Optimal Joins using Compact Data Structures

Gonzalo Navarro
DCC, U. of Chile & IMFD, Chile

Juan L. Reutter
DCC, PUC & IMFD, Chile

Javiel Rojas-Ledesma
DCC, U. of Chile & IMFD, Chile

Abstract

Worst-case optimal join algorithms have gained a lot of attention in the database literature. We now count with 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 we either need to build completely new indexes, or we 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 may be non-negligible.

We show that optimal algorithms can be obtained directly from a representation that regards the relations as point sets in variable-dimensional grids, without the need of extra storage. Our representation is a compact quadtree for the static indexes, and a dynamic quadtree sharing subtrees (which we dub a qdag) for intermediate results. We develop a compositional algorithm to process full join queries under this representation, and show that the running time of this algorithm is worst-case optimal in data complexity. Remarkably, we can extend our framework to evaluate more expressive queries from relational algebra by introducing a lazy version of qdags (lqdags). Once again, we can show that the running time of our algorithms is worst-case optimal.

2012 ACM Subject Classification Theory of computation → Database query processing and optimization (theory); Theory of computation → Data structures and algorithms for data management

Keywords and phrases Join algorithms, Compact data structures, Quadtrees, AGM bound

Digital Object Identifier 10.4230/LIPIcs.ICDT.2020.24

Funding This work was partially funded by Millennium Institute for Foundational Research on Data (IMFD), and by project CONICYT Fondecyt/Postdoctorado No. 63130209.

1 Introduction

The state of the art in query processing has recently been shaken by a new generation of join algorithms with strong optimality guarantees based on the AGM bound of queries: the maximum size of the output of the query over all possible relations with the same cardinalities [2]. One of the basic principles of these algorithms is to disregard the traditional notion of a query plan, favoring a strategy that can take further advantage of the structure of the query, while at the same time taking into account the actual size of the database [14, 16].

Several of these algorithms have been implemented and tested in practice with positive results [8, 18], 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 [21], Minesweeper [15], or InsideOut [10] must select a *global order* on the attributes, and assume that relations are indexed in a way that is consistent with these attributes [18]. If one wants to use these algorithms with more flexibility in the way attributes are processed, then one would probably need to instantiate several

combinations of B+ trees or other indexes [8]. On the other hand, more involved algorithms such as Tetris [9] or Panda [11] require heavier data structures that allow reasoning over potential tuples in the answer.

Our goal is to develop optimal join algorithms that minimize the storage for additional indexes while at the same time being independent of a particular ordering of attributes. We address this issue by resorting to compact data structures: indexes using a nearly-optimal amount of space while supporting all operations we need to answer join queries.

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 [4]. 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 with attributes $A_1, \ldots, A_d$ 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 dynamic 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 with attributes in $[0, \ell - 1]$, and let $d = |\bigcup_i \mathcal{A}_i|$. We can represent the relations using only $\sum_i (|\mathcal{A}_i| + 2 + o(1))|R_i| \log \ell + O(n \log d)$ bits, so that the result of a join query $J = R_1 \bowtie \cdots \bowtie R_n$ over a database instance D, can be computed in $\tilde{O}(AGM)$ time*[1].

Note that just storing the tuples in any R_i requires $|\mathcal{A}_i||R_i| \log \ell$ bits, thus our representation adds only a small extra space of $(2 + o(1))|R_i| \log \ell + O(n \log d)$ bits (basically, two words per tuple, plus a negligible amount that only depends on the schema). Instead, any classical index on the raw data (such as hash tables or B+-trees) would pose a linear extra space, $O(|\mathcal{A}_i||R_i| \log \ell)$ bits, often multiplied by a non-negligible constant (especially if one needs to store multiple indexes on the data).

Our join algorithm works in a rather different way than the most popular worst-case 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. [16] 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, quadtrees divide the output space, which corresponds to a grid of size ℓ^3, into 8 subgrids of size $(\ell/2)^3$, and for each of these grids it recursively evaluates the query. As it turns out, this strategy is as good as the generic strategy defined by Ngo et al. [16] to compute joins, and can even be extended to other relational operations, as we explain next.

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. But there are other operations that we can define and implement. For example, the synchronized Or of two quadtrees gives a compact

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

representation of their union, and complementing the quadtree values can be done by 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, as in relational algebra.

To support these operations we introduce lazy qdags, or lqdags for short, in which nodes may be additionally labeled with query expressions. The idea is to be able to delay the computation of an expression until we know such computation is needed to derive the output. 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 sizes $s_1, \ldots, s_n$, respectively, of a database D, then there exists a database D' with relations $R'_1, \ldots, R'_n$ of sizes $O(s_1), \ldots, O(s_n)$, respectively, where the output of F over $R'_1, \ldots, R'_n$ is of size $\Omega(T)$. We prove that lqdags allow us to maintain optimality in the presence of union and negation operators:

▶ **Theorem 1.2.** *Let Q be a relational algebra query built with joins, union and complement, and where no relation appears more than once in Q. Then there is an algorithm to evaluate Q that is worst-case optimal 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. Our algorithm is optimal because it can choose locally between both strategies: by dividing into quadrants one finds dense regions of T in which computing $\overline{T}$ is cheaper, while in sparse regions the algorithm first computes the join of R and S.

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 [13]. 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. Under the recent trend of maintaining the database in the aggregate main memory of a distributed system, a compact representation leads to using fewer computers, thus reducing hardware, communication, and energy costs, while improving performance.

2 Quadtrees

A Region Quadtree [6, 19] is a structure used to store points in two-dimensional grids of $\ell \times \ell$. We focus on the variant called MX-Quadtree [22, 19], 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. (The deepest internal nodes, whose children are 1×1 grids, store instead four integers in $\{0, 1\}$ to encode their cells.)

Assume each data point is described using the binary representation of each of its coordinates (i.e., as a pair of $\log \ell$-bit vectors). We order the grid quadrants so that the first contains all points with coordinates of the form $(0 \cdot c_x, 0 \cdot c_y)$, for $\log \ell - 1$ bit vectors c_x and c_y, the second contains points $(0 \cdot c_x, 1 \cdot c_y)$, the third $(1 \cdot c_x, 0 \cdot c_y)$, and the last quadrant stores the points $(1 \cdot c_x, 1 \cdot c_y)$. Fig. 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 used to represent data points in a d-dimensional grid G of size ℓ^d. Here, an empty grid is

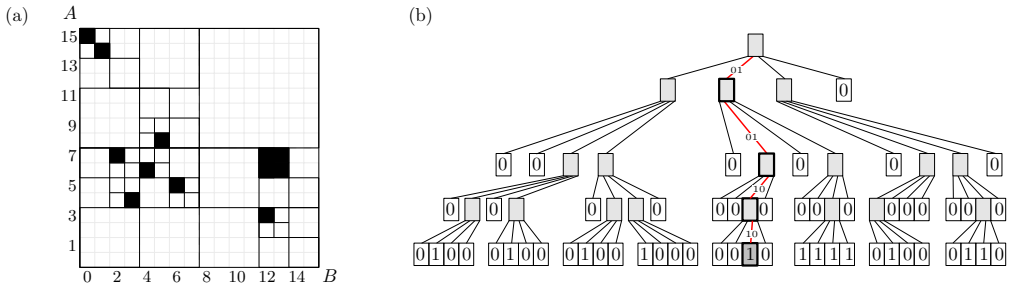

Figure 1 A quadtree representing $R(A,B) = \{(4,3),(7,2),(5,6),(6,4),(3,12),(6,12),(6,13),(7,12),(7,13),(8,5),(14,1),(15,0)\}$. (a) Representation of $R(A,B)$ in a $2^4 \times 2^4$ grid, and representation of the hierarchical partition defining the quadtree. The black cells correspond to points in R. (b) The quadtree representing R. The shadowed leaf of the tree corresponds to the point $p = (3,12)$. Concatenating the labels in the path down to p yield the bit-string '**01011010**' which encodes the first (resp. second) coordinate of p in the bits at odd (resp. even) positions ($3 = 0011, 12 = 1100$).

represented by a leaf and a nonempty grid corresponds to an internal node with 2^d children representing the 2^d subspaces spanning from combining the first bits of each dimension. Generalizing the case $d = 1$, the children are ordered using the Morton [12] partitioning of the grid: a sequence of 2^d subgrids of size $(\ell/2)^d$ in which the i-th subgrid of the partition, represented by the binary encoding b_i of i, is defined by all the points $(b_{c_1}, \ldots, b_{c_d})$ in which the word formed by concatenating the first bit of each string b_{c_j} is precisely the string b_i.

A quadtree with p points has at most $p \log \ell$ nodes (i.e., root-to-leaf paths). A refined analysis in two dimensions [7, Thm. 1] shows that quadtrees have fewer nodes when the points are clustered: if the points distribute along c clusters, p_i of them inside a subgrid of size $\ell_i \times \ell_i$, then there are in total $O(c \log \ell + \sum_i p_i \log \ell_i)$ nodes in the quadtree. The result easily generalizes to d dimensions: the cells are of size ℓ_i^d and the quadtree has $O(c \log \ell + \sum_i p_i \log \ell_i)$ internal nodes, each of which stores 2^d pointers to children (or integers, in the last level).

Brisaboa et al. [4] introduced a compact quadtree representation called the k^d-tree. They represent each internal quadtree node as the 2^d bits telling which of its quadrants is empty (0) or nonempty (1). Leaves and single-cell nodes are not represented because their data is deduced from the corresponding bit of their parent. The k^d-tree is simply a bitvector V obtained by concatenating the 2^d bits of every (internal) node in levelwise order. Each node is identified with its order in this deployment, the root being 1. Navigation on the quadtree (from a parent to its children, and vice-versa) is simulated in constant time using $o(|V|)$ additional bits on top of V. On a quadtree in dimension d storing p points, the length of the bitvector V is $|V| \leq 2^d p \log \ell$, increasing exponentially with d. This bitvector is sparse, however, because it has at most $p \log \ell$ 1s, one per quadtree node. We then resort to a representation of high-arity cardinal trees introduced by Benoit et al. [3, Thm. 4.3], which requires only $(d+2)p \log \ell + o(p \log \ell) + O(\log d)$ bits, and performs the needed tree traversal operations in constant time.

▶ **Observation 2.1.** ***(cf. Benoit et al. [3], Thm. 4.3)*** *Let Q be a quadtree storing p points in d dimensions with integer coordinates in the interval $[0, \log \ell - 1]$. Then, there is a representation of Q which uses $(d + 2 + o(1))p \log \ell + O(\log d)$ bits, can be constructed in linear expected time, and supports constant time parent-children navigation on the tree.*

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 [1]. 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 $\{A, B, C\}$ denote the domains of A, B and C respectively, 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 with 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 $n^{3/2}$ [2], while building Q^*_R costs $\Omega(n\ell)$ time and space. This inefficiency stems from the fact that quadtrees are not smart to represent relations of the form $R^*(\mathcal{A}) = R(\mathcal{A}') \times \text{All}(\mathcal{A} \setminus \mathcal{A}')$, where $\mathcal{A}' \subset \mathcal{A}$, with respect to the size 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^d (that has certain regularities) using only a reference to a d'-dimensional quadtree $Q^{d'}$, with $d' \leq d$, and an auxiliary *mapping function* that defines how to use $Q^{d'}$ to simulate navigation over Q^d. 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.

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

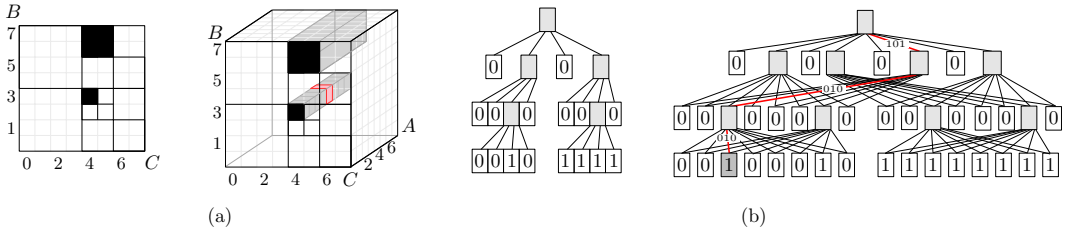

Figure 2 An illustration of a qdag for $S^*(\{A,B,C\}) = \text{All}(A) \times S(B,C)$, with $S(B,C) = \{(3,4),(6,4),(6,5),(7,4),(7,5)\}$. a) A geometric representation of $S(B,C)$ (left), and $S^*(\{A,B,C\})$ (right). b) A quadtree Q_S for $S(B,C)$ (left), and the directed acyclic graph induced by the 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 $p = (4,3,4)$. The leaf representing p in the qdag can be reached following the path highlighted in (b). Note the relation between the binary representation (**1**00,**0**10,**1**00) of p, and the Morton codes **101**, **010**, 010 of the nodes in the path from the root to the leaf for p.

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 values will in turn be identical to those 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 S (see Fig. 2 for an example). Thus, we can implicitly represent Q_S^* by the pair $(Q_S, M = [0,1,2,3,0,1,2,3])$: the root of Q_S^* is the root of Q_S, and the i-child of the root of Q_S^* is represented by the pair (C, M), where C 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 which allow the evaluation of multijoin queries in worst-case optimal time.

▶ **Definition 3.1** (qdag). *Let $Q^{d'}$ be a quadtree representing a relation with d' attributes. A qdag Q^d, for $d \geq d'$, is a pair $(Q^{d'}, M)$, with $M : [0, 2^d - 1] \to [0, 2^{d'} - 1]$. This qdag represents a quadtree Q, which is called the* completion *of Q^d, as follows:*

1. *If $Q^{d'}$ represents a single cell, then Q represents a single cell with the same value.*
2. *If $Q^{d'}$ represents a d'-dimensional grid empty of points, then Q represents a d-dimensional grid empty of points.*
3. *Otherwise, the roots of both $Q^{d'}$ and Q are internal nodes, and for all $0 \leq i < 2^d$, the i-th child of Q is the quadtree represented by the qdag $(C(Q^{d'}, M[i]), M)$, where $C(Q^{d'}, j)$ denotes the j-th child of the root node of quadtree $Q^{d'}$.*

We say that a qdag represents the same relation R represented by its completion. Note that, for any d-dimensional quadtree Q, one can generate a qdag whose completion is Q simply as the pair (Q, M), where M is the *identity mapping* $M[i] = i$, for all $0 \leq i < 2^d$. We can then describe all our operations over qdags. Note, in particular, that we can use mappings to represent any reordering of the attributes.

In terms of representation, the references to quadtree nodes consist of the identifier of the quadtree and the index of the node in level-wise order. This suffices to access the node in constant time from its compact representation. 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 3.1, will be useful for the navigation of qdags. Operation VALUE yields a 0 iff the subgrid represented by the qdag is empty (thus the qdag

Algorithm 1 VALUE

Require: qdag (Q, M) with grid side ℓ.
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 CHILDAT

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

1: **return** $(C(Q, M[i]), M)$

Algorithm 3 EXTEND

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

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

is a leaf); a 1 if the qdag is a full single cell, and ½ if it is an internal node. Operation CHILDAT lets us descend by a given child from internal nodes representing nonempty grids. The operations "integer Q", "Q is a leaf", and "$C(Q, j)$" are implemented in constant time on the compact representation of Q.

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.

▶ **Definition 3.2.** *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 $R \times All(\mathcal{A} \setminus \mathcal{A}')$.*

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 3.2. Each node of Q_R has 8 children, while each node 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 3.3.** *Let $|\mathcal{A}| = d$ in Definition 3.2. 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).

Algorithm 4 MultiJoin

Require: Relations $R_1, \ldots, R_n$, stored as qdags $Q_1, \ldots, Q_n$; 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: $\quad Q_i^* \leftarrow \textsc{Extend}(R_i, \mathcal{A})$
3: **return** $\textsc{And}(Q_1^*, \ldots, Q_n^*)$

Algorithm 5 And

Require: n qdags Q_1, Q_2, $\ldots$,Q_n representing relations $R_1(\mathcal{A}), R_2(\mathcal{A}), \ldots, R_n(\mathcal{A})$.

Ensure: A quadtree representing the relation $\bigcap_{i=1}^{n} R_i(\mathcal{A})$.

1: $m \leftarrow \min\{\textsc{Value}(Q_1), \ldots, \textsc{Value}(Q_n)\}$
2: **if** $\ell = 1$ **then return** the integer m
3: **if** $m = 0$ **then return** a leaf
4: **for** $i \leftarrow 0, \ldots, 2^d - 1$ **do**
5: $\quad C_i \leftarrow \textsc{And}(\textsc{ChildAt}(Q_1, i), \ldots, \textsc{ChildAt}(Q_n, i))$
6: **if** $\max\{\textsc{Value}(C_0), \ldots, \textsc{Value}(C_{2^d-1})\} = 0$ **then return** a leaf
7: **return** a quadtree with children $C_0, \ldots, C_{2^d-1}$

The computation of m'_d is done with a constant table (that depends only on the database dimension d) of size $O(2^{3d})$:[2] The argument $\mathcal{A}$ is given as a bitvector of size d telling which attributes are in $\mathcal{A}$, the qdag on $\mathcal{A}'$ stores a bitvector of size d telling which attributes are in $\mathcal{A}'$, and the table receives both bitvectors and m_d and returns m'_d. ◀

3.3 Join algorithm

Now that we can efficiently represent relations of the form $R(\mathcal{A}') \times \text{All}(\mathcal{A} \setminus \mathcal{A}')$, 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. Then we compute their intersection, building a quadtree representing the output of the query. 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 key 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.

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

▶ **Definition 3.4.** *Let $Q_1, \ldots, Q_n$ be qdags representing relations $R_1, \ldots, R_n$, all over the attribute set $\mathcal{A}$. Operation $\textsc{And}(Q_1, \ldots, Q_n)$ returns a quadtree Q that represents the relation $R_1 \cap \ldots \cap R_n$.*

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

[2] They can be reduced to two tables of size $O(2^{2d})$, but we omit the details for simplicity.

can represent it succinctly in linear expected time so that, for instance, it can be cached for future queries involving the output represented by Q[3].

Analysis of the algorithm. We compute the output Q of $\text{AND}(Q_1, \ldots, Q_n)$ in time $O(2^d \cdot (||Q_1|| + \cdots + ||Q_n||))$. More precisely, the time is bounded by $O(2^d \cdot |Q^+|)$, where Q^+ is the quadtree that would result from Algorithm 5 if we remove the pruning step of line 6. We name this quadtree Q^+ as the *non-pruned version* of Q. Although the size of the actual output Q can be much smaller than that of Q^+,we can still prove that our time is optimal in the worst case. We start with a technical result.

▶ **Lemma 3.5.** *The* AND *operation can be supported in time $O(M \cdot 2^d n \log \ell)$, where M is the maximum number of nodes in a level of Q^+.*

Proof. We show that Algorithm 5 meets the conditions of the lemma. Let m_j be the number of nodes of depth j in Q^+, and then $M = \max_{0 \le j < \log \ell} m_j$. The number of steps performed by Algorithm 5 is bounded by $n \cdot (\sum_{0 \le j < \log \ell} m_j \cdot 2^d) \le n \cdot M \cdot \log \ell \cdot 2^d$: In 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). ◀

All we need to prove (data) optimality is to show that $|Q^+|$ is bounded by the size of the real output of the query. Recall that, for a join query J on a database D, we use $2^{\rho^*(J,D)}$ to denote the AGM bound [2] 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 3.6.** *Let $J = R_1 \bowtie \ldots \bowtie R_n$ be a full join query, and D a database over schema $\{R_1, \ldots, R_n\}$, 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 $1 \le i \le n$ and $N = \sum_i |R_i|$ be the total amount of tuples in the database. The relations $R_1, \ldots, R_n$ can then be stored within $\sum_i (|\mathcal{A}_i| + 2 + o(1))|R_i| \log \ell + O(n \log d)$ bits, so that the output for J can be computed in time $O(2^{\rho^*(J,D)} \cdot 2^d n \log \min(\ell, N)) = \tilde{O}(2^{\rho^*(J,D)})$.*

Proof. The space usage is a simple consequence of Observation 2.1. As for the time, to solve the join query J we simply encapsulate the quadtrees representing $R_1, \ldots, R_n$ in qdags $Q_1, \ldots, Q_n$, and use Algorithm 4 to compute the result of the query. We now show that Algorithm 4 runs in time within the bound of the theorem. First, assume that $\log \ell$ is $O(\log N)$. Let each relation R_i be over attributes $\mathcal{A}_i$, and $\mathcal{A} = \bigcup \mathcal{A}_i$ with $d = |\mathcal{A}|$. Let $Q_i^* = \text{EXTEND}(Q_i, \mathcal{A})$, $Q = \text{AND}(Q_1^*, \ldots, Q_n^*)$, and Q^+ be the non-pruned version of Q. The cost of the EXTEND operations is only $O(2^d n)$, according to Lemma 3.3, so the main cost owes to the AND operation.

If the maximum M of Lemma 3.5 is reached at the lowest level of the decomposition, where we store integers 0 or 1, then we are done: each 1 at a leaf of Q^+ exists in Q as well because that single tuple is present in all the relations $R_1, \ldots, R_n$. Therefore, M is bounded by the AGM bound of J and the time of the AND operation is bounded by $O(2^{\rho^*(J,D)} \cdot 2^d n \log \ell)$.

Assume instead that M is the number of internal nodes at depth $0 < j < \log \ell$ of Q^+ (if M is reached at depth 0 then $M = 1$). Intuitively, we will take the relations at the granularity of level j, and show that there exists a database D' where such a $(2^j)^d$ relation arises in the last level and thus the answer has those M tuples.

We then construct the following database D' with relations R_i': For a binary string c, let $\text{pre}(c, j)$ denote the first j bits of c. Then, for each relation R_i and each tuple $(c_1, \ldots, c_{d_i})$

[3] This consumes linear expected time due to the use of perfect hashing in the compact representation [3].

in R_i, where $d_i = |\mathcal{A}_i|$, let R'_i contain the tuples $(0^{\log \ell - j}\text{pre}(c_1, j), 0^{\log \ell - j}\text{pre}(c_2, j) \ldots, 0^{\log \ell - j}\text{pre}(c_{d_i}, j))$, 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 a relation in D to a single tuple in D', we still have that each relation R'_i in D' contains at most as many tuples as relation R_i in D. Moreover, if we again store every R'_i as a qdag and process their join as in Algorithm 4, then by construction we have in this case that the leaves of the tree resulting from the AND operation contain exactly M nodes with 1, and that this is the maximum number of nodes in a level of this tree. Since the leaves represent tuples in the answer, we have that $M \leq 2^{\rho^*(J,D')} \leq 2^{\rho^*(J,D)}$, which completes the proof for the case when $\log \ell$ is $O(\log N)$.

Finally, when $\log N$ is $o(\log \ell)$, we can convert $O(\log \ell)$ to $O(\log N)$ in the time complexities by storing $R_1, \ldots, R_n$ using quadtrees, with a slight variation. We store the values of the attributes appearing in any relation in an auxiliary data structure (e.g., an array), and associate an $O(\log N)$-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). In this case, we represent the relations in quadtrees, but using the identifiers of the attribute values instead of the values themselves. This representation requires at most $dN \log \ell$ bits for the representation of the distinct attribute values and $O(dN \log N)$ bits for the representation of the quadtrees. Thus, in total it requires $dN \log \ell + O(dN \log N) = dN \log \ell(1 + O(\log N / \log \ell)) = dN \log \ell(1 + o(1))$, which is within the stated bound. Note that in both cases, the height of the quadtrees representing the relations is $O(\log N)$, and this is the multiplying factor in the time complexities. ◀

4 Extending Worst-Case Optimality to More General Queries

In this section we turn to design worst-case optimal algorithms for more expressive queries. At this point it should be clear that we can deal with set operations: 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 worst-case optimal algorithm to evaluate formulas expressed as combinations of join, union, and complement operations (which we refer to as JUC-queries; note that intersection is a particular case of join). We do not study other operations like *selection* and *projection* because these are easily solved in time essentially proportional to the size of the output, but refer to Appendix A for more details on how projection interplays with the rest of our framework.

The key ingredient of our algorithm is to deal with these operations in a *lazy* form, allowing unknown intermediate results so that all 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.

Algorithm 6 VALUE on extended qdags

Require: 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

While quadtree leaves representing a single cell store the cell value, 0 or 1, quadtree leaves at higher levels always represent subgrids full of 0s. We now generalize the representation, so that quadtree leaves at any level store an integer, 0 or 1, which is the value of all the cells in the subgrid represented by the leaf. The generalization impacts on the way to compute VALUE, 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: an internal node with all zero children [5]. Note that replacing a full subgrid with this configuration can only decrease the size of the representation.

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

▶ **Definition 4.1** (lqdag)**.** *An lqdag L is a pair (f, o), where f is a* functor *and o is a list of* operands*. The* completion *of L is the quadtree $Q_R = Q_R(\mathcal{A})$ representing relation $R(\mathcal{A})$ if L is as follows:*

1. $(\texttt{QTREE}, Q_R)$*, where the lqdag just represents Q_R;*
2. $(\texttt{NOT}, Q_{\overline{R}})$*, where $Q_{\overline{R}}$ is the quadtree representing the complement of Q_R;*
3. $(\texttt{AND}, L_1, L_2)$*, where L_1 and L_2 are lqdags and Q_R represents the intersection of their completions;*
4. $(\texttt{OR}, L_1, L_2)$*, where L_1 and L_2 are lqdags and Q_R represents the union of their completions;*
5. $(\texttt{EXTEND}, L_1, \mathcal{A})$*, where lqdag L_1 represents $R'(\mathcal{A}')$, $\mathcal{A}' \subseteq \mathcal{A}$, and Q_R represents $R(\mathcal{A}) = R'(\mathcal{A}') \times All(\mathcal{A} \setminus \mathcal{A}')$.*

Note that, for a quadtree Q_R representing a relation $R(\mathcal{A}')$, and a set of attributes $\mathcal{A}$, the qdag $Q^*_R = (Q_R, M_\mathcal{A})$ that represents the relation $R \times \text{All}(\mathcal{A} \setminus \mathcal{A}')$ can be expressed as the lqdag $(\texttt{EXTEND}, (\texttt{QTREE}, Q_R), \mathcal{A})$. In this sense, lqdags are extensions of qdags. To further 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

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

It is apparent that one can define other operations, like `JOIN` and `DIFF`, by combining the operations defined above:

$$\begin{aligned} (\texttt{JOIN}, L_1(\mathcal{A}_1), L_2(\mathcal{A}_2)) &= (\texttt{AND}, (\texttt{EXTEND}, L_1, \mathcal{A}_1 \cup \mathcal{A}_2), (\texttt{EXTEND}, L_2, \mathcal{A}_1 \cup \mathcal{A}_2)) \\ (\texttt{DIFF}, L_1, L_2) &= (\texttt{AND}, L_1, (\texttt{NOT}, L_2)) \end{aligned}$$

Note that in the definition of the lqdag for `NOT`, the operand is a quadtree instead of an lqdag, and then, for example, L_2 should be a quadtree in the definition of `DIFF`, in principle. However, we can easily get around that restriction by pushing down the `NOT` operators until the operand is a quadtree or the `NOT` is cancelled with another `NOT`. For instance, a `NOT` over an lqdag $(\texttt{AND}, Q_1, Q_2)$ is equivalent to $(\texttt{OR}, (\texttt{NOT}, Q_1), (\texttt{NOT}, Q_2))$, and analogously with the other functors. The restriction, however, does limit the types of formulas for which we achieve worst-case optimality, as shown later.

Algorithm 7 VALUE function for NOT

Require: A Quadtree Q.
Ensure: Value of the root of $(\texttt{NOT}, Q)$.

1: **return** $1 - \text{VALUE}(Q)$

Algorithm 8 CHILDAT function for NOT

Require: A Quadtree Q in dimension d, and an integer $0 \leq i < 2^d$.
Ensure: An lqdag for the i-th child of $(\texttt{NOT}, Q)$.

1: **return** $(\texttt{NOT}, \text{CHILDAT}(Q, i))$

Algorithm 9 VALUE function for AND

Require: Lqdags L_1 and L_2.
Ensure: The value of the root of $(\texttt{AND}, L_1, L_2)$.

1: **if** $\text{VALUE}(L_1) = 0$ **or** $\text{VALUE}(L_2) = 0$ **then return** 0
2: **if** $\text{VALUE}(L_1) = 1$ **then return** $\text{VALUE}(L_2)$
3: **if** $\text{VALUE}(L_2) = 1$ **then return** $\text{VALUE}(L_1)$
4: **return** $\Diamond$

Algorithm 10 CHILDAT function for AND

Require: Lqdags L_1 and L_2 in dimension d, integer $0 \leq i < 2^d$.
Ensure: An lqdag for the i-th child of $(\texttt{AND}, L_1, L_2)$.

1: **if** $\text{VALUE}(L_1) = 1$ **then return** $\text{CHILDAT}(L_2, i)$
2: **if** $\text{VALUE}(L_2) = 1$ **then return** $\text{CHILDAT}(L_1, i)$
3: **return** $(\texttt{AND}, \text{CHILDAT}(L_1, i), \text{CHILDAT}(L_2, i))$

To understand why we call lqdags lazy, consider the operation Q_1 AND Q_2 over quadtrees Q_1, Q_2. If any of the values 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 any of the values is 1, then the result of the operation is the other. However, if both values are ½, one cannot be sure of the value of the root until the AND between the children of Q_1 and Q_2 has been evaluated. 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 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 7–14, all constant-time. Note that CHILDAT can only be invoked when VALUE $= ½$ or $\Diamond$. The base case is $\text{VALUE}(\texttt{QTREE}, Q) = \text{VALUE}(Q)$ and $\text{CHILDAT}((\texttt{QTREE}, Q), i) = \text{CHILDAT}(Q, i)$, where we enter the quadtree and resort to the algorithms based on the compact representation of Q. We will assume that $\text{VALUE}(Q)$ returns ½ for internal nodes, and thus the implementation of VALUE for EXTEND is trivial (compare Algorithms 6 and 13 under this assumption).

Note that the recursive calls of Algorithms 7-14 traverse the nodes of the relational formula (fnodes, for short), and terminate immediately upon reaching an fnode of the form $(\texttt{QTREE}, Q)$. 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 CHILDAT, one can efficiently evaluate a relational formula using lqdags.

4.2 Evaluating JUC queries

To evaluate a formula F represented as an lqdag L_F, we compute the completion Q_F of L_F, that is, the quadtree Q_F representing the output of F.

To implement this we introduce the idea of *super-completion* of an lqdag. The super-completion Q_F^+ of L_F is the quadtree induced by navigating L_F, and interpreting the values $\Diamond$ as ½ (see Algorithm 15). 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

Algorithm 11 VALUE function for `OR`

Require: Lqdags L_1 and L_2.
Ensure: The value of the root of $(\texttt{OR}, L_1, L_2)$.

1: **if** VALUE$(L_1) = 1$ **or** VALUE$(L_2) = 1$ **then return** 1
2: **if** VALUE$(L_1) = 0$ **then return** VALUE(L_2)
3: **if** VALUE$(L_2) = 0$ **then return** VALUE(L_1)
4: **return** $\Diamond$

Algorithm 12 CHILDAT function for `OR`

Require: Lqdags L_1 and L_2 in dimension d, integer $0 \leq i < 2^d$.
Ensure: An lqdag for the i-th child of $(\texttt{OR}, L_1, L_2)$.

1: **if** VALUE$(L_1) = 0$ **then return** CHILDAT(L_2, i)
2: **if** VALUE$(L_2) = 0$ **then return** CHILDAT(L_1, i)
3: **return** $(\texttt{OR}, \text{CHILDAT}(L_1, i), \text{CHILDAT}(L_2, i))$

Algorithm 13 VALUE function for `EXTEND`

Require:
Lqdag $L_1(\mathcal{A}')$, set $\mathcal{A} \supseteq \mathcal{A}'$.
Ensure:
Value of the root of $(\texttt{EXTEND}, L_1, \mathcal{A})$.

1: **return** VALUE(L_1)

Algorithm 14 CHILDAT function for `EXTEND`

Require:
Lqdag $L_1(\mathcal{A}')$, set $\mathcal{A} \supseteq \mathcal{A}'$, integer $0 \leq i < 2^{|\mathcal{A}|}$.
Ensure: An lqdag for the i-th child of $(\texttt{EXTEND}, L_1, \mathcal{A})$.

1: $d \leftarrow |\mathcal{A}|$, $d' \leftarrow |\mathcal{A}'|$
2: $m_d \leftarrow$ the d-bits binary representation of i
3: $m_{d'} \leftarrow$ the projection of m_d to the positions in which the attributes of $\mathcal{A}'$ appear in $\mathcal{A}$
4: $i' \leftarrow$ the value in $[0, 2^{d'} - 1]$ corresponding to $m_{d'}$
5: **return** $(\texttt{EXTEND}, \text{CHILDAT}(L_1, i'), \mathcal{A})$

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 Q_F:

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

Assuming some of the trees involved have internal nodes, the super-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 $(\texttt{NOT}, Q_T)$. This implies that $(\texttt{EXTEND}, (\texttt{NOT}, Q_T))$ also yields value 0 in octants 000 and 010. Thus, when function CHILDAT is called on child 000 of Q_F, our 0 is immediately propagated and CHILDAT 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 3 for an illustration). On the other hand, if the value of the child 11 of T is 0, then $(\texttt{EXTEND}, (\texttt{NOT}, Q_T))$ 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 CHILDAT towards 101 in Q_F returns

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

If CHILDAT$((\texttt{EXTEND}, (\texttt{QTREE}, Q_R), \mathcal{A}), 101)$ and CHILDAT$((\texttt{EXTEND}, (\texttt{QTREE}, Q_S), \mathcal{A}), 101)$ 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 $\Diamond$. See Figure 3 for an illustration.

Note that the running time of Algorithm 15 is $O(|Q_F^+|)$. One can then compact Q_F^+ to obtain Q_F, in time $O(|Q_F^+|)$ as well, with a simple bottom-up traversal. 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. To prove this, the introduction of values $\Diamond$ plays a key role.[4]

[4] In an implementation, we could simply use ½ instead of $\Diamond$, without indicating that we are not yet sure

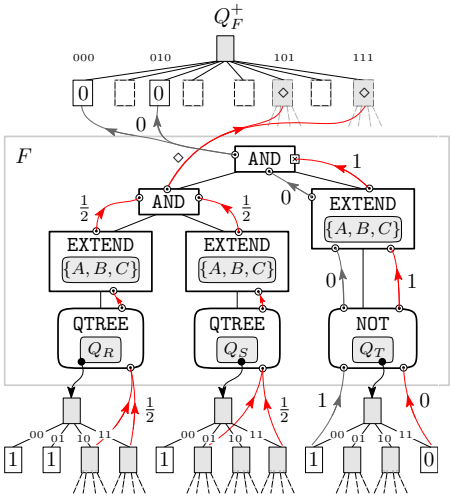

Figure 3 Illustration of the syntax tree of an lqdag for the formula $(R(A,B) \bowtie S(B,C)) \bowtie \overline{T}(A,C)$. The quadtrees Q_R, Q_S, Q_T represent the relations R, S, T, respectively. We show the top values of Q_F^+ on top and of Q_T on the bottom. The gray upward arrows show how the value 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 $\Diamond$ in Q_F^+).

Algorithm 15 SCompletion

Require: An lqdag L_F whose completion represents a formula F over relations with d attributes.
Ensure: The super-completion Q_F^+ of L_F.

1: **if** Value$(L_F) \in \{0, 1\}$ **then return** a leaf with value Value(L_F)
2: **return** an internal node with children
$\Big($SCompletion$\big($ChildAt$(L_F, 0)\big), \ldots,$ SCompletion$\big($ChildAt$(L_F, 2^d - 1)\big)\Big)$

The power of the $\Diamond$ values. Consider again Algorithm 15. The lowest places in L_F where $\Diamond$ values are introduced are the Value of `AND` and `OR` lqdags where both operands have Value $=$ ½. We must then set the Value to $\Diamond$ instead of ½ because, depending on the evaluation of the children of the operands, the Value can turn out to be actually 0 for `AND` or 1 for `OR`. Once produced, a value $\Diamond$ is inherited by the ancestors in the formula unless the other value is 0 (for `AND`) or 1 (for `OR`).

Imagine that a formula F involves n relations $R_1, \ldots, R_n$ represented as quadtrees in dimension d, including no negations. Suppose that we *trim* the quadtrees of $R_1, \ldots, R_n$ by removing all the levels at depth higher than some j (thus making the j-th level the last one) and assuming that the internal nodes at level j become leaves with value 1. We do not attempt to compact the resulting quadtrees, so their nodes at levels up to $j-1$ stay identical and with the same Value. If we now compute Q_F^+ over those (possibly non-pruned) quadtrees, the computation will be identical up to level $j-1$, and in level j every internal node in the original Q_F^+, which had value ½, will now operate over all 1s, and thus will evaluate to 1 because And and Or are monotonic.

Thus, these nodes belong to the output of F over the relations $R_1', \ldots, R_n'$ induced by the trimmed quadtrees (on smaller domains of size $\ell' = 2^j$), with sizes $|R_1|' \leq |R_1|, \ldots, |R_n|' \leq |R_n|$. This would imply, just as in the proof of Theorem 3.6, a bound on the maximum number of nodes in a level of Q_F^+, thus proving the worst-case optimality of the size of Q_F^+ (up to $\log min(\ell, N)$-factors, and factors depending on the query size), and thus the worst-case optimality of Algorithm 15 in data complexity.

However, this reasoning fails when one trims at the j-th level a quadtree Q that appears

that the value is ½: we build Q_F^+ assuming it is, and only make sure later, when we compact it into Q_F.

in an lqdag $L = (\texttt{NOT}, Q)$, because the value 1 of the nodes at the j-th level of Q after the trimming changes to 0 in L. So, to prove that our algorithm is worst-case optimal we cannot rely only on relations obtained by trimming those that appear in the formula. We need to generate new quadtrees for those relations under a NOT operation that preserve the values of the completion of NOT after the trimming. Next we formalize how to do this.

Analysis of the algorithm. Let L_F be an lqdag for a formula F. The *syntax tree* of F is the directed tree formed by the fnodes in F, with an edge from fnode L to fnode L' if L' is an operand of L. The leaves of this tree are always *atomic expressions*, that is, the fnodes, with functors QTREE and NOT, that operate on one quadtree (see Figure 3 again). We say that two atomic expressions L_1 and L_2 are equal if both their functors and operands are equal. For example, in the formula

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

there are three different atomic expressions, $(\texttt{QTREE}, Q_R)$, $(\texttt{QTREE}, Q_S)$, and $(\texttt{QTREE}, Q_T)$, while in $F' = (\texttt{AND}, (\texttt{QTREE}, Q_R), (\texttt{NOT}, Q_R))$ there are two atomic expressions. Notice that in formulas like 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 15 while evaluating a formula F.

