Intersection Joins under Updates

Yufei Tao
CUHK
taoyf@cse.cuhk.edu.hk

Ke Yi
HKUST
yike@cse.ust.hk

September 28, 2021

Abstract

In an *intersection join*, we are given t sets $R_1, ..., R_t$ of axis-parallel rectangles in $\mathbb{R}^d$, where $d \geq 1$ and $t \geq 2$ are constants, and a *join topology* which is a connected undirected graph G on vertices $1, ..., t$. The result consists of tuples $(r_1, ..., r_t) \in R_1 \times ... \times R_t$ where $r_i \cap r_j \neq \emptyset$ for all i, j connected in G. A structure is *feasible* if it stores $\tilde{O}(n)$ words, supports an update in $\tilde{O}(1)$ amortized time, and can enumerate the join result with an $\tilde{O}(1)$ delay, where $n = \sum_i |R_i|$ and $\tilde{O}(.)$ hides a polylog n factor. We provide a dichotomy as to when feasible structures exist: they do when $t = 2$ or $d = 1$; subject to the OMv-conjecture, they do not exist when $t \geq 3$ and $d \geq 2$, regardless of the join topology.

Accepted by Journal of Computer and System Sciences (JCSS).
Keywords: Intersection Joins, Enumeration, Dynamic Updates, Data Structures, OMv-Conjecture

1 Introduction

Let $R_1, R_2, ..., R_t$ be t sets of d-dimensional rectangles (note: all the rectangles in this paper are axis-parallel). An *intersection join* is defined by a *join topology*, which is a connected undirected graph G on vertices $\{1, 2, ..., t\}$. The join result is the set of tuples

$$(r_1, ..., r_t) \in R_1 \times ... \times R_t$$

where $r_i \cap r_j \neq \emptyset$ for all i, j such that G has an edge between i and j. Figure 1 shows a join topology with $t = 3$, for which the join result consists of all $(r_1, r_2, r_3) \in R_1 \times R_2 \times R_3$ satisfying:

$$(r_1 \cap r_2 \neq \emptyset) \wedge (r_2 \cap r_3 \neq \emptyset). \tag{1}$$

Set $n = \sum_i |R_i|$. We will concentrate on *data complexity* by restricting t and d to constants. Ideally, we want to maintain a *feasible* data structure with all of the following guarantees:

- It stores $\tilde{O}(n)$ words, where $\tilde{O}(.)$ hides a polylog n factor.
- It can be updated in $\tilde{O}(1)$ amortized time, when a rectangle is inserted into or deleted from any R_i $(1 \leq i \leq t)$.
- It can be used to enumerate the join result with a *delay* of $\Delta = \tilde{O}(1)$, that is:
  - within Δ time, the algorithm must either report the first result tuple or terminate (when the join result is empty);
  - after reporting a result tuple, within another Δ time, the algorithm must either report a new result tuple or terminate (when the entire result has been reported).

Notice that, if the join result has k tuples, a feasible structure finds all of them in $\tilde{O}(1 + k)$ time.

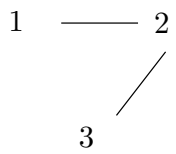

Figure 1: A join topology for $t = 3$

1.1 What Intersection Joins can Do

The most important *non*-natural joins are arguably those with a conjunction of predicates, each of which is defined with the "$\leq$" or "$\geq$" operator. Many of these joins can be modeled as intersection joins. For example,

$$(x, y, z) \quad :- \quad T_1(x), T_2(y, z), y \leq x \leq z$$

is an intersection join between two sets of 1D intervals: $R_1 = \{[x, x] \mid x \in T_1\}$ and $R_2 = \{[y, z] \mid (y, z) \in T_2 \wedge y \leq z\}$. As another example,

$$(w, x, y, z) \quad :- \quad T_1(w, x), T_2(y, z), w \leq y, x \geq z$$

is an intersection join between two sets of 2D rectangles: $R_1 = \{[w, w] \times [x, x] \mid (w, x) \in T_1\}$ and $R_2 = \{(-\infty, y] \times [z, \infty) \mid (y, z) \in T_2\}$.

Outside relational databases, intersection joins find major applications as well:

- For $d = 1$, they are frequent in temporal databases, where a record is associated with a time interval indicating the record's validity period. An intersection join is what is needed to find tuples whose validity periods overlap in a designated manner, and is a crucial operation in many scenarios [8, 9, 11, 18, 27].

- For $d \geq 2$, intersection joins are known under the name *spatial join*. This is a core operation in spatial databases, where each object is associated with a rectangle (typically, the minimum bounding rectangle of a geometric entity such as a line segment, a polygon, a circle, etc.). A spatial join is the key to extracting overlay information from different sets of objects and has received considerable attention (e.g., [5, 22, 25, 26, 28, 32]).

1.2 Related Work

"One-Shot" Intersection Joins. In the *offline* version of our problem, we want to compute the result of an intersection join on $R_1, ..., R_t$. The computation is done only once, namely, there are no updates to worry about. Surprisingly, even this problem has not been well understood, except when the join topology is a *tree*. Willard [29] showed that the problem can be solved in $\tilde{O}(n + k)$ time for any tree topology, where k is the number of result tuples. Whether the offline version can be settled using $\tilde{O}(n + k)$ time for an arbitrary topology is still open, even in 1D space.

Even just for tree topologies, it would be tempting to adapt Willard's algorithm [29] to the dynamic scenario (i.e., our problem). His algorithm in essence processes a tree topology using the "leaf-to-root" semi-join idea that Yannakakis [30] introduced for processing an acyclic natural join. A straightforward adaptation, however, entails either a large update cost of $\tilde{O}(n)$, or an uninteresting delay of $\Delta = \tilde{O}(n)$ in result enumeration. It is not clear how to improve this without introducing new ideas.

View Maintenance. Our problem can be regarded as a variant of incremental view maintenance. Define a *view* W as the set of t-tuples in the join result. We want to "maintain" W incrementally with $\tilde{O}(n)$ space and $\tilde{O}(1)$ time per update. As mentioned, the maintenance is done by storing $R_1, ..., R_t$ in a feasible structure, such that W can be extracted from the structure in $\tilde{O}(|W|)$ time whenever needed.

There are two main challenges in achieving the above goal. First, a join result may have a size up to $O(n^t)$, which rules out the possibility of *materializing* the view if the space must be kept $\tilde{O}(n)$. Second, a single update may change a significant portion of the join result. This makes result materialization an infeasible approach even if the join result has a size of $\tilde{O}(n)$.

Overcoming these challenges is non-trivial even for natural joins (a.k.a., conjunctive queries) . Note that the concept of "feasible structure" is readily extendable to a natural join $R_1 \bowtie ... \bowtie R_t$ (in fact, this definition was explicit in [3]). Finding such structures for natural joins was studied as early as in the 80's of the last century, but with success limited to binary joins [7, 12]. As a breakthrough, Berkholz et al. [3] proved that, for natural joins on $3 \leq t = O(1)$ relations, feasible structures exist if and only if the join is *q-hierarchical*, subject to the *OMv-conjecture* [13] (see also [14] for similar upper bound results). We refer the reader to [3] for the definition of q-hierarchical (this notion will not be needed in our discussion), but state the OMv-conjecture here:

OMv-conjecture [13]. Consider the following *online boolean matrix-vector multiplication* (OMv) problem. An algorithm is given an $n \times n$ boolean matrix $\boldsymbol{M}$, and is allowed to preprocess $\boldsymbol{M}$ arbitrarily in poly(n) time. Then, the algorithm is given a stream of $n \times 1$ boolean vectors $\boldsymbol{v_1}$, $\boldsymbol{v_2}$, ..., $\boldsymbol{v_n}$, and is required to compute $\boldsymbol{Mv_i}$ for each i (the additions and multiplications on the elements of the matrices are performed in the boolean semi ring). In particular, vector $\boldsymbol{v_{i+1}}$ ($i \geq 1$) is fed to the algorithm only after it has correctly output $\boldsymbol{Mv_i}$. The cost is the total time the algorithm takes in computing the n multiplications. The OMv-conjecture states that, no algorithms can solve the problem successfully with probability at least 2/3 in $O(n^{3-\epsilon})$ time, for any constant $\epsilon > 0$.

Partially inspired by the above, recent years have witnessed efforts (see the representative works [16, 17]) studying *non-feasible* structures on natural joins that can provide a good tradeoff between update efficiency and enumeration delay.

None of the above works considered *non-natural* joins. To fill the void, Idris et al. [15] studied how to maintain data structures that can answer conjunctive queries with inequality predicates, and support efficient updates on the participating relations. Like our work, the structure should allow the query result (i.e., a join result) to be enumerated with an $\tilde{O}(1)$ delay, but unlike our work, the structure is permitted to spend $\tilde{O}(n)$ time to support each update, where n is the size of the database (hence, the structure is not feasible).

It is worth mentioning that the form of maintenance discussed above is different from another (perhaps more traditional) branch of incremental view maintenance, which aims at computing the *delta* result changes of a join caused by an update, i.e., find (i) all the new result tuples created by an insertion, and (ii) the existing result tuples removed by a deletion. Indeed, many works in the literature have explicitly focused on this branch; e.g., see [3, 4, 14, 19–21, 31] and the references therein. In fact, feasible structures can be deployed to support the above style of maintenance as well, using a reduction which we explain in Appendix B.

1.3 Contributions

This paper provides a complete dichotomy on when an intersection join admits a feasible structure. Next, we provide an overview of our results and the proposed techniques.

$t = 2$ (Binary Joins). It is a good idea to start with the most fundamental: in our context, a binary intersection join in 1D space ($t = 2$, $d = 1$). For such joins, we can prove:

Theorem 1. *For an intersection join with $d = 1$ and $t = 2$, there is a structure of $O(n)$ space that can be updated in $O(\log n)$ amortized time, and can be used to enumerate the join result with a constant delay.*

The structure is asymptotically optimal in the comparison model of computation (see Appendix A for a proof), and contains just the right amount of sophistication for demonstrating two new ideas that are also applied in some other structures of the paper:

- *Productive list:* One issue in designing a feasible structure is how to enumerate a join result of an exceedingly small size k. As the enumeration can take only $\tilde{O}(1 + k)$ time, when $k \ll n$, we cannot afford to read the whole input. We remedy the issue by marking certain nodes of the structure as "productive": these nodes tell us *where* to look to start reporting result tuples immediately. If $k = 0$, no productive nodes exist, permitting us to finish in constant time.
- *Buffering:* Often times, it would be easier to come up with an intersection join algorithm that runs in $\tilde{O}(1 + k)$ time, but harder to guarantee an $\tilde{O}(1)$ delay. If we *could* always turn

such an algorithm into one with an $\tilde{O}(1)$ delay, designing feasible structures would become considerably easier. We propose a *buffering technique* to make this possible, provided that the algorithm is "aggressive" in reporting. Intuitively, such an algorithm would output most of the result during an early stage, and then possibly remain "quiet" for a long time before reporting the rest.

Our 1D structure can be extended to higher dimensionalities:

Theorem 2. *For any intersection join with $d = O(1)$ and $t = 2$, there is a structure of $\tilde{O}(n)$ space that can be updated in $\tilde{O}(1)$ amortized time, and can be used to enumerate the join result with an $\tilde{O}(1)$ delay.*

$t \geq 3$ and $d \geq 2$. For intersection joins on $t \geq 3$ sets of rectangles, we are able to show that no feasible structures are likely to exist when the dimensionality is at least 2:

Theorem 3. *Unless the OMv-conjecture [13] fails, for any intersection join with $t \geq 3$ and $d \geq 2$ (regardless of the join topology), no structure can offer the following guarantees simultaneously: for some constant $0 < \epsilon < 0.5$, it (i) can be updated in $O(n^{0.5-\epsilon})$ time, and (ii) can be used to enumerate the join result with a delay of $O(n^{0.5-\epsilon})$.*

Our proof is based on a reduction from a negative result established in [3] about the natural join $T_1 \bowtie T_2 \bowtie T_3$ where the three relations have schemas $T_1(X)$, $T_2(X, Y)$, $T_3(Y)$. The fact that this particular natural join "seals the fate" of all intersection joins of $t \geq 3$ and $d \geq 2$ is mildly surprising.

By applying Theorem 3 to *tree* topologies, one can see that our problem is inherently more difficult than its offline version (see Section 1.2) in the following sense. First, recall that Willard [29] gave an offline algorithm that runs in $\tilde{O}(n + k)$ time for any tree topology and any constant dimensionality d. On the other hand, a feasible structure immediately provides an offline algorithm: one can simply perform n insertions and then enumerate the join result. Thus, Theorem 3 points out that no such structures can offer an offline algorithm that matches Willard's solution in running time, when $t \geq 3$ and $d \geq 2$, even if the topology is a tree.

1D Joins with $t \geq 3$ Sets. This last landscape turns out to be the most challenging (perhaps the most exciting). As explained in the preceding paragraph, if a feasible structure exists for any join topology when $d = 1$, then the offline version can be settled in $\tilde{O}(n + k)$ time in 1D space for *all* topologies (not just trees as in [29]) — but whether that is achievable is still open.

We managed to overcome the challenge:

Theorem 4. *For any 1D intersection join with constant $t \geq 3$, there is a structure of $\tilde{O}(n)$ space that can be updated in $\tilde{O}(1)$ amortized time, and can be used to enumerate the join result with an $\tilde{O}(1)$ delay.*

The theorem thus solves the offline intersection join problem in 1D space for *arbitrary* topologies on any constant number t of interval sets. In particular, we guarantee not only a total output time of $\tilde{O}(1 + k)$, but also an $\tilde{O}(1)$ delay as well, provided that $\tilde{O}(n)$ preprocessing time is allowed.

Our structure incorporates a series of new ideas many of which are too detailed for the high-level discussion here, but two particular techniques are notable:

- *Lexicographic ordering:* We conceptually order all the result tuples $(r_1, ..., r_t)$ by concatenating the left endpoints of $r_1, ..., r_t$, and comparing the concatenated sequences lexicographically. The core of our structure is to find the *first* result tuple by this ordering. As the structure

supports fast updates, we can find the entire result efficiently by repeatedly finding the "first" result tuple, after deleting certain tuples appropriately. The deleted tuples are eventually added back into the structure at the end of the join. This turns out to be a powerful method, and plays an indispensable role in our proof of Theorem 4.

- *Recursive topology partitioning:* The second technique we devised is a recursive mechanism for processing a 1D intersection join. The mechanism removes certain vertices from the join topology G, and breaks the remaining parts of G into maximally connected subgraphs. Each subgraph gives rise to a smaller join to be handled by recursion. The correctness of the mechanism is based crucially on numerous characteristics of the problem in 1D space.

2 Preliminaries

Throughout the paper, we consider that the input sets $R_1, ..., R_t$ are in "general position". To state this assumption formally, take a rectangle $r \in \bigcup_i R_i$. If the projection of r onto dimension $j \in [1, d]$ is an interval $[x_1, x_2]$, we say that r *defines* the coordinates x_1 and x_2 on dimension j. The *general position assumption* says that every coordinate of any dimension is defined by at most one rectangle in $\bigcup_i R_i$. The assumption allows us to focus on explaining the new ideas behind our techniques. Removing the assumption can be done with standard tie-breaking techniques (see, e.g., [6]), and does not affect any of our claims.

2.1 Binary Search Tree (BST)

Even though the BST is a rudimentary structure, it can be described in multiple ways, whose differences are usually subtle, but can cause ambiguity when one needs to design new structures by augmenting BSTs. Next, we clarify the BST assumed in this paper and take the opportunity to define some relevant concepts and notations.

Let $\mathcal{S}$ be a set of n values in $\mathbb{R}$. A BST $\mathcal{T}$ on $\mathcal{S}$ is a binary tree with the following properties:

- Each leaf stores a distinct element of $\mathcal{S}$, and conversely, every element of $\mathcal{S}$ is stored at a leaf.
- Every internal node u has two child nodes. It stores a value — called the *search key* of u and denoted as $key(u)$ — which equals the smallest element of its right subtree.
- For each internal node u, all the elements stored in its left (or right) subtree must be less than (or at least, resp.) $key(u)$.

We conceptually associate each node u of $\mathcal{T}$ with a *slab* $\sigma(u)$, which is a semi-open interval defined as follows. If u is a leaf storing an element $p \in \mathcal{S}$, then $\sigma(u) = [p, p')$ where p' is the successor of p in $\mathcal{S}$; in the special case where p is already the largest element in $\mathcal{S}$, $\sigma(u) = [p, \infty)$. If u is an internal node, $\sigma(u)$ is the union of the slabs of its child nodes.

For each node u of $\mathcal{T}$, we use $\mathcal{T}_u$ to represent the subtree rooted at u, and define its *subtree size* — denoted as $|\mathcal{T}_u|$ — as the number of leaf nodes in $\mathcal{T}_u$.

2.2 The Interval Tree

Let $\mathcal{R}$ be a set of intervals in $\mathbb{R}$. Next, we describe what is an *interval tree* [10, 23] on $\mathcal{R}$.

Let $\mathcal{T}$ be a BST on the set of endpoints of the intervals in $\mathcal{R}$. Associate each node u in $\mathcal{T}$ with a *stabbing set* — which we denote as $stab(u)$ — including all and only intervals $r \in \mathcal{R}$ with the

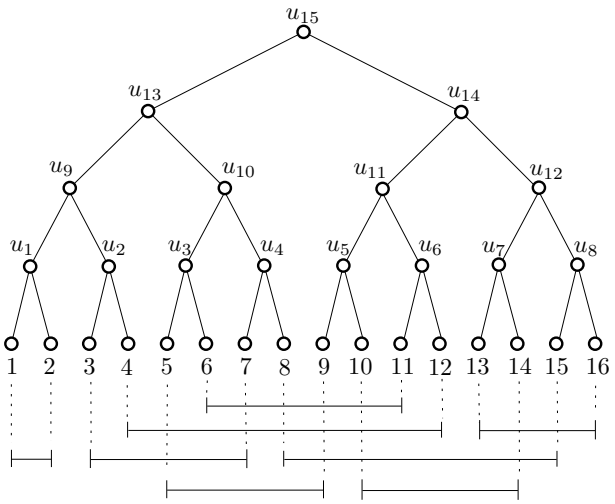

Figure 2: A BST on the endpoints of 8 intervals

property that u is the *highest* node in $\mathcal{T}$ whose search key is covered by r. At u, $stab(u)$ is stored in two separate lists: one sorted by the left endpoints of the intervals therein, and the other by their right endpoints. This completes the definition of the interval tree.

Example. Suppose that $\mathcal{R} = \{[1,2], [3,7], [4,12], [5,9], [6,11], [8,15], [10,14], [13,16]\}$. Figure 2 shows a BST on the endpoints of the 8 intervals.

Consider the root u_{15}. It has a search key $key(u_{15}) = 9$ and a stabbing set $stab(u_{15}) = \{[4,12], [5,9], [6,11], [8,15]\}$. Similarly, one can verify that $stab(u_1) = \{[1,2]\}$, $stab(u_{13}) = \{[3,7]\}$, and $stab(u_{14}) = \{[10,14],[13,16]\}$. □

We will use $stab^{<}(u)$ to represent the set of intervals stored in the stabbing sets that are in the left subtree of u. Define $stab^{>}(u)$ analogously with respect to the right subtree of u. The following facts are rudimentary:

- Every interval in $\mathcal{R}$ belongs to exactly one stabbing set.
- For each node u, $|stab^{<}(u)| + |stab(u)| + |stab^{>}(u)|$ can never exceed $|\mathcal{T}(u)|$ because the endpoints of each interval in $stab^{<}(u) \cup stab(u) \cup stab^{>}(u)$ must be stored at nodes in the subtree of u.

2.3 Weight-Balancing Lemmas for Updates

2.3.1 Weight-Balancing on the Interval Tree

The interval tree serves as the base of several structures proposed in this paper. As we will see, our structures will associate each node u in an interval tree $\mathcal{T}$ with an additional secondary structure, denoted as Γ_u. We want to avoid a full-blown description on how to *update* the resulting interval tree (i.e., augmented with all the secondary structures). There are two reasons. First, the challenges are to figure out how data should be *organized* in Γ_u, as opposed to how to update Γ_u. Second, unfolding all the details of updating would force us to ramble on many standard techniques related to *weight balancing*. The reader would find it rather tedious and unrewarding to plow through all that technical content.

Fortunately, we are able to find a "middle ground" to avoid most of the details, and yet allow the reader to verify the correctness of our algorithms in a (much) lighter way. This is achieved by

extracting the key properties that Γ_u needs to have, for the overall augmented interval tree to have the desired update efficiency.

To explain, again let $\mathcal{R}$ be the underlying set of intervals on which the interval tree $\mathcal{T}$ is built, and set n to the number of nodes in $\mathcal{T}$. Fix any node u in $\mathcal{T}$. If u is an internal node with child nodes v_1, v_2, we will assume that Γ_{v_1} and Γ_{v_2} are both ready. We prescribe four properties $P1$-$P4$ that need to be satisfied by Γ_u:

- ***P1:*** When an interval $r = [x, y] \in \mathcal{R}$ with $y < key(u)$ is inserted or deleted in $stab^{<}(u)$, we can update Γ_u in $f_1(n)$ amortized time, for some function f_1.
- ***P2:*** Given an interval $r \in \mathcal{R}$ with $x \leq key(u) \leq y$ is inserted or deleted in $stab(u)$, we can update Γ_u in $f_2(n)$ amortized time, for some function f_2.
- ***P3:*** Given an interval $r = [x, y] \in \mathcal{R}$ with $key(u) < x$ is inserted or deleted in $stab^{>}(u)$, we can update Γ_u in in $f_3(n)$ amortized time, for some function f_3.
- ***P4:*** Γ_u can be constructed in $f_4(|\mathcal{T}_u|)$ time under the condition that, the intervals in $stab(u)$ have been sorted in two separate lists: one by left endpoint, and the other by right endpoint.

When Γ_u meets the above requirements, we have following guarantee:

Lemma 5. *The augmented interval tree $\mathcal{T}$ can be updated in*

$$O\left(\log n \cdot \Big(1 + f_1(n) + f_3(n)\Big) + f_2(n) + \frac{\log n \cdot f_4(n)}{n}\right)$$

amortized time when an interval is inserted or deleted in $\mathcal{R}$.

Proof. This is a corollary of the results in [2] (see also [24]). □

2.3.2 Weight-Balancing on the BST

Next, we mention another result similar to Lemma 5 that is pertinent to BSTs. Let $\mathcal{T}$ be a BST on a set $\mathcal{S}$ of n values in $\mathbb{R}$. Suppose that we associate each node u of $\mathcal{T}$ with a secondary structure Γ_u having the following guarantees:

- When an element is inserted/deleted in the subtree $\mathcal{T}_u$ of u, Γ_u can be updated in $\tilde{O}(1)$ amortized time.
- Γ_u can be reconstructed in $\tilde{O}(|\mathcal{T}_u|)$ time.

Then, we have:

Lemma 6. *$\mathcal{T}$ can updated in $\tilde{O}(1)$ amortized time per insertion and deletion in $\mathcal{S}$.*

Proof. This is a corollary of the results in [2] (see also [24]). □

2.4 A Result from Computational Geometry

Let R be a set of n rectangles in $\mathbb{R}^d$ for some constant dimensionality d. Each rectangle in R is associated with a *weight* drawn from some ordered domain. Given a query rectangle q, a *range min* query returns the rectangle in R with the smallest weight, among all the rectangles in R that intersect with q. The result below is well-known:

Lemma 7. *We can store R in a structure of $\tilde{O}(n)$ space that answers any range min query in $\tilde{O}(1)$ time. The structure can be updated in $\tilde{O}(1)$ amortized time per insertion or deletion in R.*

Proof. Achievable in many ways; see, for example, [1]. □

3 Multi-Way Joins with ≥ 2 Dimensions

Let us start with our negative result. In this section, we will concentrate on intersection joins on $t \geq 3$ sets $R_1, ..., R_t$ of rectangles in $\mathbb{R}^d$ where $d \geq 2$. We will show that, subject to the OMV-conjecture, no feasible structures can exist.

A Natural-Join Result of [3]. Consider three relations with schema $T_1(X)$, $T_2(X, Y)$, and $T_3(Y)$ (each relation is under the *set* semantics, i.e., no duplicate tuples). We refer to $T_1 \bowtie T_2 \bowtie T_3$ as a *wedge natural join*.

Suppose that the values of attributes X and Y are integers in $[1, D]$, i.e., the domain size of each attribute is D. We want to incrementally maintain a feasible structure, that is, one that supports an update in $\tilde{O}(1)$ time, and can be used to enumerate the result of $T_1 \bowtie T_2 \bowtie T_3$ with a small delay Δ.

Lemma 8. ([3]) *For wedge natural joins, subject to the OMv-conjecture [13], no structure can offer the following guarantees simultaneously: for some constant $0 < \epsilon < 1$, it (i) can be updated in $O(D^{1-\epsilon})$ time, and (ii) admits the join result to be enumerated with a delay of $O(D^{1-\epsilon})$.*

The above lemma holds regardless of the space of the structure.

Hardness of the "Wedge" Topology. Consider again the topology shown in Figure 1. We will refer to the join with this topology as a *wedge intersection join*. Recall that it describes an intersection join on R_1, R_2, and R_3 that returns all $(r_1, r_2, r_3) \in R_1 \times R_2 \times R_3$ satisfying the conditions in (1). We will show that the existence of any feasible structures on such joins in 2D space will defy Lemma 8.

Given the sets $T_1(X), T_2(X, Y)$, and $T_3(Y)$ participating in the wedge natural join, we construct three sets of rectangles in two-dimensional space as follows:

$$\begin{aligned} R_1 &= \{x \times (-\infty, \infty) \mid x \in T_1\} \\ R_2 &= \{(x, y) \mid (x, y) \in T_2\} \\ R_3 &= \{(-\infty, \infty) \times y \mid y \in T_3\}. \end{aligned}$$

Note that the rectangles in all three sets are degenerated: each rectangle in R_1 (or R_3) is a line perpendicular to dimension 1 (or 2, resp.), whereas each rectangle in R_2 is a point. Our construction guarantees:

Proposition 1. *Tuples $x \in T_1$, $(x, y) \in T_2$, and $y \in T_3$ make a pair (x, y) in the result of $T_1 \bowtie T_2 \bowtie T_3$ if and only if (r_1, r_2, r_3) is in the result of the wedge intersection join on R_1, R_2, and R_3, where r_1 is the rectangle $x \times (-\infty, \infty)$ in R_1, r_2 is the rectangle (x, y) in R_2, and r_3 the rectangle $(-\infty, \infty) \times y$ in R_3.*

Suppose that, we are given a wedge-intersection-join structure which can be updated in $U(n)$ time, and be used to enumerate the join result in $\Delta(n)$ time, where $n = |R_1| + |R_2| + |R_3|$. By our construction, the sizes of R_1 and R_2 are at most D, while that of R_2 is at most D^2. It thus follows from Lemma 8 that, subject to the OMv-conjecture, $U(O(D^2)) = O(D^{1-\epsilon})$ and $\Delta(O(D^2)) = O(D^{1-\epsilon})$ cannot hold at the same time. Combining this with $n \leq D^2$ shows:

Lemma 9. *For wedge intersection joins, subject to the OMv-conjecture [13], no structure can offer the following guarantees simultaneously: for some constant $0 < \epsilon < 1$, it (i) can be updated in $O(n^{0.5-\epsilon})$ time, and (ii) admits the join result to be enumerated with a delay of $O(n^{0.5-\epsilon})$.*

Before proceeding, let us point out a property of our construction. Consider the *triangle topology* that has an edge between i, j for all $1 \leq i < j \leq 3$. On the constructed R_1, R_2, R_3, a tuple (r_1, r_2, r_3) is in the join result under the wedge topology *if and only if* it is in the joint result under the triangle topology. This is a crucial property that we rely on to extend the hardness result to arbitrary intersection joins with $t \geq 3$ and $d \geq 2$, as shown next.

Hardness of $t \geq 3$ and $d \geq 2$. Consider an arbitrary intersection join with topology G with $t \geq 3$ vertices. Since G is connected, it must have at least one vertex with two edges. Without loss of generality, assume that G contains an edge between 1 and 2, and an edge between 2 and 3 (otherwise, rename the input sets).

Suppose that we are given a feasible structure for G in two-dimensional space. We can use the structure to maintain the result of the 2D wedge intersection join on R_1, R_2, R_3 that we constructed earlier. For this purpose, we add $t-3$ dummy input sets $R_4, R_5, ..., R_t$, each of which contains only a single rectangle, which is simply $\mathbb{R}^2$ (i.e., the whole data space). The dummy input sets are never updated. It is clear that (r_1, r_2, r_3) is in the result of the wedge intersection join if and only if

$$(r_1, r_2, r_3, \underbrace{\mathbb{R}^2, ..., \mathbb{R}^2}_{t-3})$$

is in the result of the (constructed) t-way intersection join with topology G. Note that this is true no matter whether G has an edge between 2 and 3, due to the property mentioned below Lemma 9.

Finally, the absence of efficient structures when $t \geq 3$ and $d = 2$ implies the same when $t \geq 3$ and $d > 2$ (by adding dummy dimensions). We now conclude the proof of Theorem 3.

4 Binary Joins

Having explained what cannot be done, in the rest of the paper we will focus on what *can* be done. We will prove the existence of feasible structures in all the *other* scenarios, namely, either $t = 2$ (binary joins) or $d = 1$ (1D joins).

This section will focus on $t = 2$. First, Section 4.1 will present the structure of Theorem 1, which as mentioned solves 1D binary joins optimally in the comparison model. Then, Section 4.2 will extend our solutions to constant dimensionalities $d \geq 2$.

4.1 An Optimal 1D Structure

4.1.1 Interval-Point Joins

In the 1D version, R_1 and R_2 are two sets of intervals in $\mathbb{R}$. We will first deal with a special instance of the problem where every interval of R_2 degenerates into a real value, i.e., a point. For clarity, let us denote R_1 simply as R, and use P to represent the set of values in R_2. The join result can now be re-defined in a simpler manner: it reports all $(r, p) \in R \times P$ satisfying $p \in r$. We will refer to this as the *interval-point join*.

Structure. Build an interval tree $\mathcal{T}$ on $R \cup P$, regarding each point in P as a degenerated interval.

