25	36	513	2	106	TO
	MonetDB	88	78	95	TO	TO	TO	TO	TO	TO	12	11	10	TO	TO	TO	TO	TO	TO	TO	TO	TO
2-Comb	Qdag	2	59	363	8	400	TO	7	264	TO	1	30	300	TO	TO	TO	TO	TO	TO	TO	TO	TO
	EmptyHeaded	0	1	1	0	1	2	0	1	2	0	0	1	1	2	18	4	6	391	6	TO	TO
	LTJ	1	15	180	1	8	117	1	11	101	0	5	41	11	140	1780	66	1161	TO	395	TO	TO
	MS	2	7	12	1	4	6	1	4	6	1	1	3	64	156	272	128	282	507	312	575	TO
	PSQL	2	205	TO	0	5	1014	0	6	936	0	3	288	14	196	TO	153	1111	TO	162	TO	TO
	MonetDB	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
3-Path	Qdag	2	55	391	10	364	TO	6	258	TO	1	30	328	TO	TO	TO	TO	TO	TO	TO	TO	TO
	EmptyHeaded	0	0	2	0	1	2	0	1	2	0	0	1	1	3	7	4	5	263	5	21	560
	LTJ	1	13	144	1	8	98	1	8	110	0	5	27	9	120	1521	61	1035	TO	60	825	TO
	MS	1	5	18	1	4	10	1	4	10	0	1	4	25	129	408	68	259	TO	111	451	TO
	PSQL	2	215	TO	0	5	938	0	6	890	0	2	243	8	166	TO	142	1011	TO	TO	TO	TO
	MonetDB	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
	Virtuoso	7	59	1435	8	52	1433	6	65	1268	2	15	403	75	784	TO	TO	TO	TO	TO	TO	TO
	Neo4j	57	323	TO	28	370	TO	41	405	TO	15	88	877	TO	TO	TO	TO	TO	TO	TO	TO	TO
4-Path	Qdag	239	TO	TO	TO	TO	TO	TO	TO	TO	110	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
	EmptyHeaded	16	OM	OM	45	OM	OM	13	OM	OM	6	33	OM	OM	OM	OM	OM	OM	OM	TO	TO	TO
	LTJ	103	1286	TO	3	203	TO	62	240	TO	4	68	TO	710	TO	TO	TO	TO	TO	TO	TO	TO
	MS	8	22	46	7	13	24	7	14	23	2	6	10	206	556	TO	470	TO	TO	697	TO	TO
	PSQL	TO	TO	TO	9	1211	TO	10	1637	TO	1	470	TO	94	TO	TO	TO	TO	TO	1378	TO	TO
	MonetDB	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
	Virtuoso	710	TO	TO	1058	TO	TO	657	TO	TO	46	1785	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
	Neo4j	TO	TO	TO	TO	TO	TO	TO	TO	TO	1097	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO