Towards Efficient Random-Order Enumeration for Join Queries

Pengyu Chen
pchen.research@gmail.com
Harbin Institute of Technology
Harbin, China

Zizheng Guo
zguo.research@gmail.com
Harbin Institute of Technology
Zhengzhou Advanced Research Institute
Zhengzhou, China

Jianwei Yang
yangjianwei006@cnpc.com.cn
Harbin Institute of Technology
Harbin, China

Dongjing Miao
miaodongjing@hit.edu.cn
Harbin Institute of Technology
Harbin, China

ABSTRACT

In many data analysis pipelines, a basic and time-consuming process is to produce join results and feed them into downstream tasks. Numerous enumeration algorithms have been developed for this purpose. To be a statistically meaningful representation of the whole join result, the result tuples are required to be enumerated in uniformly random order. However, existing studies lack an efficient random-order enumeration algorithm with a worst-case runtime guarantee for (cyclic) join queries. In this paper, we develop an efficient random-order enumeration algorithm for join queries with no large hidden constants in its complexity, achieving expected $O(\frac{\mathrm{AGM}(Q)}{|Res(Q)|}\log^2|Q|)$ delay, $O(\mathrm{AGM}(Q)\log|Q|)$ total running time after $O(|Q|\log|Q|)$-time index construction, where $|Q|$ is the size of input, $\mathrm{AGM}(Q)$ is the AGM bound, and $|Res(Q)|$ is the size of the join result. We prove that our algorithm is near-optimal in the worst case, under the combinatorial k-clique hypothesis. Our algorithm requires no query-specific preprocessing and can be flexibly adapted to many common database indexes with only minor modifications. We also devise non-trivial techniques to speed up enumeration and reduce memory usage, and present an experimental study of their impact on our algorithm. The experimental results show that our algorithm, enhanced with the proposed techniques, significantly outperforms existing state-of-the-art methods.

PVLDB Reference Format:
Pengyu Chen, Zizheng Guo, Jianwei Yang, and Dongjing Miao. Towards Efficient Random-Order Enumeration for Join Queries. PVLDB, 19(5): 889 - 901, 2026.
doi:10.14778/3796195.3796203

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Chen-Py/JoinREnum.

Proceedings of the VLDB Endowment, Vol. 19, No. 5 ISSN 2150-8097.
doi:10.14778/3796195.3796203

1 INTRODUCTION

As one of the most fundamental types of queries in database systems, join queries[1] combine columns from one or more tables into a new table based on equality conditions. The efficient evaluation algorithm of join queries has been extensively studied theoretically and practically in the database community. Many of these algorithms [1, 15, 18–22, 25] can achieve asymptotically optimal performance (sometimes up to a polylogarithmic factor) in worst-case conditions. However, joins remain computationally expensive, even with the implementation of worst-case optimal algorithms. The major reason for this is the size of the join result: a join query Q may produce tuples as many as the AGM bound [2], which is usually much larger than the size of the relation tables of Q. Then, even listing all tuples in the join result requires a large amount of time in the worst case. This poses a significant challenge in database systems and data analysis pipelines [13].

To address the aforementioned challenge, numerous algorithms have been proposed that generate and output result tuples to downstream tasks in a continuous manner [3–5, 7, 8], without waiting for all join results to be generated in advance. These algorithms, say *enumeration algorithms*, offer the advantage that the number of intermediate results is proportional to the processing time. This is useful when the query is a part of a larger data analysis pipeline, where the query results are sent to downstream processing [9]. For example, in machine learning pipelines, we may use query results as the training data or the features for the task of training a model [14, 17, 24]. In such pipelines, intermediate results can be fed to the next-step process without waiting for complete result materialization. This can be seen in streaming learning algorithms [24]. In addition, users are able to halt the training when the learning process reaches a stable state.

Furthermore, enumeration in random order is required, *i.e.*, each query result tuple is enumerated uniformly at random from the set of result tuples not yet enumerated [9]. In this way, at any time, the algorithm outputs a statistically meaningful representation of all the join results. Many downstream tasks, where the complete query result is not necessarily required, can benefit significantly from the random samples, and the benefit increases with the size of the samples. To this end, Carmeli et al. [9] introduced a random-order enumeration algorithm for free-connex acyclic conjunctive queries with a polylogarithmic delay after a linear-time preprocessing phase.

[1]The join queries discussed in this paper are more precisely known as equi-joins.

They also prove that (under complexity assumptions) there is no random-order enumeration algorithm with linear preprocessing and polylogarithmic delay for non-free-connex conjunctive queries (such as cyclic join queries), and argue that it would be interesting to understand more precisely the time required (both in terms of enumeration delay and preprocessing time) for enumerating results of non-free-connex queries in random order.

For general join queries, a straightforward method [6, 13] is to develop a trivial transformation from a uniform sampling (with replacement) algorithm [12, 13, 23, 27] into a sampling without replacement process. This is typically done by treating the sampling algorithm as a black box that generates candidate results uniformly and discarding any duplicates that have already been seen. To sample (cyclic) join results, early works [23, 27] decompose the original cyclic query into an acyclic join query (say the skeleton query) and a residual component. They then apply an acyclic-join sampling algorithm on the skeleton query and employ a trivial procedure to sample the join results between the samples of the skeleton query and the residual component. Although these methods are practically motivated, they do not achieve near-optimal running time, and require at least linear time preprocessing for each query before sampling. More recent works [13, 16, 26] avoid decomposition and directly perform sampling on cyclic joins. These algorithms achieve near-optimal expected sampling time under complexity assumptions, without query-specific preprocessing. However, the constant and polylogarithmic factors of their running time are typically large, which scale exponentially with the query size (*e.g.*, the expected running time of [13] is actually $O(\frac{\mathrm{AGM}(Q)}{\max\{1,|Res(Q)|\}} \log^{d+1} |Q|)$, where $Res(Q)$ is the set of join results and d is the maximum arity of the relations). In addition, to the best of our knowledge, these algorithms have not yet been implemented and practically evaluated.

However, the sampling-based enumeration method wastes a lot of time in practice, as many generated result tuples will be discarded. There is also no worst-case guarantee on its total running time, and there is no strict guarantee that all result tuples can be output when the algorithm ends. Strictly speaking, it would not even be counted as an enumeration algorithm with a delay sublinear to the number of query results [9]. Worse still, sampling-based random-order enumeration inherits the aforementioned weaknesses of existing join sampling algorithms.

In this paper, we present an efficient random-order enumeration algorithm for join queries that provides a worst-case guarantee on the total running time. Theoretically, our algorithm enumerates the join result tuples in expected $O(\frac{\mathrm{AGM}(Q)}{|Res(Q)|+1} \log^2 |Q|)$ delay and $O(\mathrm{AGM}(Q) \log |Q|)$ total running time, after an $O(|Q| \log |Q|)$-time index construction phase. And it is proved to be a nearly worst-case optimal algorithm, and achieves the same total running time complexity with several worst-case optimal join algorithms such as Generic Join [22]. Practically, we develop techniques to speed up the enumeration and reduce the memory usage, which lead to a significant performance improvement in our experiments, especially when $\mathrm{AGM}(Q) \gg |Res(Q)|$. Our algorithm can not only serve as a near-optimal random-order enumeration algorithm without large constant or polylogarithmic factors in its enumeration delay and total running time; it can also be used as an efficient join sampling algorithm without preprocessing, *i.e.*, the indexes we build over the relational tables can be reused for different join queries, without the need for linear-time preprocessing for each query. Moreover, our algorithm does not require complex index structures built for relation tables. It supports various hierarchical indexing structures that are widely used in database systems, such as balanced trees, B-trees, skip lists, and tries. When updates are not required, simply sorting the tuples in lexicographic order suffices as a lightweight indexing mechanism for our algorithm. This flexibility enables our algorithm to be integrated into different database systems with minimal development overhead.

Concretely, we list our contributions in the following.

First, we develop an efficient random-order enumeration algorithm framework for join queries, based on a novel concept, RRAccess (the Relaxed Random-Access algorithm) and a well-organized data structure (the Ban-Pick tree), which are first proposed in this paper. Specifically, RRAccess is defined to calculate a bijection α from a set of integers S (not necessarily consecutive) to the set of join result tuples $Res(Q)$, and it returns a trivial interval I with $I \cap S = \emptyset$ when inputting an integer not in S. The Ban-Pick tree maintains a collection of disjoint intervals, referred to as the banned intervals, and it supports picking an integer from the remaining (unbanned) intervals uniformly at random. Based on the Ban-Pick tree, we show that if there is an RRAccess algorithm that runs in $O(\log^2 |Q|)$ worst-case time and $O(\log |Q|)$ amortized time and satisfies $\forall s \in S, s \le \mathrm{AGM}(Q)$, then our algorithm enumerates result tuples in $Res(Q)$ with expected $O(\frac{\mathrm{AGM}(Q)}{|Res(Q)|+1} \log^2 |Q|)$ delay and $O(\mathrm{AGM}(Q) \log |Q|)$ total running time. And it is proved as a nearly worst-case optimal algorithm since its expected enumeration delay and total running time are only polylogarithmically higher than the theoretical lower bound. Moreover, our framework enables practical speed-up techniques that substantially enhance the efficiency of enumeration in practice.

Second, we design an RRAccess algorithm based on a logical tree structure referred to as the relaxed random-access tree (RRATree), where each node of RRATree corresponds to a filter, and the set of result tuples satisfying a parent node's filter is recursively partitioned into subsets corresponding to its children's filters. By fully leveraging the properties of filters in RRATree, we construct efficient data structures and develop an upper-bound estimation algorithm and a children exploration algorithm, both of them run in $O(\log |Q|)$ time. These components enable RRAccess to achieve the $O(\log^2 |Q|)$ worst-case time and $O(\log |Q|)$ amortized time. Thus yielding the nearly worst-case optimal REnum. Moreover, we analyze the memory usage of REnum and introduce three techniques to reduce its practical memory overhead.

Third, based on our algorithm framework, two non-trivial techniques are developed to speed up the enumeration. The first is *larger trivial interval discovery* (LTI). Observe that the trivial integers in $\mathbb{N}[1, \mathrm{AGM}(Q)] \setminus \alpha^{-1}(Res(Q))$ are often continuous and can be grouped into intervals (called trivial intervals). LTI discovers large trivial intervals during the running of RRAccess to avoid picking more trivial integers in the follow-up steps. Consequently, the probability that RRAccess returns $\perp$ is reduced, so that the total running time and the enumeration delay are reduced. To further reduce the probability that RRAccess returns *perp*, we develop the second technique named *tighter upper-bound estimation* (TU). TU

provides a tighter upper-bound estimation of the number of the filtered join results for each filter in the RRATree, so that more and larger trivial intervals are discovered and banned earlier. As a result, it improves the effectiveness of LTI.

Finally, we experimentally evaluate the efficiency, memory usage, and scalability of our algorithm with the optimization techniques. The results show that with these techniques, our algorithm significantly outperforms the sampling-based methods [13, 23].

The remainder of this paper is structured as follows. The basic notations and some necessary techniques are introduced in Section 2. In Section 3 we introduce our main random-order enumeration algorithm framework. Section 4 details the RRATree and the RRAccess algorithm, establishes bounds on the enumeration delay and the total running time, and presents memory-reduction techniques. Section 5 discusses non-trivial techniques that significantly speed up the enumeration in practice. And Section 6 presents our experimental study. Due to space constraints, some proofs and details are omitted and provided in the technical report [11].

2 PRELIMINARIES

In this paper, we denote the set of all integers by $\mathbb{Z}$, and denote the set of all natural numbers by $\mathbb{N}$. For any natural numbers i and j with $i \leq j$, we define $\mathbb{N}[i, j] = [i, j] \cap \mathbb{N}$.

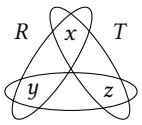
2.1 Join Query

Given a finite attribute set Att and a set $U \subseteq$ Att, a *tuple* over U is a function $t : U \to \mathbb{Z}$, and a projection of a tuple t on $V \subseteq U$, say $t[V]$, is a tuple that satisfies $t[V](v) = t(v)$ for each $v \in V$. A relation R is a set of tuples over an identical attribute set U, say $\text{att}(R) = U$. Then a join query Q is defined as a set of relations $\{R_1, \ldots, R_m\}$, and can be expressed as $Q = R_1 \bowtie \cdots \bowtie R_m$. Then let $|Q| = \sum_{i=1}^{m} |R_i|$ be the sum of sizes of the relations in Q (*i.e.* the size of input), and the result of the query Q is defined as $Res(Q) := \{t \text{ over } \text{att}(Q) | \forall R \in Q : t[\text{att}(R)] \in R\}$, where $\text{att}(Q) = \bigcup_{R \in Q} \text{att}(R)$. Let $\text{dom}_Q(v)$ denote the *active domain* of attribute v, *i.e.*, $\text{dom}_Q(v) = \bigcup_{R \in Q} \bigcup_{v \in \text{att}(R)} \{t(v) | t \in R\}$, then we have $Res(Q) \subseteq \prod_{v \in \text{att}(Q)} \text{dom}_Q(v)$.

2.2 AGM Bound

Intuitively, the AGM bound provides an upper bound on how large the join result could be using only the cardinalities of the input relations, without requiring knowledge of the actual data values. This characterizes the "worst-case" size of the join result, which is widely used for estimating the potential cost of query evaluation and for designing efficient join algorithms. Formally, given a join query Q, the *schema graph* of Q is defined as a hypergraph $G_Q = (V, E)$, where $V = \text{att}(Q)$ and $E = \{\text{att}(R) | R \in Q\}$. Let $c : E \to (0, 1)$ be a fractional edge cover of G_Q, *i.e.*, $\forall v \in V$, $\sum_{v \in e} c(e) \geq 1$. Then $|Res(Q)| \leq \text{AGM}_c(Q)$ [2], where $\text{AGM}_c(Q) := \prod_{R \in Q} |R|^{c(\text{att}(R))}$.

Moreover, let $EC(G_Q)$ denotes the set of all fractional edge covers of G_Q, the minimized AGM bound of Q, say $\text{AGM}(Q) = \min_{c \in EC(G_Q)} \text{AGM}_c(Q)$, is tight: there exists a join query Q^* that satisfies $|Res(Q^*)| = \Omega(\text{AGM}(Q^*))$ [2]. Note that the AGM bound can be efficiently computed with only the sizes of the relation tables, and the minimized AGM bound can be calculated by solving a linear program in $O(1)$ time under data complexity.



(a) G_{Q_Δ}

x	y
1	2
2	3
3	4
4	1

(b) R

y	z
1	3
3	4
4	4
4	1

(c) S

x	z
2	4
3	1
3	4
4	2

(d) T

x	y	z
2	3	4
3	4	1
3	4	4

(e) $Res(Q_\Delta)$

Table 1: $Q_\Delta := R \bowtie S \bowtie T$

2.3 Uniform Sampling

A *join sampling algorithm* outputs each join result tuple with an identical probability. Formally, a join sampling algorithm is a randomized algorithm $\mathcal{G}$ that takes a join query Q as input and outputs a tuple in $Res(Q)$, such that $\Pr(\mathcal{G}(Q) \text{ outputs } t) = \frac{1}{|Res(Q)|}$ for $\forall t \in Res(Q)$. By Deng et al. [13], there is a nearly worst-case optimal join sampling algorithm (under a complexity hypothesis):

Theorem 1 ([13]). *There is a uniform join sampling algorithm running in expected $\tilde{O}(\frac{\text{AGM}(Q)}{\max\{1, |Res(Q)|\}})$ time after a $\tilde{O}(|Q|)$-time index construction phase. Moreover, under the combinatorial k-clique hypothesis, for any $\varepsilon > 0$, there is no uniform sampling algorithm for join queries that runs in $\tilde{O}(|Q| + \frac{|Q|^{\rho^* - \varepsilon}}{|Res(Q)|})$ time with high probability, where ρ^* is the fractional edge cover number of G_Q.*

2.4 Random-Order Enumeration

The random-order enumeration algorithm for join queries is a random algorithm that takes a join query Q as input and outputs all tuples in $Res(Q)$ in random order. In other words, for each $1 \leq i < |Res(Q)|$, after the first i result tuples $t_1, \ldots, t_i$ have been output, the $(i+1)$-th output result tuple is a uniform sample of $Res(Q) \setminus \{t_1, \ldots, t_i\}$. Note that after all result tuples have been output, the algorithm may still need some additional computation to confirm that no further tuples remain. The (expected) enumeration delay of a random-order enumeration algorithm is defined as the maximum (expected) length of the following time intervals: (1) the time from the start of the algorithm to the output of the first result tuple, (2) the time from the output of any result tuple to the output of the next result tuple, and (3) the time from the output of the last result tuple to the termination of the algorithm.

3 ENUMERATION FRAMEWORK OVERVIEW

In this section, we develop an efficient random-order enumeration algorithm framework. To facilitate an intuitive exposition of our algorithm, we begin by defining a representative cyclic join query, denoted by Q_Δ, which will serve as a running example in the subsequent discussion.

Example 1 (Q_Δ). *Let $Q_\Delta := R \bowtie S \bowtie T$, in which $\text{att}(R) = \{x, y\}, \text{att}(S) = \{y, z\}$ and $\text{att}(T) = \{x, z\}$. The schema graph, relation tables, and join results of Q_Δ are shown in Table 1.*

A natural idea for random-order enumeration is to construct a bijection between a consecutive natural number set $\mathbb{N}[1, |Res(Q_\Delta)|]$ and the set of result tuples $Res(Q_\Delta)$. For example, let $Res(Q_\Delta)^\pi(i)$ denote the i-th tuple of $Res(Q_\Delta)$ in lexicographic order π. In this way, a random-order enumeration of $Res(Q_\Delta)$ can be obtained by

listing $Res(Q_\Delta)^\pi(\sigma(i))$ for $i = 1, \ldots, |Res(Q_\Delta)|$, where σ denotes a random permutation of integers from 1 to $|Res(Q_\Delta)|$. *E.g.*, if the random permutation of the integers in Example 1 is 2, 3, 1, then the enumerated tuples will be $(3, 4, 1)$, $(3, 4, 4)$, $(2, 3, 4)$. This approach works for acyclic join queries, as shown in [9], which presents such a bijection that can be calculated in polylogarithmic time after linear-time preprocessing. However, for general (especially cyclic) join queries, no such efficiently computable bijection exists under certain complexity hypotheses [9].

In our approach, we relax the requirement of constructing a bijection. Instead, we allow some integers in the domain not to correspond to any result tuple and to be mapped to "$\perp$". We refer to these integers as "trivial integers" and the rest as "non-trivial integers". We can efficiently test whether a given integer is trivial or not. It is worth noting that the mapping between all non-trivial integers and the result tuples is bijective. Thus, by sampling integers uniformly at random without replacement, and outputting the tuples corresponding to non-trivial integers in the sampled order, we obtain a valid random-order enumeration of the join results. To further improve efficiency, when a *trivial integer* is encountered, we identify and record a trivial interval to which it belongs. Once a trivial interval is discovered and recorded, the integers in it will no longer be sampled in the follow-up steps.

The remainder of this section will introduce (1) the formal definition of the mapping between integers and join results, and an algorithm to calculate it (*i.e.*, the relaxed random-access algorithm), (2) a novel data structure that supports online banning of intervals and uniform sampling of unbanned integers (*i.e.,* the Ban-Pick tree), and (3) an efficient random-order enumeration algorithm framework based on (1) and (2).

3.1 The Relaxed Random-Access Algorithm

Given a family of functions $\varphi = \{\varphi_Q | Q \in \mathcal{Q}\}$ and a join query Q, $\varphi_Q : \mathbb{N}^+ \to Res(Q) \cup \{\perp\}$ satisfies that for each tuple $t \in Res(Q)$ there exists only one $i \in \mathbb{N}^+, \varphi_Q(i) = t$. Let N be an upper-bound of $|Res(Q)|$ satisfying $\forall i > N, \varphi_Q(i) = \perp$. We observe that the integers in $\{i | \varphi^*(i) = \perp\}$, say the *trivial integers*, are often continuous and can be grouped into intervals (say the *trivial intervals*), especially when $\mathrm{AGM}(Q) \gg |Res(Q)|$. In order to prevent picking as many trivial integers as possible, the relaxed random-access algorithm reports the trivial intervals. Formally, we define $\mathrm{RRAccess}^{Q,\varphi}$ as the algorithm that takes an integer i as input and behaves as follows:

(1) returns $\varphi_Q(i)$ if $\varphi_Q(i) \neq \perp$,
(2) otherwise, returns a trivial interval $[a, b] \subseteq \mathbb{N}^+$ (with $a \leq i \leq b$) containing only trivial integers.

We will introduce the implementation of the relaxed random-access algorithm that runs in $O(\log^2 |Q|)$ worst-case time and $O(\log |Q|)$ amortized time in Section 4.

3.2 The Ban-Pick Tree

To avoid repeatedly picking integers that have already been picked or that are known not to correspond to any join result, we dynamically maintain a collection of intervals representing such integers, which are excluded from further picking. Specifically, we define two types of integers: (1) *trivial integers*, which do not correspond to any join result, and (2) *picked integers*, which have already been picked during previous steps. We maintain a set B of disjoint intervals, each consisting of either trivial or picked integers, using a data structure called the *Ban-Pick tree*. This structure, together with two operations, guarantees that newly picked integers do not fall within any interval in B. Formally, the Ban-Pick tree enables the following two operations:

(1) **the ban operation B.*ban***, that takes an interval disjoint from all intervals in B as input and inserts it into B,
(2) **the pick operation B.*pick***, that takes an integer H satisfying $\forall I \in B, I \subseteq [1, H]$ as input and returns an integer $i \in [1, H] \setminus \cup_{I \in B} I$ uniformly at random.

Based on the Ban-Pick tree, the trivial and picked integers in the banned intervals in B are prevented from being picked.

In general, the set $\mathbb{N}[1, N] \setminus \bigcup_{I \in B} I$ is not a contiguous interval. This results in a failure of the trivial generator working on an interval. Therefore, we need to provide a pick operation that works on a union of disjoint intervals. Given $B = \{I_i = [l_i, h_i] | i \in \mathbb{N}[1, |B|]\}$ which is the set of disjoint intervals already banned, *w.l.o.g.*, assume $h_i < l_{i+1}$ for every $i \in \mathbb{N}[1, |B| - 1]$. Let $L = |\mathbb{N}[1, N] \setminus \cup_{i=1}^{|B|} I_i|$, that is, $L = N - \sum_{i=1}^{|B|} |I_i|$. Our pick operation works as follows:

(1) sample an integer $y \in \mathbb{N}[1, L]$ uniformly at random,
(2) compute the offset $b = \sum_{i=1}^{k^*} |I_i|$ such that $h_{k^*} < y + b$ and $y + b < l_{k^*+1}$ (if $k^* < |B|$),
(3) return $y + b$.