As defined in Section 2.2, each node u of $\mathcal{T}$ is associated with a stabbing set $stab(u)$. Denote by $R(u)$ the set of intervals in $stab(u)$ from R. Sort $R(u)$ in two separate lists: the first by left endpoint and the other by right endpoint.

At each internal node u, keep:

- A *left pilot value*, which is the largest point of P stored at a leaf in the left subtree of u;

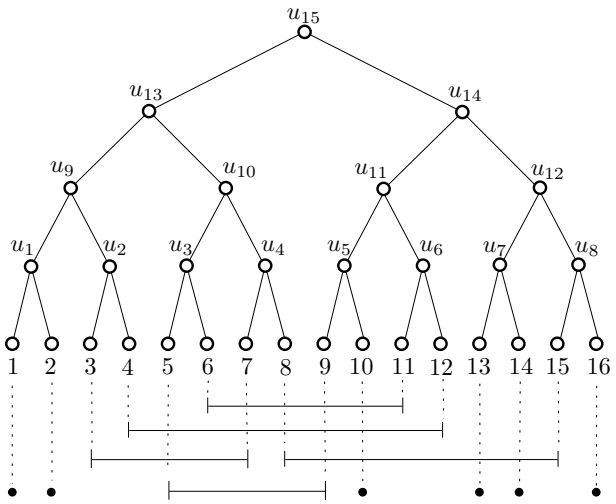

Figure 3: Illustration of the interval-point-join structure

- A *right pilot value*, which is the smallest point of P stored at a leaf in the right subtree of u.

We say that an internal node u is *productive* if $R(u)$ has at least one interval r that covers at least one point in P. Whether u is productive can be decided in constant time as follows:

- Find the interval $[x, y] \in R(u)$ with the smallest left endpoint x. Decide u as productive, if x is no greater than the left pilot value of u.
- Otherwise, find the interval $[x, y] \in R(u)$ with the largest right endpoint y. Decide u as productive, if y is no less than the right pilot value of u.
- Otherwise, decide u as non-productive.

We link up all the productive nodes with a doubly linked list $\mathcal{L}$ (ordering does not matter), referred to as the *productive list*.

Finally, we keep P in a sorted list Σ_P (managed by a BST). Recall that each value $p \in P$ is stored at a leaf u in $\mathcal{T}$. We keep a *cross pointer* from u to the position of p in Σ_P. Remember that p may also be stored as a pilot value at several internal nodes u' in $\mathcal{T}$ as well. We also keep a cross pointer from each such u' to the position of p in Σ_P.

The overall space consumption of our structure is clearly $O(n)$.

Example: Figure 3 shows a BST created on $R = \{[3, 7], [4, 12], [5, 9], [6, 11], [8, 15]\}$, and $P = \{1, 2, 10, 13, 14, 16\}$.

For the root u_{15}, $R(u_{15}) = \{[4, 12], [5, 9], [6, 11], [8, 15]\}$. It has a left pilot value 2, and a right pilot value 10. It is productive because $[4, 12] \in R(u_{15})$ covers a value in P. On the other hand, node u_{13} — with $key(u_{13}) = 5$, $R(u_{13}) = \{[3, 7]\}$, left pilot value 2, and no right pilot value — is non-productive.

Point 10 is stored as the right pilot value of u_{15}, u_5, and as the left pilot value of u_{14}, u_{11}. Hence, each of u_{15}, u_{14}, u_{11}, and u_5 keeps a cross pointer to the position of 10 in Σ_P (the sorted list of P). □

Join Result Enumeration. If $\mathcal{L}$ is empty, we finish immediately, declaring that the join result is empty. The time in this case is constant.

```
enumerate(u)
1.  p_left ← the left pilot value at u
2.  Σ_left ← the list that sorts R(u) in ascending order of
    left endpoint
3.  r ← the first interval in Σ_left
4.  repeat
5.      if p_left ∈ r then
6.          report all such result pairs (r, p) that are produced by r
            and a point p in the left subtree of u
            /* use Σ_P for this purpose; see main texts */
7.          r ← the next interval in Σ_left
8.      else break
9.  until r = null, i.e., Σ_left has been exhausted

10. p_right ← the right pilot value at u
11. Σ_right ← the list that sorts R(u) in descending order of
    right endpoint
12. r ← the first interval in Σ_right
13. repeat
14.     if p_right ∈ r then
15.         report all such result pairs (r, p) that are produced by r
            and a point p in the right subtree of u
            /* use Σ_P for this purpose; see main texts */
16.         r ← the next interval in Σ_right
17.     else return
18. until r = null
```

Figure 4: Enumerating the result pairs at a productive node

Otherwise, for each productive node $u \in \mathcal{L}$, we use the algorithm in Figure 4 to report all result pairs (r, p) satisfying $r \in R(u)$. The algorithm does so with a constant delay (as will be explained shortly) and guarantee outputting at least one pair (by definition of productive node). Because all the productive nodes have been *explicitly* stored in $\mathcal{L}$, we can run the algorithm on every node in $\mathcal{L}$ to report the *entire* query result with a constant delay. No result pair (r, p) can be missed because the interval r must reside in the $R(u)$ of exactly one productive node u.

The algorithm of Figure 4 has two parts: (i) Lines 2-9, which find such (r, p) where p is in the left subtree of u, and (ii) Lines 10-18, which find such (r, p) where p is in the right subtree of u. Due to symmetry, we will discuss only the first part.

After obtaining at Line 1 the left pilot value p_{left} at u, the algorithm (Lines 2-9) processes the intervals of $R(u)$ in ascending order of left endpoint. To explain how, let $r \in R(u)$ be the interval being processed. If r does not cover p_{left}, we are sure that, for any $r' \in R(u)$ that has *not* been processed yet (that is, the left endpoint of r' is greater than that of r), r' cannot make a result pair with p_{left} and — due to the definition of p_{left} — cannot make a result pair with any point in the left subtree of u. In this case, we move on to the second part of the algorithm starting at Line 10.

If, on the other hand, r *does* cover p_{left}, we (at Line 6) find all those points p that (i) are in the left subtree of u and (ii) are covered by r. Every such p makes a result pair with r. A constant delay can be ensured by resorting to the sorted list Σ_p. First, use a cross pointer stored at u to find the position of p_{left} in Σ_p. Then, scan Σ_p from p_{left} in *descending* order until seeing the first point

that falls outside r.

Example. Let us illustrate the algorithm using Figure 3. Node u_{15} is the only productive node. At Line 1, $p_{left} = 2$. Then, we scan $R(u_{15})$ in this order: $[4, 12], [5, 9], [6, 11], [8, 15]$, starting at Line 3 with $r = [4, 12]$. At Line 4, we find that r does not cover p_{left}. The scan is therefore aborted; and the execution jumps to Line 10.

After setting $p_{right} = 10$ at Line 10, we scan $R(u_{15})$ in this order: $[8, 15], [4, 12], [6, 11], [6, 9]$, starting with $r = [8, 15]$ (Line 12). After seeing at Line 14 that r contains p_{right}, we extract all the points $p \in P$ such that $p \geq p_{right}$ and p is covered by r. There are 3 such points: 10, 13, 14. They can be found by scanning Σ_P in ascending order from $10 = p_{right}$.

Next, r is moved to the next interval $[4, 12]$ in $R(u_{15})$, and processed in the same fashion. The rest of the execution is omitted. □

Update. Thanks to Lemma 5, it becomes much easier to explain why our structure can be updated in $O(\log n)$ amortized time per insertion and deletion.

Given a node u of $\mathcal{T}$, regard the following together as its secondary structure Γ_u:

- The two sorted lists of $R(u)$, i.e., one sorted by left endpoint, and the other by right endpoint;
- (Only if u is an internal node) its left and right pilot values, and the cross pointers associated with those values;
- (Only if u is a leaf node and stores a point $p \in P$) the cross pointer associated with p.

For an internal node u, its pilot values (and the cross pointers) can be obtained from its child nodes v_1, v_2 in $O(1)$ time, assuming that $\Gamma_{v_1}, \Gamma_{v_2}$ are both ready. Thus, with respect to the properties $P1$-$P4$ prescribed in Section 2.3.1, it is straightforward to achieve: $f_1(n) = f_3(n) = O(1)$, $f_2(n) = O(\log n)$, and $f_4(|\mathcal{T}_u|) = O(|\mathcal{T}_u|)$. By Lemma 5, $\mathcal{T}$ (augmented with Γ_u) can be maintained in $O(\log n)$ amortized time per update.

It remains to explain how to modify the productive list $\mathcal{L}$. This can be "piggybacked" on the updates on $\mathcal{T}$. In general, whenever the secondary structure Γ_u of a node u is affected by an update, one can spend $O(1)$ time to determine the current productive status of u, and insert/delete u in $\mathcal{L}$ (remember that the ordering in $\mathcal{L}$ does not matter). Therefore, the maintenance of $\mathcal{L}$ cannot be more expensive than maintaining $\mathcal{T}$.

4.1.2 Intersection Joins

We now return to the intersection join problem with $d = 1$ and $t = 2$ where the inputs are two sets R_1 and R_2 of intervals.

For two intervals r_1 and r_2, if $r_1 \cap r_2 \neq \emptyset$, then either r_1 covers at least an endpoint of r_2, or r_2 covers at least an endpoint of r_1. This suggests that the problem can be reduced to four interval-point joins. Specifically, the first (or second) interval-point join sets R to R_1 and P to the set of left (or right, resp.) endpoints of the intervals in R_2, while the other two interval-point joins are defined analogously by reversing the roles of R_1, R_2. Each pair (r_1, r_2) in the result of the original intersection join is output by at least one interval-point join.

We, therefore, maintain four structures of Section 4.1.1, one for each interval-point join. The space consumption and the update cost apparently remain as $O(n)$ and $O(\log n)$ amortized, respectively.

To obtain the result of the intersection join, we enumerate the result of each of the four interval-point structures in tandem. However, two issues arise:

- How to avoid reporting the same pair twice?
- How to ensure a constant delay? Note that even though enumerating the result of each interval-point join guarantees a constant delay, it does *not* directly imply a constant delay on the intersection join. The reason is that result pairs from an interval-point join may have already been found by an earlier interval-point join. In the worst case, the result pairs of an interval-point join may have all been found, thus forcing its enumeration algorithm to incur a long delay without reporting any *new* pairs.

For the first issue, it suffices to adopt a consistent policy regarding which interval-point join should report a pair (r_1, r_2). For example, suppose that r_1 covers both endpoints of r_2. We may follow the policy that in this case only the first interval-point join (i.e., $R = R_1$ and P includes the left endpoints of the intervals in R_2) should report it. The pair is simply ignored when discovered by another interval-point join.

To resolve the second issue, we introduce a *buffering technique*. We actually aim at achieving a more general purpose, which has been briefly described in Section 1.3. Formally, suppose that we are given an algorithm A that does not guarantee a short delay in enumerating the join result, but has the following α-*aggressive* property:

> For any integer $x \geq 1$, after running for an x amount of time, A definitely has found $\lfloor x/\alpha \rfloor$ distinct result tuples.

We remind the reader that, in the RAM model, the *running time* is defined as the number of *atomic operations* (i.e., operations each taking one unit of time, e.g., addition, multiplication, comparison, accessing a memory word, etc.) performed. The above property essentially says that A must have found $\lfloor x/\alpha \rfloor$ result tuples after x atomic operations, for all $x \geq 1$.

Our buffering technique ensures:

Lemma 10. *Given an α-aggressive algorithm A for a join, we can design an algorithm with a delay of at most α.*

Proof. We run A with a buffer, which includes all the result pairs that have been found, but not yet reported. Divide the overall execution A into *epochs*, each consisting of α atomic operations. We keep counting the number of atomic operations performed, and report a pair from the buffer at the end of each epoch.

To prove that the above strategy works, we need to show that the buffer is never empty at the end of each epoch. Consider the end of the i-th epoch ($i \geq 1$). By α-aggressiveness, A must have found at least $\alpha \cdot i/\alpha = i$ distinct result pairs. This completes the proof. □

Let us now go back to our algorithm that runs the four interval-point joins sequentially. As mentioned, each interval-point join ensures a delay $\Delta = O(1)$; and the four interval-point joins perform in total at most $c(1 + k)$ atomic operations, for some constant $c \geq \Delta$. Next, we argue that this algorithm is $8c$-aggressive. This, together with Lemma 10, will complete the proof of Theorem 1.

Suppose that there exists an integer $x \geq 1$ such that, after x atomic operations, the algorithm has found less than $\lfloor x/(8c) \rfloor$ result pairs. As each pair can be reported at most 4 times, strictly less than $x/(2c)$ result pairs — counting duplicate ones — have been found. Thus, the delay before at least one pair must be strictly larger than $\frac{x}{x/2c} = 2c$. However, since all interval-point joins ensure a delay at most Δ, our in-tandem algorithm should find a (new or duplicated) pair with a delay at most $2\Delta \leq 2c$, thus creating a contradiction.

4.2 A Multi-Dimensional Structure

Next, we discuss intersection joins on $t = 2$ sets R_1, R_2 of rectangles in $\mathbb{R}^d$ with a constant dimensionality d.

4.2.1 Dominance Joins

We say that a rectangle r is *d-sided* if it has the form $(-\infty, x_1] \times (-\infty, x_2] \times ... \times (-\infty, x_d]$. This section focuses on a special instance of the problem — referred to as *dominance join* — where all the rectangles in R_1 are d-sided, and all the rectangles in R_2 degenerate into points. For convenience, we rename R_1 as R, and R_2 as P; the join result contains all $(r, p) \in R \times P$ where r contains p.

Set $n = |R| + |P|$. We will design a feasible structure with a recursive approach. The base case is $d = 1$ and has been resolved in Theorem 1 (particularly, Section 4.1.1). Assuming the availability of a feasible structure for $(d-1)$-dimensional dominance joins, next we will describe how to achieve the purpose in d-dimensional space.

Structure. We will refer to dimension 1 the *x-dimension*. Accordingly, the *x-range* of a rectangle r is the projection of r on the first dimension.

Create a BST $\mathcal{T}$ on the set of values that includes (i) the endpoints of the x-ranges of the rectangles in R, and (ii) the x-coordinates of the points in P. We assign each rectangle of R and each point of P to $O(\log n)$ nodes in $\mathcal{T}$ as follows:

- For a rectangle $r \in R$, let $(-\infty, x]$ be its x-range. Descend the path Π from the root of $\mathcal{T}$ to the leaf storing x_1. Every time we go into the right child at some node u on Π, we assign r to the left child of u.
- A point $p \in P$ is assigned to all the proper ancestors of the leaf storing the x-coordinate of p.

Denote by $R_u \subseteq R$ the set of rectangles assigned to a node u of $\mathcal{T}$, and by $P_u \subseteq P$ the set of points assigned to u. The projection of each $r \in R_u$ (or $p \in P_u$) onto dimensions 2, 3, ..., d defines a $(d-1)$-dimensional rectangle (or point, resp.). Clearly, for any $r \in R_u$ and any $p \in P_u$, the x-range of r covers the x-coordinate of p. Hence, r contains p if and only if the $(d-1)$-dimensional rectangle defined by r contains the $(d-1)$-dimensional point defined by p. We denote by R'_u the set of $(d-1)$-dimensional rectangles obtained from R_u, and by P'_u the set of $(d-1)$-dimensional rectangles obtained from P_u.

Motivated by this, we associate u with a secondary structure Γ_u, which is a $(d-1)$-dimensional dominance-join structure on R'_u and P'_u. Node u is *productive* if the $(d-1)$-dimensional join on R'_u and P'_u returns a non-empty result. Because Γ_u is a feasible structure, whether u is productive can be decided in $\tilde{O}(1)$ time. All the productive nodes are collected into a *productive list* $\mathcal{L}$.

The size of Γ_u is $\tilde{O}(|\mathcal{T}_u|)$ by the inductive assumption. Therefore, our structure uses $\tilde{O}(n)$ space overall.

Reporting the Join Result. For each node u in $\mathcal{L}$, use Γ_u to report the result of the $(d-1)$-dimensional join on R'_u and P'_u with an $\tilde{O}(1)$ delay.

Observe that any (r, p) in the result of the original d-dimensional join is reported by exactly one $(d-1)$-dimensional join. Specifically, suppose that $(-\infty, x]$ is the x-range of r. Let Π be the path in $\mathcal{T}$ from the root to the leaf of x, and z be the leaf in $\mathcal{T}$ storing the x-coordinate of p. Set node u to the lowest ancestor of z on Π. Then, the pair (r, p) is reported at the left child of u.

We therefore achieve an $\tilde{O}(1)$ delay overall.

Update. For each node u of $\mathcal{T}$, whenever a rectangle (or point) inserted/deleted from R_u (or P_u), we inserted/deleted it in the $(d-1)$-dimensional structure Γ_u, which takes $\tilde{O}(1)$ time by the inductive assumption. Furthermore, Γ_u can be reconstructed by simply re-inserting all the rectangles in R'_u and P'_u, which by the inductive assumption takes $\tilde{O}(|\mathcal{T}_u|)$ time. It immediately follows from Lemma 6 that our structure can be updated in $\tilde{O}(1)$ time (both conditions in Section 2.3.2 have been satisfied).

We now have officially established the claim that any dominance join of a fixed dimensionality d admits a feasible structure.

4.2.2 Intersection Joins

We now attend to the d-dimensional intersection join between two sets R_1 and R_2 of rectangles. It turns out that, as shown below, such a join can be converted to $4^d = O(1)$ dominance joins, each of which has dimensionality at most $3d = O(1)$.

Consider two intersecting rectangles $r \in R_1$ and $r' \in R_2$. Fix a dimensionality $i \in [1, d]$. Let $[x_i, y_i]$ and $[x'_i, y'_i]$ be the projections of r and r' on this dimension, respectively. If we look at the permutation that sorts the four coordinates in ascending order, there are 4 possible permutations:

- x_i, x'_i, y_i, y'_i
- x_i, x'_i, y'_i, y_i
- x'_i, x_i, y_i, y'_i
- x'_i, x_i, y'_i, y_i.

We can enforce each of the above permutations using the conjunction of at most 3 conditions of the form "$a \in (-\infty, b]$". Specifically, the permutation x_i, x'_i, y_i, y'_i is enforced by:

$$x_i \in (-\infty, x'_i] \wedge -y_i \in (-\infty, -x'_i] \wedge y_i \in (-\infty, y'_i].$$

The above is equivalent to requiring that the 3D point $(x_i, -y_i, y_i)$ be covered by the 3-sided rectangle $(-\infty, x'_i] \times (-\infty, -x'_i] \times (-\infty, y'_i]$. Likewise, the permutation x_i, x'_i, y'_i, y_i is enforced by

$$x_i \in (-\infty, x'_i] \wedge -y_i \in (-\infty, -y'_i].$$

which is equivalent to requiring that the 2D point $(x_i, -y_i)$ be covered by the 2-sided rectangle $(-\infty, x_i] \times (-\infty, -y'_i]$. The other two permutations can also be enforced in a symmetric manner.

If one chooses a permutation independently for every dimension, the number of choices is 4^d. This is precisely the number of different ways that r can intersect with r'. Let us refer to each of them as a *configuration*.

It is clear from the above discussion that, we can create a dominance-join structure for each of the 4^d configurations. For each configuration, we convert a rectangle $r_1 \in R_1$ into a point p by creating 3 or 2 new dimensions for every original dimension. This creates a point of dimensionality $d' \leq 3d$. Accordingly, a rectangle $r_2 \in R_2$ is converted to a d'-sided rectangle r, such that r_1 intersects with r_2 under *that* configuration if and only if r covers p.

Since each result pair $(r_1, r_2) \in R_1 \times R_2$ is reported by only one dominance join, we have obtained a feasible structure for the intersection join between R_1 and R_2. This completes the proof of Theorem 2.

5 One-Dimensional Multi-Way Joins

We now proceed to discuss 1D joins (i.e., d fixed to 1) on a constant number t of input sets $R_1, ..., R_t$. We will prove Theorem 4 by presenting a feasible structure for any join topology G

5.1 Min/Max Intersection Joins

Next, we introduce the "min" and "max" versions of intersection joins whose purposes will be clear in the next subsection.

First, let us impose two total orders on $R_1 \times ... \times R_t$. Let $(r_1, ..., r_t)$ and $(r'_1, ..., r'_t)$ be two distinct tuples from the cartesian product. Identify the first $i \in [1, t]$ such that $r_i \neq r'_i$. Then, we say:

- $(r_1, ..., r_t)$ is *left-smaller* (or *left-larger*) than $(r'_1, ..., r'_t)$ if the left endpoint of r_i is smaller (or larger) than that of r'_i;
- $(r_1, ..., r_t)$ is *right-smaller* (or *right-larger*) than $(r'_1, ..., r'_t)$ if the right endpoint of r_i is smaller (or larger) than that of r'_i.

Note that the above are always well-defined because of the general position assumption stated in Section 2 (specifically, r_i and r'_i must differ in both left endpoint and right endpoint).

Let J represent the set of tuples returned by the intersection join on $R_1, ..., R_t$ under the topology G. We define:

- **Min-IJ Query:** return the *left-smallest* tuple in J;
- **Max-IJ Query:** return the *right-largest* tuple in J.

The two queries can be supported efficiently:

Lemma 11. *There exists a structure that consumes $\tilde{O}(n)$ space, supports an update (i.e., insertion/deletion in any of $R_1, ..., R_t$) in $\tilde{O}(1)$ amortized time, and answers any min-/max-IJ query in $\tilde{O}(1)$ time.*

We will refer to the structure of the above lemma as an *IJ-heap* on $(R_1, ..., R_t)$ under G. The proof of the lemma is deferred to Section 6.

5.2 Reduction to min-IJ Queries

This subsection serves as a proof for:

Lemma 12. *Given an IJ-heap on $(R_1, ..., R_t)$ under G, we can report all the result tuples in the intersection join with an $\tilde{O}(1)$ delay.*

Combining the above with Lemma 11 gives a feasible structure needed for Theorem 4.

Proof of Lemma 12. We answer an IJ query by calling the algorithm in Figure 5 as

$$\text{IJ}(0, \emptyset)$$

which performs recursive calls at Line 8. The proposition below establishes the correctness of our algorithm:

IJ$(\lambda, \{\rho_1, ..., \rho_\lambda\})$

/* **requirements:** if $\lambda \geq 1$ then

***C*1 :** $\rho_i \in R_i$ for each $i \in [1, d]$.
***C*2 :** $\rho_1, ..., \rho_\lambda$ produce at least one result tuple.
***C*3 :** The minimum result tuple from the current $R_1, ..., R_t$ (whose content may shrink and grow during the algorithm's execution) is a tuple $(r_1^*, ..., r_t^*)$ satisfying $r_i^* = \rho_i$ for all $i \in [1, \lambda]$.

output: all result tuples $(r_1, ..., r_t)$ satisfying $r_i = \rho_i$ for all $i \in [1, \lambda]$ */

1. **if** $\lambda = t$ **then** output $(\rho_1, ..., \rho_\lambda)$ and **return**
2. $S_{del} = \emptyset$
3. **repeat**
4. $\quad \rho_{\lambda+1} \leftarrow$ the interval in $R_{\lambda+1}$ with the smallest left endpoint s.t. $\rho_1, ..., \rho_\lambda, \rho_{\lambda+1}$ produce at least one result tuple
   $\quad$ /* this requires a min-IJ query; see Proposition 3 */
5. $\quad$ **if** $\rho_{\lambda+1} =$ null **then**
6. $\quad\quad$ insert all the tuples of S_{del} back into $R_{\lambda+1}$
7. $\quad\quad$ **return**
8. $\quad$ IJ$(\lambda + 1, \{\rho_1, ..., \rho_\lambda, \rho_{\lambda+1}\})$
9. $\quad$ delete $\rho_{\lambda+1}$ from $R_{\lambda+1}$
10. $\quad$ add $\rho_{\lambda+1}$ to S_{del}

Figure 5: Reduction from intersection joins to min-IJ

Proposition 2. *$C1$, $C2$, and $C3$ in Figure 5 are fulfilled by each recursive call to IJ during the execution of IJ$(0, \emptyset)$. Furthermore, Every result tuple is output exactly once.*

Proof. See Appendix C. □

We now use the supplied IJ-heap to implement Line 4:

Proposition 3. *Line 4 takes $\tilde{O}(1)$ time.*

Proof. Use the IJ-heap to perform a min-IJ query. If the query returns nothing, set $\rho_{\lambda+1}$ to null. Otherwise, suppose that it returns $(r_1, ..., r_t)$. Check whether $r_i = \rho_i$ for all $i \in [1, \lambda]$. If so, set $\rho_{\lambda+1}$ to $r_{\lambda+1}$; otherwise, set $\rho_{\lambda+1}$ to null. Requirement $C3$ ensures the correctness of the above strategy. □

The following fact is crucial for proving that our algorithm has a short delay:

Proposition 4. *At any moment of our algorithm, if Line 1 has output x tuples, at most $t \cdot x$ deletions have been performed at Line 9 (counting the deletions made at all levels of the recursion).*

Proof. An interval (in any of $R_1, ..., R_t$) is deleted after it has produced at least a result tuple. Each result tuple output at Line 1 can trigger at most t deletions at Line 9. The proposition thus follows. □

We complete the proof of Lemma 12 by combining the following with Lemma 10:

Proposition 5. *Our algorithm is $\tilde{O}(1)$-aggressive.*

Proof. Using the supplied IJ-heap, every insertion and deletion into any R_i ($i \in [1,t]$) takes $\tilde{O}(1)$ amortized time.

Consider any moment during the execution of our algorithm. Let n_{del} be the total number of deletions that have been made at Line 9 so far (counting all levels of recursion). This implies that the total number of insertions at Line 6 is at most n_{del}. It follows that the running time thus far is $\tilde{O}(n_{del})$.

The $\tilde{O}(1)$-aggressiveness then follows from Proposition 4, which indicates that we must have reported at least n_{del}/t result tuples. □

6 The IJ-Heap

This section is dedicated to proving Lemma 11. We actually aim to support a more general form of min-/max-IJ queries. Remember that we have a constant number t of interval sets $R_1, ..., R_t$, and a join topology G. A min-/max-IJ query is now given t pairs of values

$$(a_i, b_i)$$

for $i \in [1,t]$. Let J be the set of tuples $(r_1, ..., r_t) \in R_1 \times ... \times R_t$ satisfying all of the following:

- $(r_1, ..., r_t)$ is in the result of the intersection join under topology G;
- for each $i \in [1,t]$, r_i intersects with both $(-\infty, a_i]$ and $[b_i, \infty)$. Note that, if $b_i \leq a_i$, then this condition means that r_i must intersect with $[b_i, a_i]$.

Then, the min-/max-IJ query should return the left-smallest/right-largest tuple in J. Clearly, by setting $a_i = \infty$ and $b_i = -\infty$ for all $i \in [1,t]$, a min-/max-IJ query degenerates into the version defined in Section 5.1.

The *IJ-heap* we aim to design should use of $\tilde{O}(n)$ space ($n = \sum_i |R_i|$), can be updated in $\tilde{O}(1)$ amortized time (per insertion/deletion in any R_i, $1 \leq i \leq t$), and answer any (re-defined) min-/max-IJ query in $\tilde{O}(1)$ time.

6.1 Notations

Our strategy is to break G into smaller subgraphs and handle the "sub-joins" represented by those subgraphs recursively. Some extra concepts and notations are needed to reason about those subjoins effectively.

Recall that G has the vertex set $\{1, 2, ..., t\}$, which we will call the *universe* and represent as U. Let V be any non-empty subset of U. A vector $\boldsymbol{v}$ is said to be *defined in* V if:

- $\boldsymbol{v}$ has length $|V|$;
- for each $i \in V$, $\boldsymbol{v}$ has a distinct component which we denote as $\boldsymbol{v}[i]$;
- $\boldsymbol{v}$ lists its components $\boldsymbol{v}[i]$ ($i \in V$) in ascending order of i.

Henceforth, we will write $\boldsymbol{v}$ as $\boldsymbol{v}_V$ to indicate explicitly the set V. The only exception arises when $V = U$, in which case V is omitted but implicitly understood.

Consider two non-empty subsets V, V' of U such that $V' \subset V$. Given a vector $\boldsymbol{v}_V$ defined in V, its *projection in* V' is the vector $\boldsymbol{v}'_{V'}$ where $\boldsymbol{v}'_{V'}[i] = \boldsymbol{v}_V[i]$ for each $i \in V'$.

Given a non-empty subset $V \subseteq U$, we refer to a vector $\boldsymbol{R}_V$ as an *instance vector in* V if

$$\boldsymbol{R}_V[i] \subseteq R_i$$

for each $i \in V$. Define

$$\times(\boldsymbol{R}_V) \quad = \quad \boldsymbol{R}_V[i_1] \times \boldsymbol{R}_V[i_2] \times ... \times \boldsymbol{R}_V[i_{|V|}]$$

where $i_1, i_2, ..., i_{|V|}$ list out the integers in V in ascending order. We will reserve $\boldsymbol{\mathcal{R}}$ to denote the special instance vector $(R_1, ..., R_t)$. An instance vector, in general, gives the interval sets that participate in a join.

A vector $\boldsymbol{r}_V$ is said to be a *data vector in* V if

$$\boldsymbol{r}_V[i] \in R_i$$

for each $i \in V$. Note that a data vector differs from an instance vector in that, each component of the former is a *rectangle* while each component of the latter is a *set* of rectangles.

We use G_V to represent the subgraph of G induced by the vertices in V. Given an instance vector $\boldsymbol{R}_V$, define

$$J(G_V, \boldsymbol{R}_V) \quad = \quad \big\{\boldsymbol{r}_V \in \times(\boldsymbol{R}_V) \;\big|\; \boldsymbol{r}_V[i] \cap \boldsymbol{r}_V[j] \neq \emptyset \text{ for any distinct } i, j \in V \text{ adjacent in } G_V\big\}.$$

Note that $J(G_V, \boldsymbol{R}_V)$ is the result of the intersection join defined by G_V on the interval sets $\{\boldsymbol{R}_V[i] \mid i \in V\}$. In particular, $J(G, \boldsymbol{\mathcal{R}})$ is the result of the (full) intersection join on $R_1, ..., R_t$ and G.