▶ **Lemma 4.2.** *Let F be a relational formula represented by an lqdag L_F in dimension d, and let Q_F^+ be the super-completion of F. Let $Q_1, \ldots, Q_n$ be the quadtree operands of the different atomic expressions of F, and $R_1(\mathcal{A}_1), \ldots, R_n(\mathcal{A}_n)$ be the (not necessarily different) relations represented by these quadtrees, respectively. Let M be the maximum number of nodes in a level of Q_F^+. Then, there is a database with relations $R'_1(\mathcal{A}_1), \ldots, R'_n(\mathcal{A}_n)$ of respective sizes $O(2^d|Q_1|), \ldots, O(2^d|Q_n|)$, such that the output of F evaluated over it has size $\Omega(M/2^d)$.*

Proof. Let m_l be the number of nodes in level l of Q_F^+ and j be a level where $M = m_j$ is maximum. We assume that $j > 1$, otherwise $M = O(1)$ and the result is trivial. We first bound the number of nodes with value ½ at the $(j-1)$-th level. By hypothesis, $m_j \geq m_{j-1}$, and since a node in Q_F^+ is present at level j only if its parent at level $j-1$ has value ½, in the $(j-1)$-th level there are at least $m_j/2^d$ nodes with value ½.

Now, let $A_1, \ldots, A_n$ be the atomic expressions of F, and 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. Consider the completion A_i^* of A_i evaluated over Q_i, and the completion $A_i^{*\prime}$ of A_i evaluated over (the possibly non-pruned) Q'_i, for all $1 \leq i \leq n$. If it is always the case that the first $j-1$ levels of A_i^* are respectively equal to the $j-1$ levels of $A_i^{*\prime}$ then we are done. To see why, let $Q_F^{+\prime}$ be the super-completion of F when evaluated over $Q'_1, \ldots, Q'_n$. The first $j-2$ levels of Q_F^+ will be the same as those of $Q_F^{+\prime}$ because the same results of the operations are propagated up from the leaves of the syntax tree of F before and after the trimming. Moreover, in the $(j-1)$-th level $Q_F^{+\prime}$ (its last level) the nodes with value 1 are precisely the nodes with value 1 or ½ in Q_F^+, where we note that: (i) there are at least $m_j/2^d$ of them; and (2) they belong to the output of F over the relations $R'_1, \ldots, R'_n$ represented by $Q'_1, \ldots, Q'_n$.

We know that $|R'_1| \leq |R_1|, \ldots, |R'_n| \leq |R_n|$. However, the values of $R'_1, \ldots, R'_n$ correspond to a smaller universe. This can be remedied by simply appending $(\log \ell - j)$ 0's at the beginning of the binary representation of these values. This would yield the desired result: we have n relations over the same set of attributes as the original ones, with same respective cardinality, and such that when F is evaluated over them the output size is $\Omega(m_j/2^d)$.

However, for atomic expressions of the type $A_i = (\texttt{NOT}, Q_i)$ it is not the case that the first $j-1$ levels of A_i^* coincide with the $j-1$ levels of $A_i^{*\prime}$. Anyway, we can deal with this case: their first $j-2$ levels will coincide, and in the last level, the value of a node present in A_i^* is the negation of the value of the homologous node in $A_i^{*\prime}$. Thus, instead of choosing the quadtree Q_i' that results from trimming Q_i, we choose the quadtree Q_i'' in which the first $j-2$ levels are the same as Q_i', and the $(j-1)$-th level results from negating the value of every node in Q_i'. Note that if we let now $A_i^{*\prime}$ be the completion of A_i evaluated over Q_i'', then the first $j-1$ levels of A_i^* will be exactly same as the $j-1$ levels of $A_i^{*\prime}$. Finally, note that the size of the relation represented by Q_i'' cannot be larger than $2^d|Q_i|$. The result of the lemma follows. ◀

Using the same reasoning as before we can bound the time needed to compute the super-completion Q_F^+ of an lqdag L_F in dimension d involving quadtrees representing $R_1, \ldots, R_n$. Since M is the maximum number of nodes in a level of Q_F^+, the number of nodes in Q_F^+ is at most $M \log \ell$. Now, each node in Q_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 super-completion can be computed in time $O(M \cdot 2^d |F| \log \ell)$. If we use $F(D)^*$ to denote the size of the maximum output of the query F over instances with relations $R_1, \ldots, R_n$ of respective sizes $O(2^d|Q_1|), \ldots, O(2^d|Q_n|)$, then by Lemma 4.2 the query F can be computed in time $O(F(D)^* \cdot 2^{2d}|F| \log \ell)$. This means that the algorithm is indeed worst-case optimal.

The requirement of different atomic expressions is because we need to consider R and Not R as different relations. To see this, consider again our example formula $F' = (\texttt{AND}, (\texttt{QTREE}, Q_R), (\texttt{NOT}, Q_R))$. We clearly have that the answer of this query is always empty, and therefore $|Q_{F'}| = 0$. However, here $|Q_{F'}^+| = \Theta(|R|)$ for every R, and thus our algorithm is worst-case optimal only if we consider the possible output size of a more general formula, $F'' = (\texttt{AND}, (\texttt{QTREE}, Q_R), (\texttt{NOT}, Q_R'))$. This impacts in other operations of the relational algebra. We can write all of them as lqdags, but for some of them we will not ensure their optimal evaluation. For instance, the expression Q_R And (Not $(Q_R$ And $Q_S)$), which expresses the antijoin between R and S, is not optimal because both Q_R and Not Q_R appear in the formula. A way to ensure that our result applies is to require that the atomic expressions (once the NOT operations are pushed down) refer all to different relations.

▶ **Theorem 4.3.** *Let F be a relational formula represented by an lqdag L_F. If the number of different relations involved in F equals the number of different atomic expressions, then Algorithm 15 evaluates F in worst-case optimal time in data complexity.*

Note that this result generalizes Theorem 3.6. 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 $(R \bowtie (S \bowtie T))$.

5 Final Remarks

The evaluation of join queries using qdags provides a competitive alternative to current worst-case optimal algorithms [9, 11, 15, 17, 21]. When compared to them, we find the following time-space tradeoffs.

Regarding space, qdags require only just a few extra words per tuple, which is generally much less than what standard database indexes require, and definitely less than those required by current worst-case optimal algorithms (e.g. [9, 17, 21]). Moreover, in both NPRR [17] and leapfrog [21], 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 (i.e., $d!$ indexes). In contrast, all we need to store is one quadtree per relation, and that works for any query. Even if we resort to the (simpler) k^d-tree representation by Brisaboa et al. [4], the extra space increases by a factor of 2^d bits, which is still considerably less than the alternative of $d!$ standard indexes for any order of variables (e.g., for $d = 10$, $2^d = 1024$, while $d! = 3628800$, i.e., ≈ 3500 times bigger).

Regarding time, the first comparison that stands aside is the $\log(N)$ factor, present in our solution but not in others like NPRR [17] and leapfrog [21]. 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 [17, footnote 3]. This was also noted for leapfrog [21, 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 [11] 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 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 [9, 15]. This is, as far as we are aware, unavoidable: it is the price to pay for using so little space. Note, however, that this factor does not depend on the data, and that it can be compensated by the fact that our native indexes are compressed, and thus might fit entirely in faster memory.