Next, we impose two total orders on $\times(\boldsymbol{R}_V)$, in a way consistent with the total orders defined in Section 5.1 on $\times(\boldsymbol{\mathcal{R}})$. Take any distinct elements $\boldsymbol{r}_V, \boldsymbol{r}'_V$ from $\times(\boldsymbol{R}_V)$. Let i be the smallest integer in V such that $\boldsymbol{r}_V[i] \neq \boldsymbol{r}'_V[i]$. Then, we say:

- $\boldsymbol{r}_V$ is *left-smaller* (or *left-larger*) than $\boldsymbol{r}'_V$ if the left endpoint of $\boldsymbol{r}_V[i]$ is smaller (or larger) than that of $\boldsymbol{r}'_V[i]$;
- $\boldsymbol{r}_V$ is *right-smaller* (or *right-larger*) than $\boldsymbol{r}'_V$ if the right endpoint of $\boldsymbol{r}_V[i]$ is smaller (or larger) than that of $\boldsymbol{r}'_V[i]$;

A vector $\boldsymbol{q}_V$ is said to be a *constraint vector in* V if $\boldsymbol{q}_V[i]$ is a pair

$$(\boldsymbol{q}_V[i].a, \boldsymbol{q}_V[i].b)$$

Define

$$\begin{aligned} J(G_V, \boldsymbol{R}_V, \boldsymbol{q}_V) = \big\{\boldsymbol{r}_V \in J(G_V, \boldsymbol{R}_V) \;\big|\; &\text{for all } i \in V \\ &\boldsymbol{r}_V[i] \cap (-\infty, q[i].a] \neq \emptyset \text{ and } \boldsymbol{r}_V[i] \cap [q[i].b, \infty) \neq \emptyset\big\} \end{aligned}$$

Given a constraint vector $\boldsymbol{q}_V$, a *min-IJ query* (or a *max-IJ query*) on $\boldsymbol{R}_V$ under G_V returns the left-smallest (right-largest) element in $J(G_V, \boldsymbol{R}_V, \boldsymbol{q}_V)$.

Finally, it is worth pointing out that, all the above definitions apply to $t = 1$ as well.

6.2 The Endpoint Property

Let $\boldsymbol{q}$ be a constraint vector. Set:

$$a = \max_{i=1}^{t} \boldsymbol{q}[i].a \quad (2)$$

$$b = \min_{i=1}^{t} \boldsymbol{q}[i].b \quad (3)$$

Consider an arbitrary data vector $\boldsymbol{r} = (r_1, ..., r_t)$ in $J(G, \mathfrak{R}, \boldsymbol{q})$. Define:

$$\sqcup(\boldsymbol{r}) = r_1 \cup ... \cup r_t. \quad (4)$$

Since G is connected, $\sqcup(\boldsymbol{r})$ must be a consecutive interval. Specifically, if x (or y, resp.) is the smallest (or largest, resp.) left (or right, resp.) endpoint of $r_1, ..., r_t$, then $\sqcup(\boldsymbol{r}) = [x, y]$. We must have:

Lemma 13 (Endpoint Property)**.** *If $b \leq a$, then $\sqcup(\boldsymbol{r})$ must have a non-empty intersection with $[b, a]$. Otherwise, $\sqcup(\boldsymbol{r})$ must contain both a and b.*

Proof. We will prove that $\sqcup(\boldsymbol{r})$ must intersect with both of $(-\infty, a]$ and $[b, \infty)$. Then, the lemma will follow.

Assume that $\sqcup(\boldsymbol{r})$ is disjoint with $(-\infty, a]$, that is, $\sqcup(\boldsymbol{r})$ is entirely to the right of a. Suppose that $a = \boldsymbol{q}[i].a$, for some $i \in [1, d]$. Thus, $\sqcup(\boldsymbol{r})$ is disjoint with $(-\infty, \boldsymbol{q}[i].a]$. This contradicts the fact that at least one of $r_1, ..., r_t$ must intersect with $(-\infty, \boldsymbol{q}[i].a]$.

A symmetric argument shows that $\sqcup(\boldsymbol{r})$ must also intersect with $[b, \infty)$. □

6.3 Structure Overview

Given $(R_1, ..., R_t)$ and G, we build an IJ-heap using a recursive approach, which works by induction on t.

Base: $t = 1$. Recall that the definition of min-/max-IJ queries have been extended to $t = 1$ in Section 6.1. In this case, only one interval set R_1 exists. Given a pair of real values (a, b), a min- (max-, resp.) IJ query returns the interval r with the smallest left (or largest right, resp.) endpoint, among all the intervals of R_1 that intersect with both $(-\infty, a]$ and $[b, \infty)$.

The query can be regarded as a 2D "range min" query described in Section 2.4. For this purpose, convert r into a 2D rectangle $r \times r$. Clearly, r intersects with both $(-\infty, a]$ and $[b, \infty)$ if and only if $r \times r$ intersects with the 2D rectangle $(-\infty, a] \times [b, \infty)$. Thus, the structure of Lemma 7 adequately serves our purposes.

Inductive: $t \geq 2$. Assume that we already know how to obtain an IJ-heap when G has at most $t - 1$ vertices. Next, we will design an IJ-heap for any G with t vertices.

Build an interval tree $\mathcal{T}$ on $R_1 \cup ... \cup R_t$. Denote by $\mathcal{S}$ the set of endpoints of all the intervals in $R_1 \cup ... \cup R_t$. Note that $\mathcal{S}$ is also the set of keys stored in the leaves of $\mathcal{T}$.

Consider a min-IJ query with a constraint vector $\boldsymbol{q}$. Define Π_1 (or Π_2) be the path in $\mathcal{T}$ from the root to the leaf storing the successor of a (or predecessor of b) in $\mathcal{S}$, where a and b are given in (2) and (3), respectively. Next, we introduce a taxonomy of the tuples in $J(G, \mathfrak{R}, \boldsymbol{q})$. The taxonomy will naturally lead to our strategy of answering the query.

We say that a data vector $\boldsymbol{r} = (r_1, ..., r_t)$ *hinges on* a node u in $\mathcal{T}$ if

- at least one of $r_1, ..., r_t$ belongs to $stab(u)$ (i.e., the stabbing set of u; see Section 2.1);

- none of $r_1, ..., r_t$ belongs to the stabbing set of any proper ancestor of u.

As each interval appears in exactly one stabbing set, $\boldsymbol{r}$ must hinge on exactly one node. If $\boldsymbol{r}$ hinges on u, then the interval $\sqcup(\boldsymbol{r})$ as defined in (4) must be covered by $\sigma(u)$ (i.e., the slab of u; see Section 2.1).

By the endpoint property in Lemma 13, a data vector $\boldsymbol{r} \in J(G, \mathfrak{R}, \boldsymbol{q})$ must belong to one of the following categories:

- **Category 1:** $\boldsymbol{r}$ hinges on a node on Π_1 or Π_2.
- **Category 2:** (Applicable only if $b \leq a$) $\boldsymbol{r}$ hinges on a node u whose slab $\sigma(u)$ is covered by $[b, a]$.

We will find the left-smallest data vectors from Categories 1 and 2, respectively. Then, the final answer to the min-IJ query is the left-smaller one between the two fetched data vectors.

In Section 6.4, we will describe a secondary structure associated with u, which is crucial to retrieving the data vectors of Categories 1 and 2. The algorithms for retrieving those categories will be presented in Section 6.5.

6.4 The Combination Structure

To motivate the problem to be tackled in this subsection, let us fix a node u in the interval tree $\mathcal{T}$ and consider any data vector $\boldsymbol{r} = (r_1, ..., r_t)$ that hinges on u. By definition, there must be at least one integer $i \in [1, t]$ such that r_i appears in $stab(u)$. Consider any other integer $j \in [1, t]$ that is different from i. Where can r_j appear in the interval tree $\mathcal{T}$? There are three possibilities: in the stabbing set of (i) u itself, (ii) a node in the left subtree of u, or (iii) a node in the right subtree of u.

We can divide all the data vectors hinging on u into disjoint "groups" as follows. For each $i \in [1, t]$, the r_i component of such a data vector $(r_1, ..., r_t)$ can independently take any of the aforementioned three possibilities, which gives 3^t "possibility combinations". Imagine placing two data vectors in the same group if their possibility combinations are identical. At first glance, this yields 3^t groups, but 2^t groups must be empty and, hence, useless. Specifically, a group is useless if there does not exist any $i \in [1, t]$ such that r_i takes the possibility of appearing in $stab(u)$. In a useless group, each r_i has only two possibilities; hence, the number of useless groups is 2^t. We thus conclude that the number of useful groups is $3^t - 2^t$, which is a constant.

Recall that our goal in Section 6.3 is to identify the left-smallest data vector in $J(G, \mathfrak{R}, \boldsymbol{q})$, and every data vector in $J(G, \mathfrak{R}, \boldsymbol{q})$ hinges on a node u of Category 1 or 2. Suppose that, for *every* u, we can fetch the left-smallest data vector of $J(G, \mathfrak{R}, \boldsymbol{q})$ in *every* useful group of u. Then, the overall left-smallest data vector in $J(G, \mathfrak{R}, \boldsymbol{q})$ is simply the left-smallest from all the data vectors fetched earlier. The challenge is to design a secondary structure for every u that allows fast retrieval of the aforementioned data vector from each of its useful groups. The structure must support efficient updates as well.

Next, we formalize the above strategy. Fix an arbitrary internal u in $\mathcal{T}$. For each $i \in [1, t]$, define:

- $stab_i^{<}(u)$: the set of intervals from R_i in the stabbing sets of the nodes in the left subtree of u;
- $stab_i^{=}(u)$: the set of intervals from R_i in the stabbing set of u;
- $stab_i^{>}(u)$: the set of intervals from R_i in the stabbing sets of the nodes in the right subtree of u.

We define a *combination* of u — denoted as $\mathcal{C}$ — as the cartesian product

$$stab_1^?(u) \times stab_2^?(u) \times ... \times stab_t^?(u)$$

where each of the t question marks "?" can independently take "<", "=", or ">", subject to the constraint that *at least one* of those symbols must take "=". The number of combinations is $3^t - 2^t = O(1)$.

Phrased differently, a combination $\mathcal{C}$ of u is determined by three disjoint sets $V^<$, $V^=$, and $V^>$, whose union equals the universe $U = \{1, ..., t\}$. Construct a vector $\boldsymbol{R}^{\mathcal{C}}$ where, for each $i \in U$, $\boldsymbol{R}^{\mathcal{C}}[i]$ equals

- $stab_i^<(u)$ if $i \in V^<$
- $stab_i^=(u)$ if $i \in V^=$
- $stab_i^>(u)$ if $i \in V^>$.

Thus, the combination is simply $\times(\boldsymbol{R}^{\mathcal{C}})$. Remember that $|V^=| \geq 1$.

The rest of the subsection serves as a proof of:

Lemma 14. *For each combination $\mathcal{C}$ of u, we build a structure of $\tilde{O}(|\mathcal{T}_u|)$ space to meet both requirements below:*

- *Any min-/max-IJ query on $\boldsymbol{R}^{\mathcal{C}}$ under the topology G can be answered in $\tilde{O}(1)$ time. Specifically, for any constraint vector $\boldsymbol{q}$, we can return in $\tilde{O}(1)$ time the left-smallest tuple in $J(G, \boldsymbol{R}^{\mathcal{C}}, \boldsymbol{q})$.*
- *Given an insertion/deletion in $\boldsymbol{R}^{\mathcal{C}}[i]$ for any $i \in [1, t]$, we can update the structure in $\tilde{O}(1)$ amortized time.*

We refer to the structure of the above lemma as the *combination structure* of $\mathcal{C}$.

Structure. Does G have an edge between a vertex $i \in V^<$ and a vertex $j \in V^>$? If so, we answer any min-IJ query on $\boldsymbol{R}^{\mathcal{C}}$ under G by returning nothing at all. To see why, notice that no intervals in $stab_i^<(u)$ can intersect with any intervals in $stab_j^>(u)$. Hence, $J(G, \boldsymbol{R}^{\mathcal{C}}) = \emptyset$; and accordingly, $J(G, \boldsymbol{R}^{\mathcal{C}}, \boldsymbol{q}) \subseteq J(G, \boldsymbol{R}^{\mathcal{C}}) = \emptyset$ regardless of $\boldsymbol{q}$.

Next, we focus on the situation where G has no edges between $V^<$ and $V^>$. Consider the subgraph $G^{<>}$ of G that is induced by the vertices in $U \setminus V^=$; in other words, $G^{<>}$ is obtained by removing the vertices in $V^=$ from G.

Compute the connected components (CC) of $G^{<>}$. Every CC must be a subset of either $V^<$ or $V^>$. Let h_1 be the number of CCs that are subsets of $V^<$; and if $h_1 > 0$, denote the CCs as

$$V_1^<, ..., V_{h_1}^<;$$

note that their union is $V^<$. Likewise, let h_2 be the number of CCs that are subsets of $V^>$; and if $h_2 > 0$, represent them as

$$V_1^>, ..., V_{h_2}^>;$$

note that their union is $V^>$.

The combination structure of $\mathcal{C}$ has three parts:

- (if $h_1 > 0$) for each $j \in [1, h_1]$, build an IJ-heap on the instance vector $\boldsymbol{R}_{V_j^<}$ under the topology $G_{V_j^<}$, where

$$\boldsymbol{R}_{V_j^<} = \text{the projection of } \boldsymbol{R}^{\mathcal{C}} \text{ in } V_j^<.$$

  Recall that (as defined in Section 6.1) $G_{V_j^<}$ is the subgraph of G induced by $V_j^<$. Since $|V_j^<| \leq t-1$, we already know how to build the IJ-heap on $G_{V_j^<}$ by the inductive assumption in Section 6.3.

- (if $h_2 > 0$) for each $j \in [1, h_2]$, build an IJ-heap on the instance vector $\boldsymbol{R}_{V_j^>}$ under the topology $G_{V_j^>}$, where for each $i \in V_j^>$:

$$\boldsymbol{R}_{V_j^>} = \text{the projection of } \boldsymbol{R}^{\mathcal{C}} \text{ in } V_j^>.$$

- for each $i \in V^=$, build a structure on $stab_i^=(u)$ to support:

  > Given real values λ_1, λ_2 satisfying $\lambda_1 \leq key(u) \leq \lambda_2$ and arbitrary real values a, b, this operation finds the interval with the smallest left endpoint, among all the intervals r in $stab_i^=(u)$ such that r (i) covers the entire interval $[\lambda_1, \lambda_2]$, and (ii) r intersects with $(-\infty, a]$ and $[b, \infty)$.

  Note that this operation can be supported by a 4D range-min structure of Lemma 7. To see why, let us convert $r = [x, y]$ to a 4D rectangle $[x, \infty) \times (-\infty, y] \times r \times r$. For any λ_1, λ_2, a, and b, we know: r satisfies the two conditions aforementioned, if and only if $[x, \infty) \times (-\infty, y] \times r \times r$ intersects with the 4D rectangle $(-\infty, \lambda_1] \times [\lambda_2, \infty) \times (-\infty, a] \times [b, \infty)$.

By the inductive assumption in Section 6.3, the combination structure uses $\tilde{O}(|\mathcal{T}_u|)$ space overall, and can be updated in $\tilde{O}(1)$ amortized time per insertion/deletion in $\boldsymbol{R}^{\mathcal{C}}[i]$, for any $i \in [1, t]$.

Query. We consider only min-IJ queries on $\boldsymbol{R}^{\mathcal{C}}$ under G because max-IJ queries are symmetric. Our algorithm answers a min-IJ query with constraint vector $\boldsymbol{q}$ in five steps.

Step 1: Skip this step if $h_1 = 0$. Otherwise, for each $j \in [1, h_1]$, construct a constraint vector:

$$\boldsymbol{q}_{V_j^<} = \text{the projection of } \boldsymbol{q} \text{ in } V_j^<.$$

Then, perform a max-IJ query with this vector on $\boldsymbol{R}_{V_j^<}$ under $G_{V_j^<}$ (an IJ-heap has been built for this purpose). Denote the data vector retrieved as $\boldsymbol{r}^{max}_{V_j^<}$; if the vector is null, we terminate and return nothing.

Construct a data vector $\boldsymbol{r}^{max}_{V^<}$ by setting for each $i \in V^<$

$$\boldsymbol{r}^{max}_{V^<}[i] = \boldsymbol{r}^{max}_{V_j^<}[i]$$

where j is the only integer in $[1, h_1]$ such that $i \in V_j^<$.

Step 2: Skip this step if $h_2 = 0$. Otherwise, for each $j \in [1, h_2]$, we will issue a *min*-IJ query recursively on the subjoin defined by $G_{V_j^>}$. Construct a constraint vector:

$$\boldsymbol{q}_{V_j^>} = \text{the projection of } \boldsymbol{q} \text{ in } V_j^>.$$

Perform a min-IJ query with this vector on $\boldsymbol{R}_{V_j^>}$ under $G_{V_j^>}$. Denote the data vector retrieved as $\boldsymbol{r}^{min}_{V_j^>}$; if the vector is null, we terminate and return nothing.

Construct a data vector $\boldsymbol{r}^{min}_{V^>}$ by setting for each $i \in V^>$

$$\boldsymbol{r}^{min}_{V^>}[i] = \boldsymbol{r}^{min}_{V_j^>}[i]$$

where j is the only integer in $[1, h_2]$ such that $i \in V_j^>$.

Step 3: For each $i \in V^=$, we will retrieve the interval r_i^{min} with smallest left endpoint, from all the intervals in $stab_i^=(u)$ that (i) intersect with $(-\infty, \boldsymbol{q}[i].a]$ and $[\boldsymbol{q}[i].b, \infty)$, and (ii) contain a range $[\lambda_1, \lambda_2]$, where λ_1 and λ_2 are decided in the following manner.

Denote by $N^<(i)$ the set of vertices in $V^<$ that are adjacent to i in G. If $N^<(i)$ is empty, then set $\lambda_1 = key(u)$. Otherwise, set λ_1 to the smallest right endpoint of the intervals in the following set:

$$\Big\{\boldsymbol{r}^{max}_{V^<}[i'] \;\Big|\; i' \in N^<(i)\Big\}. \tag{5}$$

Conversely, denote by $N^>(i)$ the set of vertices in $V^>$ that are adjacent to i in G. If $N^>(i)$ is empty, then set $\lambda_2 = key(u)$. Otherwise, set λ_2 to the largest left endpoint of the intervals in the following set:

$$\Big\{\boldsymbol{r}^{min}_{V^>}[i'] \;\Big|\; i' \in N^>(i)\Big\}. \tag{6}$$

Now that λ_1, λ_2 are ready, we use the 4D range min structure (of the combination structure) to find r_i^{min}. If r_i^{min} does not exist, we terminate and return nothing. Otherwise, proceed to the next step.

Step 4: For each $j \in [1, h_1]$, we will issue yet another *min*-IJ query on $\boldsymbol{R}_{V_j^<}$. First, construct a constrain vector:

$$\boldsymbol{q}'_{V_j^<} \quad = \quad \text{the projection of } \boldsymbol{q} \text{ on } V_j^<.$$

Consider each $i \in V_j^<$ in turn. Denote by $N^=(i)$ the set of vertices in $V^=$ that are adjacent to i in G. If $N^=(i) = \emptyset$, then the current $\boldsymbol{q}'_{V_j^<}[i]$ is finalized. Otherwise, we update $\boldsymbol{q}'_{V_j^<}[i].b$ to the maximum between $\boldsymbol{q}[i].b$ and λ_3, where λ_3 is the largest left endpoint of the intervals in the following set:

$$\Big\{r^{min}_{i'} \;\Big|\; i' \in N^=(i)\Big\}. \tag{7}$$

Now, perform a min-IJ query on $\boldsymbol{R}_{V_j^<}$ under $G_{V_j^<}$ with $\boldsymbol{q}'_{V_j^<}$. Denote the data vector retrieved as $\boldsymbol{r}^{min}_{V_j^<}$; if the vector is null, we terminate and return nothing.

Step 5: Return a data vector $\boldsymbol{\rho}$ where for each $i \in [1, t]$:

- $\boldsymbol{\rho}[i] = \boldsymbol{r}^{min}_{V^<}[i]$ if $i \in V^<$;
- $\boldsymbol{\rho}[i] = r_i^{min}$ if $i \in V^=$;
- $\boldsymbol{\rho}[i] = \boldsymbol{r}^{min}_{V^>}[i]$ if $i \in V^>$.

Each of the above steps finishes in $\tilde{O}(1)$ time, by the inductive assumption in Section 6.3. The overall query time is therefore $\tilde{O}(1)$. Deferring the correctness proof of the query algorithm to Appendix D, we have now established Lemma 14.

6.5 Structures for Categories 1 and 2

For every node u in the interval tree $\mathcal{T}$, we build the structure of Lemma 14 on each combination of u. All these structures occupy $\tilde{O}(n)$ space in total.

We now resume our discussion in Section 6.3 and explain how to find the left-smallest data vector $\boldsymbol{r}$ from Categories 1 and 2.

Category 1. Let u be a node on Π_1 or Π_2. A data vector that hinges on u must belong to one combination of u. We issue a min-IJ query with the constraint vector $\boldsymbol{q}$ on all the $3^t - 2^t = O(1)$ combination structures, and find the left-smallest data vector $\boldsymbol{r}$ from the data vectors fetched by those queries. It is guaranteed that $\boldsymbol{r}$ must be the left-smallest among all the data vectors in $J(G, \mathfrak{R}, \boldsymbol{q})$ that hinge on u. By Lemma 14, this takes $\tilde{O}(1)$ time.

As Π_1 and Π_2 have $\tilde{O}(1)$ nodes only, the left-smallest data vector of Category 1 can be found in $\tilde{O}(1)$ time in total.

Category 2. This category applies only if $b \leq a$. Any node u that needs to be considered must be have its slab $\sigma(u)$ covered by $[b, a]$. We pre-compute at each node u the "best answer" that u has to offer. The pre-computed information is organized in such a way that, when a min-IJ query comes, "the best of the best" answers from the *relevant* nodes can be retrieved efficiently.

To implement the above idea, let us define the *local minimum* of u, as the left-smallest among all data vectors $\boldsymbol{r} = (r_1, ..., r_t)$ in $J(G, \mathfrak{R})$ that hinge on u. The local minimum must belong to one of the combinations of u and can be found in $\tilde{O}(1)$ time as follows:

- Create a dummy constraint vector $\boldsymbol{q}'$ where $\boldsymbol{q}'[i].a = \infty$ and $\boldsymbol{q}[i].b = -\infty$ for all $i \in [1, t]$.
- Issue a min-IJ query on each of the $3^t - 2^t$ combination structures of u using $\boldsymbol{q}'$.
- Set the local minimum to the left-smallest of all the data vectors fetched by the queries at the previous step.

We now use a 2D range-min structure of Lemma 7 — denoted as $\mathcal{T}'$ — to manage the slabs of all the nodes in $\mathcal{T}$. Specifically, given a node u with slab $\sigma(u) = [x, y]$, we create a (degenerated) 2D rectangle $[x, x] \times [y, y]$, treating the local minimum of u as the rectangle's "weight". A weight $\boldsymbol{r}$ is *smaller* than another $\boldsymbol{r'}$ if $\boldsymbol{r}$ is left-smaller than $\boldsymbol{r'}$. By Lemma 7, $\mathcal{T}'$ requires only $\tilde{O}(n)$ space.

This completes the description of our IJ-heap, whose space consumption is $\tilde{O}(n)$ overall. Given a min-IJ query with constraint vector $\boldsymbol{q}$ (on $\mathcal{R}$ under G), we find the left-smallest data vector of Category 2 as follows. First, compute the values of a and b using (2) and (3). If $a < b$, ignore this category. Otherwise, perform a 2D range-min query on $\mathcal{T}'$ using the rectangle $[b, \infty) \times (-\infty, a]$. The result of the range-min query is the left-smallest local minimum of the nodes whose slabs $\sigma(u) = [x, y]$ are covered by $[b, a]$, and is what we look for in Category 2.

Therefore, we now conclude that a min-IJ query on $\mathfrak{R}$ under G can be answered in $\tilde{O}(1)$ time.

6.6 Update

The update algorithm is fairly straightforward, utilizing the result in Lemma 5.

For any node u in the interval tree $\mathcal{T}$, its secondary structure Γ_u involves only $O(1)$ combination structures of Lemma 14. Whenever an interval is insert/deleted in any of $stab^{<}(u), stab(u)$, or $stab^{>}(u)$, we update the Γ_u using Lemma 14 in $\tilde{O}(1)$ time. This means that all the functions $f_1(n), f_2(n), f_3(n)$, and $f_4(n)$ in Lemma 5 are $\tilde{O}(1)$. Thus, Lemma 5 tells us that $\mathcal{T}$ can be updated in $\tilde{O}(1)$ amortized time per insertion/deletion in any of $R_1, ..., R_t$.

Finally, the cost of updating $\mathcal{T}'$ can be piggybacked on the cost of updating $\mathcal{T}$. Specifically, for every node u in $\mathcal{T}'$ that is affected by an insertion/deletion in $\mathcal{T}$, we re-compute its local minimum in $\tilde{O}(1)$ time in the way described in Section 6.5, and update $\mathcal{T}'$ accordingly in $\tilde{O}(1)$ amortized time.

The overall update time of our IJ-heap is therefore $\tilde{O}(1)$ amortized. This completes the proof of Lemma 11 and, hence, also the proof of Theorem 4.

7 Conclusions

Given (i) t sets of d-dimensional rectangles $R_1, R_2, ..., R_t$ and (ii) a connected undirected graph G (called a topology graph) on vertices $\{1, 2, ..., t\}$, an intersection join returns all $(r_1, ..., r_t) \in R_1 \times ... \times R_t$ satisfying the condition that $r_i \cap r_j \neq \emptyset$ for all i, j such that G has an edge between vertices i and j. The paper investigates the question "*when do feasible structures exist for intersection joins?*", where a feasible structure needs to use $\tilde{O}(n)$ space (note: $n = \sum_{i=1}^{t} |R_i|$), supports an insertion/deletion in $\tilde{O}(1)$ amortized time, and permits the join result to be reported with an $\tilde{O}(1)$-time delay. Subject to the OMv-conjecture, we have answered the question by showing that a feasible structure exists if and only if $t = 2$ (binary joins, regardless of d) or $d = 1$ (1D joins, regardless of t).

A natural (and promising) direction for future research is to study how to relax the feasibility requirements to support intersection joins under updates in the scenarios where feasible structures do not exist, namely, $t \geq 3$ and $d \geq 2$ (multidimensional multiway joins). An interesting question is: if the update time has to be $\tilde{O}(1)$ amortized, what is the smallest delay (in join result reporting) achievable? An equally interesting question lies in the other extreme: if the delay has to be $\tilde{O}(1)$, what is the fastest amortized update time possible? Settling these questions would provide helpful hints towards resolving the ultimate puzzle: what is the precise tradeoff between the amortized update cost and the delay? We suspect that joins with different topology graphs G may exhibit various tradeoffs.

Acknowledgements

This work was supported in part by GRF projects 14207820, 16201318, 16201819, and 16205420 from HKRGC.

Appendix

A Optimality of Theorem 1

It suffices to show that $\Omega(\log n)$ time is needed to handle an update.

We achieve the purpose via a reduction from *predecessor search*. In that problem, we are given a set P of real values, and want to answer the following queries efficiently: given an arbitrary real value q, find its predecessor in P, namely, the largest value in P that is smaller than or equal to q. Under the comparison model, this query requires $\Omega(\log n)$ time to solve, regardless of the preprocessing on P.

Suppose that for intersection joins with $d = 1$ and $t = 2$, we are given a structure that ensures constant delay enumeration, and can be updated in $U(n)$ time. Then, we can deploy the structure to answer predecessor search in $O(U(n))$ time as explained below.

In preprocessing, convert P into an interval set

$$R_1 = \{[pre(x), x] \mid x \in P\}$$

where $pre(x)$ is the value in P immediately preceding x (if x is the minimum in P, then $pre(x) = -\infty$). Create an intersection join structure T on R_1 and an empty R_2. Given a predecessor query q on P, insert a degenerated interval $[q, q]$ into R_2, and then use T to enumerate the join result. Note that the join result contains only one tuple $[pre(q), suc(q)]$ if $q \notin P$, where $suc(q)$ is the value in P immediately succeeding q (or ∞ if no such value exists). If $q \in P$, then the join result contains 2 tuples: $[pre(q), q]$ and $[q, suc(q)]$. In either case, the predecessor of q can be found in $O(1)$ time.

The query time is $O(U(n))$, which implies $U(n) = \Omega(\log n)$.

B "Traditional" Incremental View Maintenance

As mentioned in Section 1.2, a more conventional form of maintenance aims to report the delta changes in the join result. In our intersection-join context where $R_1, ..., R_t$ are the input sets of intervals, the objectives are two fold:

1. When an interval r is inserted into R_i (for some $i \in [1, t]$), we must enumerate all the *new* result tuples of the intersection join (i.e., those involving r) with an $\tilde{O}(1)$ delay;
2. When an interval r is deleted R_i (for some $i \in [1, t]$), we must enumerate all the *disappearing* result tuples of the intersection join (i.e., those involving r) with an $\tilde{O}(1)$ delay.

The feasible structures formulated in this paper can be combined with a *leave-one-out* approach to achieve the above objectives. Specifically, we maintain t feasible structures, such that the i-th ($i \in [1, t]$) one is built on:

$$R_1, ..., R_{i-1}, \emptyset, R_{i+1}, ..., R_t.$$

Note that the i-th input set is deliberately set to empty. We will refer to the i-th feasible structure as "structure i".

To insert/delete an interval $r \in R_i$, we first insert/delete r in structures $1, ..., i-1, i+1, ..., t$. To fulfill objective (1)/(2), insert r into structure i, thereby turning the i-th input set of this structure from $\emptyset$ to $\{r\}$. Now, use structure i to enumerate *its* "join result", namely, the result of the intersection join on

$$R_1, ..., R_{i-1}, \{r\}, R_{i+1}, ..., R_t.$$

The result is exactly what is needed to achieve objective (1)/(2). After this is done, remove r from the i-th feasible structure.

Apart from enumerating the delta result changes, the update cost is $\tilde{O}(1)$. The space consumption is $\tilde{O}(n)$ where $n = \sum_i |R_i|$.

C Proof of Proposition 2

C.1 Assumptions Never Violated

This is obvious about $C1$ and $C2$ (in particular, $C2$ is ensured by the if-condition at Line 5). Next, we focus on $C3$.