One important benefit of our framework is that answers to queries can be delivered in their compact 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. One could even cache the top half of the (uncompacted) tree containing the answer, and leave the bottom half in the form of lqdags. The upper half, which is used the most, is cached, and the bottom half is computed on demand. Our framework also permits sharing lqdags as common subexpressions that are computed only once. Additionally, we envision two main uses for the techniques presented in this paper. On one hand, one could take advantage of the low storage cost of these indexes, and add them as a companion to a more traditional database setting. Smaller joins and selections could be handled by the database, while multijoins could be processed faster because they would be computed over the quadtrees. On the other hand, we could use lqdags instead, so as to evaluate more expressive queries over quadtrees. Even if some operations are not optimal, what is lost in optimality may be gained again because these data structures allow operating in faster memory levels.

There are several directions for future work. For instance, we are trying to improve our structures to achieve good bounds for acyclic queries (see Appendix A), and we see an opportunity to apply quadtrees in the setting of parallel computation (see, e.g., Suciu [20]). We also comment in Appendix A on bounds for clustered databases, another topic deserving further study.

References

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

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

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

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

5 G. de Bernardo, S. Alvarez-García, N. Brisaboa, G. Navarro, and O. Pedreira. Compact querieable representations of raster data. In *Proc. 20th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 96–108, 2013.

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

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

8 A. Hogan, C. Riveros, C. Rojas, and A. Soto. Extending sparql engines with multiway joins. In *Proc. 18th International Semantic Web Conference (ISWC)*, 2019. To appear.

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

10 M. A. Khamis, H. Q. Ngo, and A. Rudra. Faq: Questions asked frequently. In *Proc. 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS)*, pages 13–28, 2016.

11 M. A. Khamis, H. Q. Ngo, and D. Suciu. What do shannon-type inequalities, submodular width, and disjunctive datalog have to do with one another? In *Proc. 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS)*, pages 429–444, 2017.

12 G. M. Morton. A computer oriented geodetic data base; and a new technique in file sequencing. Technical report, IBM Ltd., 1966.

13 G. Navarro. *Compact Data Structures – A practical approach.* Cambridge University Press, 2016.

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

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

16 H. Q. Ngo, C. Ré, and A. Rudra. Skew strikes back: New developments in the theory of join algorithms. *ACM SIGMOD Record*, 42(4):5–16, 2014.

17 Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*, pages 37–48. ACM, 2012.

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

19 H. Samet. *Foundations of Multidimensional and Metric Data Structures.* Morgan Kaufmann, 2006.

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

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

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

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

A Appendix

A.1 Additional comments on Projections

Including projection in our framework is not difficult: in a quadtree Q storing a relation R with attributes $\mathcal{A}$, one can compute the projection $\pi_{\mathcal{A}'}(R)$, for $\mathcal{A}' \subseteq \mathcal{A}$ as follows. Assume that $|\mathcal{A}| = d$ and $|\mathcal{A}'| = d'$. Then the projection is the quadtree defined inductively as follows. If $\textsc{Value}(Q)$ is 0 or 1 then the projection is a leaf with the same value. Otherwise Q has 2^d children. The quadtree for $\pi_{\mathcal{A}'}(R)$ has instead $2^{d'}$ children, where the i-th child is defined as the $\textsc{Or}$ of all children j of Q 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. For example, computing $\pi_{A_1,A_2}R(A_1, A_2, A_3)$ means creating a tree with four children, resulting of the $\textsc{Or}$ of children 0 and 1, 2 and 3, 4 and 5 and 6 and 7, respectively.

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 [23] to compute it in time $O(|R| + |S| + |R \bowtie S|)$. This is commonly achieved by first computing $\pi_B(R)$ and $\pi_B(S)$, joining them, and then using this join to filter out R and S. Unfortunately, adopting this strategy in our lqdag framework would still give us a quadratic algorithm, even for queries with small output, because after the projection we would need to extend the result again. The same holds for the general Yannakakis' algorithm when computing the final join after performing all necessary semijoins.

More generally, this also rules out the possibility to achieve optimal bounds for queries with bounded treewidth or similar measures. Of course, this is not much of a limitation because one can always compute the most complex queries with our compact representation and then carry out Yannakakis' algorithms on top of these results with standard database techniques, but it would be better to resolve all within our framework. We are currently looking at improving our data structures in this regard.

A.2 Geometric representation and finer analysis

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. [9]. 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 answer of the query. Perhaps more interestingly, the 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 [9], quadtrees provide a particularly natural and compact representation. Each 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. We can understand qdags as a set of gap boxes as well: precisely those

in its completion. Now let $J = R_1 \bowtie \cdots \bowtie R_n$ be a join query over d attributes, and let $R_1^*, \ldots, R_n^*$ denote the extension of each R_i with the attributes of J that are not in R_i. As in Khamis et al. [9], 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.

▶ **Proposition A.1.** *Given query J and 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 optimality results, as discovered [9, 15] 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. It is also worth mentioning that our algorithms 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.

A.3 Better runtime on clustered databases

Quadtrees have been shown to work well in applications such as RDF stores or web graphs, where data points are distributed in clusters [4, 1]. It turns out that combining the analysis described in Section 2 for clustered grids with the technique we used to show that joins are worst-case optimal, results in a better bound for the running time of our algorithms, and a small refinement of the AGM bound itself.

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 of 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^d internal nodes per cluster (where we are in dimension $d = 3$): at this level one can think of the trimmed quadtree as representing a coarser grid of cells of size s^d, 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 3.6 yields a bound for the number of internal nodes at level $\log(\ell/s)$ of the quadtree Q^+ of the output before the compaction step (or, equivalently, of the super-completion of the lqdag of the triangle query): 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 $(c \cdot 2^d)^{3/2}$. 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 $(c \cdot 2^d)^{3/2}$ clusters of size at most s^d. 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 that the size of Q^+ is bounded by $O((c \cdot 2^d)^{3/2} \log \ell + p^{3/2} \log s)$, and therefore the running time of the algorithm is $O\big(((c \cdot 2^d)^{3/2} \log \ell + p^{3/2} \log s) \cdot 2^d\big)$. This is an important reduction in running time 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 $\log \ell$.

To generalize, let us use $D^{c,d}$ as the database "trimmed" to $c \cdot 2^d$ points. The discussion above can be extended to prove the following.

▶ **Proposition A.2.** *Let $J = R_1 \bowtie \cdots \bowtie R_n$ be a full join query, and D a database over schema $\{R_1, \ldots, R_n\}$, with d attributes in total, where the domains of the relations are in $[0, \ell - 1]$, and where the points in each relation are distributed in c clusters of width s. Then Algorithm 4 works in time $O\big((2^{\rho^*(J,D^{c,d})} \log \ell + 2^{\rho^*(J,D)} \log s) \cdot 2^d n\big)$.*