First observe that, because of Line 6, we definitely the following *clean return property*:

When IJ$(\lambda, ...)$ finishes, $R_{\lambda+1}, ..., R_t$ have been restored to their original content (i.e., same as before the query started).

Next we prove that $C3$ always holds by induction on λ. In general, when the IJ algorithm is invoked with parameters $(\lambda, ...)$, we say that a *level-λ* call has been made.

Base case: $\boldsymbol{\lambda = 0}$. $C3$ obviously holds for the first recursive call made by IJ$(0, \emptyset)$ (at Line 8).

Consider two consecutive recursive calls made by IJ$(0, \emptyset)$: let the first be IJ$(1, \{\rho_1\})$, and the second be IJ$(1, \{\rho'_1\})$. $R_2, ..., R_t$ are the same for both calls, due to the clean return property. Regarding R_1, the difference is that, for the second call, R_1 contains one less interval: ρ_1 (which has been deleted at Line 9 after the first call finished). Assuming that $C3$ holds for the first call, next we show that it must also hold for the second.

Suppose that this is not true. Thus, for the second call, there exists an interval $\rho''_1 \in R_1$ such that ρ''_1 produces a result tuple and has a smaller left endpoint than ρ'_1. But this contradicts how ρ'_1 was obtained at Line 4 right before the second call.

Inductive case $\boldsymbol{\lambda = i + 1}$. Assume that $C3$ holds for all the level-λ calls with $\lambda \leq i$. We will prove that the same is true for $\lambda = i + 1$.

For this purpose, fix an arbitrary level-i call IJ$(i, \{\rho_1, ..., \rho_i\})$. It suffices to show that all the calls it makes at Line 8 have $C3$ fulfilled.

By the inductive assumption, at the beginning of IJ$(i, \{\rho_1, ..., \rho_i\})$, $\rho_1, ..., \rho_i$ produce the minimum result tuple from the current $R_1, ..., R_t$. Thus, $C3$ holds for the first call made by IJ$(i, \{\rho_1, ..., \rho_i\})$.

Consider two consecutive recursive calls made by IJ$(i, \{\rho_1, ..., \rho_i\})$: let the first be IJ$(i + 1, \{\rho_1, ..., \rho_i, \rho_{i+1}\})$, and the second be IJ$(i + 1, \{\rho_1, ..., \rho_i, \rho'_{i+1}\})$. $R_{i+2}, ..., R_t$ are the same for both calls, due to the clean return property. $R_1, ..., R_i$ are also the same because they are not modified within IJ$(i, \{\rho_1, ..., \rho_i\})$. Regarding R_i, the difference is that, for the second call, R_i contains one less interval: ρ_{i+1} (which has been deleted at Line 9 after the first call finished). Assuming that $C3$ holds for the first call, next we show that it must also hold for the second.

Suppose that this is not true. Thus, for the second call, there exists an interval $\rho''_{i+1} \in R_{i+1}$ such that ρ''_{i+1} produces a result tuple with $\rho_1, ..., \rho_i$, and has a smaller left endpoint than ρ'_{i+1}. But this contradicts how ρ'_{i+1} was obtained at Line 4 right before the second call.

C.2 Correctness of the Output

That no result tuple is reported twice follows from the fact that the tuple $(\rho_1, ..., \rho_t)$ output at Line 1 becomes monotonically left-larger (see the definition of "left-larger" in Section 5.1).

To prove that no result tuple is missed, suppose that the algorithm fails to output some result tuple $(r_1, ..., r_t)$. Let $i \geq 0$ be the largest integer such that a call to IJ was made with the parameters $(i, \{r_1, ..., r_i\})$. Hence, $i < t$.

By the clean return property stated in the proof of Proposition 2, we know that when IJ$(i, \{r_1, ..., r_i\})$ started, $R_{i+1}, ..., R_t$ had been fully restored, i.e., no tuples were missing in $R_{i+1}, ..., R_t$. This immediately implies that Line 8 must have made a recursive call IJ$(i + 1, \{r_1, ..., r_i, r_{i+1}\})$, giving a contradiction.

D Correctness Proof of the Query Algorithm in Section 6.4

We will first prove in Section D.1 an important property before establishing the query algorithm's correctness in Section D.2.

D.1 The Local-Extreme Property

Consider any non-empty $V \subseteq U$, and any instance vector $\boldsymbol{R}_V$ in V. Given a constraint vector $\boldsymbol{q}_V$ in V, define for each $i \in V$:

$$J_i(G_V, \boldsymbol{R}_V, \boldsymbol{q}_V) \;=\; \Big\{ r \;\Big|\; \exists \boldsymbol{r}_V \in J(G_V, \boldsymbol{R}_V, \boldsymbol{q}_V) \text{ s.t. } \boldsymbol{r}_V[i] = r \Big\}.$$

One can regard $J_i(G_V, \boldsymbol{R}_V, \boldsymbol{q}_V)$ as the "projection" of $J(G_V, \boldsymbol{R}_V, \boldsymbol{q}_V)$ on i.

Let r_i^{min} (or r_i^{max}) be the interval in $J_i(G_V, \boldsymbol{R}_V, \boldsymbol{q}_V)$ with the smallest left (or largest right, resp.) endpoint. We have:

Lemma 15 (Local-Extreme Property)**.** *Let $\boldsymbol{\rho}_V^{\boldsymbol{min}}$ and $\boldsymbol{\rho}_V^{\boldsymbol{max}}$ be the left-smallest and right-largest elements of $J(G_V, \boldsymbol{R}_V, \boldsymbol{q}_V)$, respectively. Then, for every $i \in V$, it must hold that*

$$\begin{aligned} \boldsymbol{\rho}_V^{\boldsymbol{min}}[i] &= r_i^{min} \\ \boldsymbol{\rho}_V^{\boldsymbol{max}}[i] &= r_i^{max}. \end{aligned}$$

Proof. We will prove only the part of the lemma about $\boldsymbol{\rho}_V^{\boldsymbol{min}}$, because a symmetric argument will then apply to $\boldsymbol{\rho}_V^{\boldsymbol{max}}$.

Construct a vector $\boldsymbol{r}_V^{\boldsymbol{min}}$ with $\boldsymbol{r}_V^{\boldsymbol{min}}[i] = r_i^{min}$ for every $i \in V$. It suffices to show that $\boldsymbol{r}_V^{\boldsymbol{min}}$ is in $J(G_V, \boldsymbol{R}_V, \boldsymbol{q}_V)$. Suppose that this is not true. Thus, there exist distinct i, j in V such that (i) G_V has an edge between i and j, (ii) but r_i^{min} is disjoint with r_j^{min}. Without loss of generality, assume that r_i^{min} is to the left of r_j^{min}.

By the fact that $r_i^{min} \in J_i(G_V, \boldsymbol{R}_V, \boldsymbol{q}_V)$, $J(G_V, \boldsymbol{R}_V, \boldsymbol{q}_V)$ has a data vector $\boldsymbol{r}'_V$ with $\boldsymbol{r}'_V[i] = r_i^{min}$. The fact $\boldsymbol{r}'_V \in J(G_V, \boldsymbol{R}_V, \boldsymbol{q}_V)$ means that $\boldsymbol{r}'_V[i]$ intersects with $\boldsymbol{r}'_V[j]$. Thus, $\boldsymbol{r}'_V[j]$ must have a smaller left endpoint than r_j^{min}, contradicting the definition of r_j^{min}. □

The lemma suggests a direction for answering a min-IJ query. We can "detach" the query by looking at the "projection" $J_i(G_V, \boldsymbol{R}_V, \boldsymbol{q}_V)$ on each $i \in V$ *individually*. Once we have found r_i^{min} for each i, putting them together gives the answer to the min-IJ query. A symmetric strategy works for max-IJ queries.

D.2 Correctness Proof

We are now ready to establish the correctness of the query algorithm in Section 6.4. Remember that, given a constraint vector $\boldsymbol{q}$, a min-IJ query finds the left-smallest element in $J(G, \boldsymbol{R}^{\mathbb{C}}, \boldsymbol{q})$. We will denote that element as $\boldsymbol{\rho}^{min}$; note that when $J(G, \boldsymbol{R}^{\mathbb{C}}, \boldsymbol{q})$ is empty, $\boldsymbol{\rho}^{min}$ is null. We need to prove that (i) if $\boldsymbol{\rho}^{min}$ is null, our algorithm returns empty, and (ii) otherwise, the data vector $\boldsymbol{\rho}$ we return must be $\boldsymbol{\rho}^{min}$.

In this proof, given an interval r, we will use $r.\vdash$ to denote its left endpoint, and $r.\dashv$ to denote its right endpoint. Note that both $r.\vdash$ and $r.\dashv$ are real values. Also, remember that u is the node whose combination structure is being searched.

Proposition 6. *$J(G, \boldsymbol{R}^{\mathbb{C}}, \boldsymbol{q}) = \emptyset$ when either of the following happens:*

- *$\boldsymbol{r}_{V_j^<}^{max}$ (computed in Step 1) is null for any $j \in [1, h_1]$;*
- *$\boldsymbol{r}_{V_j^>}^{min}$ (computed in Step 2) is null for any $j \in [1, h_2]$.*

Proof. We will prove only the first bullet, because a similar argument works for the second. Note that $\boldsymbol{r}^{max}_{V_j^<}$ being null implies $J(G_{V_j^<}, \boldsymbol{R}_{V_j^<}, \boldsymbol{q}_{V_j^<}) = \emptyset$. If $J(G, \boldsymbol{R}^{\mathcal{C}}, \boldsymbol{q})$ is not empty, let $\boldsymbol{\rho}$ be an arbitrary data vector therein. It is easy to verify that the projection of $\boldsymbol{\rho}$ in $V_j^<$ is in $J(G_{V_j^<}, \boldsymbol{R}_{V_j^<}, \boldsymbol{q}_{V_j^<})$, contradicting $J(G_{V_j^<}, \boldsymbol{R}_{V_j^<}, \boldsymbol{q}_{V_j^<}) = \emptyset$. □

Proposition 7. *If $J(G, \boldsymbol{R}^{\mathcal{C}}, \boldsymbol{q})$ is not empty, then for every $i \in V^>$, it must hold that*

$$\boldsymbol{\rho}^{min}[i] = \boldsymbol{r}^{min}_{V^>}[i].$$

Proof. Suppose that this is not true. Construct a different data vector $\boldsymbol{\rho}'$ as follows:

- for every $i \in V^< \cup V^=$, set $\boldsymbol{\rho}'[i] = \boldsymbol{\rho}^{min}[i]$;
- for every $i \in V^>$, set $\boldsymbol{\rho}'[i] = \boldsymbol{r}^{min}_{V^>}[i]$.

For any $j \in [1, h_2]$, the projection of $\boldsymbol{\rho}^{min}$ in $V_j^>$ is in $J(G_{V_j^>}, \boldsymbol{R}_{V_j^>}, \boldsymbol{q}_{V_j^>})$. By applying the local-extreme property of Lemma 15 on $G_{V_j^>}$, we know $\boldsymbol{r}^{min}_{V^>}[i].\vdash \leq \boldsymbol{\rho}^{min}[i].\vdash$ for every $i \in V^>$. Hence, $\boldsymbol{r}^{min}_{V^>}[i].\vdash \leq \boldsymbol{\rho}^{min}[i].\vdash$ for every $i \in V^>$. This suggests that $\boldsymbol{\rho}'$ must be left-smaller than $\boldsymbol{\rho}^{min}$.

Next we will show that $\boldsymbol{\rho}'$ is in $J(G, \boldsymbol{R}^{\mathcal{C}}, \boldsymbol{q})$, thus contradicting the role of $\boldsymbol{\rho}^{min}$. It suffices to prove: for any $i \in V^>$ and $i' \in V^=$ such that G has an edge between i, i', we must have: $\boldsymbol{\rho}'[i]$ intersects with $\boldsymbol{\rho}'[i']$. Clearly, $\boldsymbol{\rho}^{min}[i]$ intersects with $\boldsymbol{\rho}^{min}[i']$. As $\boldsymbol{\rho}^{min}[i']$ covers $key(u)$ but $\boldsymbol{\rho}^{min}[i].\vdash > key(u)$, we know that $\boldsymbol{\rho}^{min}[i']$ must cover $\boldsymbol{\rho}^{min}[i].\vdash$ and, thus, also $\boldsymbol{r}^{min}_{V^>}[i].\vdash$ (using the fact $key(u) < \boldsymbol{r}^{min}_{V^>}[i].\vdash \leq \boldsymbol{\rho}^{min}[i].\vdash$). We therefore conclude that $\boldsymbol{\rho}'[i]$ intersects with $\boldsymbol{\rho}'[i']$. □

Proposition 8. *If r_i^{min} (computed in Step 3) is null for any $i \in V^=$, then $J(G, \boldsymbol{R}^{\mathcal{C}}, \boldsymbol{q}) = \emptyset$.*

Proof. Suppose that $J(G, \boldsymbol{R}^{\mathcal{C}}, \boldsymbol{q}) \neq \emptyset$; thus, $\boldsymbol{\rho}^{min}$ is not null. By definition, $\boldsymbol{\rho}^{min}[i]$ intersects with $(-\infty, \boldsymbol{q}[i].a]$ and $[\boldsymbol{q}[i].b, \infty)$. We will prove that $\boldsymbol{\rho}^{min}[i]$ contains the interval $[\lambda_1, \lambda_2]$ obtained in Step 3 for i, thus contradicting the fact that r_i^{min} is null.

By definition, $\lambda_1 \leq key(u) \leq \lambda_2$. Hence, it suffices to prove that $\boldsymbol{\rho}^{min}[i]$ contains both $[\lambda_1, key(u)]$ and $[key(u), \lambda_2]$. We will prove this only for $[\lambda_1, key(u)]$ because a symmetric argument proves the same for $[key(u), \lambda_2]$.

If $N^<(i)$ is empty, then $\lambda_1 = key(u)$, in which case $\boldsymbol{\rho}^{min}[i]$ obviously covers $[\lambda_1, key(u)]$. Next, we focus on the scenario where $N^<(i)$ is not empty.

For every $i' \in N^<(i)$, $\boldsymbol{\rho}^{min}[i']$ lies to the left of $key(u)$. Therefore, $\boldsymbol{\rho}^{min}[i]$ must cover $\boldsymbol{\rho}^{min}[i'].\dashv$. This indicates that $\boldsymbol{\rho}^{min}[i]$ must cover the smallest right endpoint λ_1' of the intervals in the following set:

$$\Big\{\boldsymbol{\rho}^{min}[i'] \;\Big|\; i' \in N^<(i)\Big\}. \tag{8}$$

It remains to show that $\lambda_1' \leq \lambda_1$. By comparing (8) to (5), one can see that we only need to show that $\boldsymbol{\rho}^{min}[i'].\dashv \leq \boldsymbol{r}^{max}_{V^<}[i'].\dashv$, for each $i' \in N^<(i)$.

Fix an arbitrary $i' \in N^<(i)$. Identify the only $j \in [1, h_1]$ satisfying $i' \in V_j^<$. Let $\boldsymbol{\rho}^{min}_{V_j^<}$ be the projection of $\boldsymbol{\rho}^{min}$ in $V_j^<$. Since $\boldsymbol{\rho}^{min}_{V_j^<} \in J(G_{V_j^<}, \boldsymbol{R}_{V_j^<}, \boldsymbol{q}_{V_j^<})$, applying the local-extreme property in Lemma 15 on $G_{V_j^<}$ shows that $\boldsymbol{\rho}^{min}_{V_j^<}[i'].\dashv \leq \boldsymbol{r}^{max}_{V_j^<}[i'].\dashv = \boldsymbol{r}^{max}_{V^<}[i'].\dashv$. □

Proposition 9. *If $J(G, \boldsymbol{R}^{\mathcal{C}}, \boldsymbol{q})$ is not empty, then for every $i \in V^=$, it must hold that*

$$\boldsymbol{\rho}^{min}[i] = r_i^{min}.$$

Proof. Suppose that this is not true. Fix an $i \in V^{=}$ such that $\boldsymbol{\rho}^{min}[i] \neq r_i^{min}$. In Proposition 8, we have proved that $\boldsymbol{\rho}^{min}[i]$ contains the interval $[\lambda_1, \lambda_2]$ obtained in Step 3 for i. Furthermore, $\boldsymbol{\rho}^{min}[i]$ needs to intersect with $(-\infty, \boldsymbol{q}[i].a]$ and $[\boldsymbol{q}[i].b, \infty)$. By how r_i^{min} is computed, it must hold that $r_i^{min}.\vdash < \boldsymbol{\rho}^{min}[i].\vdash$.

Construct a data vector $\boldsymbol{\rho}'$ as follows:

- for every $i' \neq i$, set $\boldsymbol{\rho}'[i'] = \boldsymbol{\rho}^{min}[i]$;
- set $\boldsymbol{\rho}'[i] = r_i^{min}$.

We will prove that $\boldsymbol{\rho}'$ is in $J(G, \boldsymbol{R}^{\mathcal{C}}, \boldsymbol{q})$ which, given the fact that $\boldsymbol{\rho}'$ is left-smaller than $\boldsymbol{\rho}^{min}$, contradicts the role of $\boldsymbol{\rho}^{min}$.

It suffices to show that:

- for any $i' \in V^{<}$ that is adjacent to i in G, $\boldsymbol{\rho}'[i']$ intersects with $\boldsymbol{\rho}'[i]$;
- for any $i' \in V^{>}$ that is adjacent to i in G, $\boldsymbol{\rho}'[i']$ intersects with $\boldsymbol{\rho}'[i]$.

To prove the first bullet, first note that $\boldsymbol{\rho}^{min}[i']$ intersects with $\boldsymbol{\rho}^{min}[i]$. Since $\boldsymbol{\rho}^{min}[i]$ contains $key(u)$ but $\boldsymbol{\rho}^{min}[i'].\dashv < key(u)$, we know $\boldsymbol{\rho}^{min}[i].\vdash \leq \boldsymbol{\rho}^{min}[i'].\dashv$. This leads to $r_i^{min}.\vdash \leq \boldsymbol{\rho}^{min}[i'].\dashv$, indicating that r_i^{min} intersects with $\boldsymbol{\rho}^{min}[i']$(recall that r_i^{min} covers $key(u)$). We therefore conclude that $\boldsymbol{\rho}'[i]$ intersects with $\boldsymbol{\rho}'[i']$.

To prove the second bullet, let j be the only integer in $V^{>}$ such that $i' \in V_j^{>}$. Consider the $\boldsymbol{r}_{V_j^{>}}^{min}$ obtained in Step 2. Proposition 7 guarantees that $\boldsymbol{\rho}'[i'] = \boldsymbol{\rho}^{min}[i'] = \boldsymbol{r}_{V_j^{>}}^{min}[i']$. Since $\boldsymbol{r}_{V_j^{>}}^{min}[i']$ belongs to the set in (6), the value λ_2 obtained at Step 3 for i must satisfy

$$\lambda_2 \geq \boldsymbol{r}_{V_j^{>}}^{min}[i'].\vdash > key(u).$$

By how r_i^{min} is computed, r_i^{min} must cover $key(u)$ and λ_2. Hence, r_i^{min} covers $\boldsymbol{r}_{V_j^{>}}^{min}[i'].\vdash$ as well, implying that r_i^{min} intersects with $\boldsymbol{r}_{V_j^{>}}^{min}[i']$. We thus conclude that $\boldsymbol{\rho}'[i]$ intersects with $\boldsymbol{\rho}'[i']$. □

Proposition 10. *If $\boldsymbol{r}_{V_j^{<}}^{min}$ (computed in Step 4) is null for any $j \in [1, h_1]$, then $J(G, \boldsymbol{R}^{\mathcal{C}}, \boldsymbol{q}) = \emptyset$.*

Proof. That $\boldsymbol{r}_{V_j^{<}}^{min}$ is null implies $J(G_{V_j^{<}}, \boldsymbol{R}_{V_j^{<}}, \boldsymbol{q}'_{V_j^{<}}) = \emptyset$. Suppose that $J(G, \boldsymbol{R}^{\mathcal{C}}, \boldsymbol{q}) \neq \emptyset$; thus, $\boldsymbol{\rho}^{min}$ is not null. Let $\boldsymbol{\rho}_{V_j^{<}}^{min}$ the projection of $\boldsymbol{\rho}^{min}$ in $V_j^{<}$. We will show that $\boldsymbol{\rho}_{V_j^{<}}^{min} \in J(G_{V_j^{<}}, \boldsymbol{R}_{V_j^{<}}, \boldsymbol{q}'_{V_j^{<}})$, thus contradicting $J(G_{V_j^{<}}, \boldsymbol{R}_{V_j^{<}}, \boldsymbol{q}'_{V_j^{<}}) = \emptyset$.

It suffices to show that, for each $i \in V_j^{<}$, $\boldsymbol{\rho}_{V_j^{<}}^{min}[i]$ intersects with $[\boldsymbol{q}'_{V_j^{<}}.b, \infty)$. This is obvious if $N^{=}(i) = \emptyset$ because (i) in this case $\boldsymbol{q}'_{V_j^{<}}$ is the projection of $\boldsymbol{q}$ in $V_j^{<}$, and (ii) by definition $\boldsymbol{\rho}^{min}[i]$ must intersect with $[\boldsymbol{q}[i].b, \infty)$.

Consider now $N^{=}(i) \neq \emptyset$. For each $i' \in N^{=}(i)$, we know from Proposition 9 that $\boldsymbol{\rho}^{min}[i'] = r_{i'}^{min}$. As $\boldsymbol{\rho}^{min}[i].\dashv < key(u)$ but $\boldsymbol{\rho}^{min}[i']$ covers $key(u)$, we know that $\boldsymbol{\rho}^{min}[i]$ must intersect with $[\boldsymbol{\rho}^{min}[i'].\vdash, \infty)$. As the above holds for every $i' \in N^{=}(i)$, we assert that $\boldsymbol{\rho}^{min}[i]$ must intersect with $[\lambda_3, \infty)$ (recall how λ_3 is derived from (7)). Therefore, $\boldsymbol{\rho}_{V_j^{<}}^{min}[i]$ must intersect with $[\boldsymbol{q}'_{V_j^{<}}.b, \infty)$, meaning that $\boldsymbol{\rho}_{V_j^{<}}^{min}$ is in $J(G_{V_j^{<}}, \boldsymbol{R}_{V_j^{<}}, \boldsymbol{q}'_{V_j^{<}})$. □

Proposition 11. *If $J(G, \boldsymbol{R}^{\complement}, \boldsymbol{q})$ is not empty, then for every $i \in V^{<}$, it must hold that*

$$\boldsymbol{\rho}^{min}[i] = \boldsymbol{r}^{min}_{V^{<}}[i].$$

Proof. For each $j \in [1, h_1]$, We have proved in Proposition 10 that the projection $\boldsymbol{\rho}^{min}_{V_j^{<}}$ of $\boldsymbol{\rho}^{min}$ in $V_j^{<}$ must belong to $J(G_{V_j^{<}}, \boldsymbol{R}_{V_j^{<}}, \boldsymbol{q}'_{V_j^{<}})$. The local-extreme property of Lemma 15 tells us that that

$$\boldsymbol{r}^{min}_{V^{<}}[i].\vdash \leq \boldsymbol{\rho}^{min}_{V_j^{<}}[i].\vdash$$

holds for every $i \in V_j^{<}$.

Suppose that the proposition does not hold. Construct a data vector $\boldsymbol{\rho}'$ as follows:

- for every $i \in V^{=} \cup V^{>}$, set $\boldsymbol{\rho}'[i] = \boldsymbol{\rho}^{min}[i]$;
- for every $i \in V^{<}$, set $\boldsymbol{\rho}'[i] = \boldsymbol{r}^{min}_{V^{<}}[i]$.

Thus, $\boldsymbol{\rho}'$ is left-smaller than $\boldsymbol{\rho}^{min}$. Next, we will show that $\boldsymbol{\rho}'$ is in $J(G, \boldsymbol{R}^{\complement}, \boldsymbol{q})$, thus contradicting the role of $\boldsymbol{\rho}^{min}$.

It suffices to show that $\boldsymbol{\rho}'[i]$ intersects with $\boldsymbol{\rho}'[i']$, for any $i \in V^{<}$ and $i' \in V^{=}(i)$. Fix j to be the unique integer in $[1, h_1]$ such that $i \in V_j^{<}$. Since $\boldsymbol{\rho}'[i']$ is in the set of (7), the λ_3 computed in Step 4 for i is at least $\boldsymbol{\rho}'[i'].\vdash$. By how $\boldsymbol{r}^{min}_{V_j^{<}}$ is computed, $\boldsymbol{r}^{min}_{V_j^{<}}[i]$ must intersect with $[\lambda_3, \infty)$ and, hence, must also intersect with $\boldsymbol{\rho}'[i']$ (here, we used the fact that $\boldsymbol{\rho}'[i']$ covers $key(u)$). We thus conclude that $\boldsymbol{\rho}'[i] = \boldsymbol{r}^{min}_{V^{<}}[i] = \boldsymbol{r}^{min}_{V_j^{<}}[i]$ intersects with $\boldsymbol{\rho}'[i']$. □

The correctness of our algorithm follows from all the above propositions, and the fact that the vector $\boldsymbol{\rho}$ constructed in Step 5 is in $J(G, \boldsymbol{R}^{C}, \boldsymbol{q})$, implying that $J(G, \boldsymbol{R}^{C}, \boldsymbol{q})$ is not empty.

References

[1] Pankaj K. Agarwal, Lars Arge, Haim Kaplan, Eyal Molad, Robert Endre Tarjan, and Ke Yi. An optimal dynamic data structure for stabbing-semigroup queries. *SIAM J. of Comp.*, 41(1):104–127, 2012.

[2] Lars Arge and Jeffrey Scott Vitter. Optimal external memory interval management. *SIAM J. of Comp.*, 32(6):1488–1508, 2003.

[3] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering conjunctive queries under updates. In *PODS*, pages 303–318, 2017.

[4] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering FO+MOD queries under updates on bounded degree databases. In *ICDT*, pages 8:1–8:18, 2017.

[5] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Efficient processing of spatial joins using R-trees. In *SIGMOD*, pages 237–246, 1993.

[6] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 3rd edition, 2008.

[7] Bipin C. Desai. Performance of a composite attribute and join index. *IEEE Trans. Software Eng.*, 15(2):142–152, 1989.

[8] David J. DeWitt, Jeffrey F. Naughton, and Donovan A. Schneider. An evaluation of non-equijoin algorithms. In *VLDB*, pages 443–452, 1991.

[9] Anton Dignös, Michael H. Böhlen, and Johann Gamper. Overlap interval partition join. In *SIGMOD*, pages 1459–1470, 2014.

[10] Herbert Edelsbrunner. Dynamic data structures for orthogonal intersection queries. *Report F59, Inst. Informationsverarb., Tech. Univ. Graz*, 1980.

[11] Jost Enderle, Matthias Hampel, and Thomas Seidl. Joining interval data in relational databases. In *SIGMOD*, pages 683–694, 2004.

[12] Pankaj Goyal, Hon Fung Li, Eric Regener, and Fereidoon Sadri. Scheduling of page fetches in join operations using Bc-trees. In *ICDE*, pages 304–310, 1988.

[13] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *STOC*, pages 21–30, 2015.

[14] Muhammad Idris, Martín Ugarte, and Stijn Vansummeren. The dynamic yannakakis algorithm: Compact and efficient query processing under updates. In *SIGMOD*, pages 1259–1274. ACM, 2017.

[15] Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. Conjunctive queries with inequalities under updates. *PVLDB*, 11(7):733–745, 2018.

[16] Ahmet Kara, Hung Q. Ngo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Maintaining triangle queries under updates. *TODS*, 45(3):11:1–11:46, 2020.

[17] Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Trade-offs in static and dynamic evaluation of hierarchical queries. In Dan Suciu, Yufei Tao, and Zhewei Wei, editors, *PODS*, pages 375–392, 2020.

[18] Zuhair Khayyat, William Lucia, Meghna Singh, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Panos Kalnis. Fast and scalable inequality joins. *VLDB J.*, 26(1):125–150, 2017.

[19] Christoph Koch. Incremental query evaluation in a ring of databases. In *PODS*, pages 87–98, 2010.

[20] Christoph Koch, Daniel Lupei, and Val Tannen. Incremental view maintenance for collection programming. In *PODS*, pages 75–90, 2016.

[21] Katja Losemann and Wim Martens. MSO queries on trees: enumerating answers under updates. In *Joint Meeting of the Annual Conference on Computer Science Logic (CSL) and the Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS*, pages 67:1–67:10, 2014.

[22] Nikos Mamoulis and Dimitris Papadias. Multiway spatial joins. *TODS*, 26(4):424–475, 2001.

[23] Edward M. McCreight. Efficient algorithms for enumerating intersecting intervals and rectangles. *Report CSL-80-9, Xerox Palo Alto Res. Center*, 1980.

[24] Jurg Nievergelt and Edward M. Reingold. Binary search trees of bounded balance. *SIAM J. of Comp.*, 2(1):33–43, 1973.

[25] Dimitris Papadias, Nikos Mamoulis, and Yannis Theodoridis. Processing and optimization of multiway spatial joins using r-trees. In *PODS*, pages 44–55, 1999.

[26] Jignesh M. Patel and David J. DeWitt. Partition based spatial-merge join. In *SIGMOD*, pages 259–270, 1996.

[27] Danila Piatov, Sven Helmer, and Anton Dignös. An interval join optimized for modern hardware. In *ICDE*, pages 1098–1109, 2016.

[28] Farhan Tauheed, Thomas Heinis, and Anastasia Ailamaki. THERMAL-JOIN: A scalable spatial join for dynamic workloads. In *SIGMOD*, pages 939–950. ACM, 2015.

[29] Dan E. Willard. An algorithm for handling many relational calculus queries efficiently. *JCSS*, 65(2):295–331, 2002.

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

[31] Thomas Zeume and Thomas Schwentick. Dynamic conjunctive queries. *JCSS*, 88:3–26, 2017.

[32] Rui Zhang, Jianzhong Qi, Dan Lin, Wei Wang, and Raymond Chi-Wing Wong. A highly optimized algorithm for continuous intersection join queries over moving objects. *VLDB J.*, 21(4):561–586, 2012.