To enable efficient computation of the offset in Step (2), we define the Ban-Pick tree as a balanced tree T_B such that

(1) each node $u \in T_B$ corresponds bijectively to an interval $I_u = [u.l, u.h] \in B$, and stores it via $u.l$ and $u.h$,
(2) each node $u \in T_B$ maintains $u.left$ and $u.right$ that point to its left child and right child, and if v is the left (right) child of u, then $v.h < u.l$ ($v.l > u.h$),
(3) each node $u \in T_B$ maintains $u.take$, which denotes the sum of lengths of the intervals in the subtree rooted in u,
(4) the height of T_B is $O(\log |B|)$.

Algorithm 1: *B.pick*

Input: H
Output: a uniform sample from $\mathbb{N}[1, H] \setminus \cup_{I \in B} I$

```
u ← the root of T_B;
sample an integer y ∈ ℕ[1, H − u.take] uniformly;
b ← 0, temp ← 0;
while u ≠ nil do
    if u.left = nil then temp ← 0;
    else temp ← u.left.take;
    if (y + b) + temp < u.l then u ← u.left;
    else b ← b + temp + (u.h − u.l + 1), u ← u.right;
return y + b
```

Then the ban operation takes $O(\log |B|)$ time for each interval insertion. We also design an efficient implementation of *B.pick* as Algorithm 1. The algorithm conceptually views the unbanned elements of $\mathbb{N}[1, H]$ as being concatenated into a single "compact available space" $\mathbb{N}[1, H - r.take]$, where r is the root of T_B. First, it samples a random integer y uniformly from $\mathbb{N}[1, H - r.take]$, then

it traverses T_B to locate the unbanned interval that corresponds to position y and computes the offset b, which equals the total length of all the preceding banned intervals. Finally, the algorithm returns $y+b$, which is the mapped value in the original space $\mathbb{N}[1,H]\setminus\cup_{I\in B}I$. For example, suppose $H=8$ and $B=\{[2,3],[6,6]\}$. The unbanned intervals are $[1,2]$, $[4,5]$ and $[7,8]$, whose concatenation forms a compact available space $\mathbb{N}[1,5]$. If $y=4$ is sampled from $\mathbb{N}[1,5]$, it corresponds to the third unbanned interval, $[7,8]$. Therefore, the algorithm returns $y+b=4+(|[2,3]|+|[6,6]|)=7$.

Since the height of T_B is at most $O(\log|B|)$, Algorithm 1 runs in $O(\log|B|)$ time. Moreover, since y is uniformly sampled from the compact available space, the probability of y corresponding to a particular unbanned interval is proportional to its size, and $y+b$ is uniformly distributed within that interval. Hence, Algorithm 1 produces a uniform sample from the set $\mathbb{N}[1,H]\setminus\bigcup_{I\in B}I$.

Remarks. The Ban-Pick tree is independent of the underlying data storage and access mechanisms, enabling integration into different database systems without code modification.

3.3 The Enumeration Framework

Given any join query Q, let N be an upper bound of $|Res(Q)|$ such that $\forall i>N, \varphi_Q(i)=\perp$. Algorithm 2 enumerates all tuples in $Res(Q)$ by repeatedly picking integers uniformly at random from $[1,N]\setminus B$, where B is a set of banned intervals containing trivial and picked integers. Each picked integer i is passed to $\text{RRAccess}^{Q,\varphi}$. If $\varphi_Q(i)\neq\perp$, in which case $\text{RRAccess}^{Q,\varphi}$ returns a valid result tuple, then the algorithm outputs the tuple, and $[i,i]$ is inserted into B to avoid repetition; otherwise, if $\varphi_Q(i)=\perp$, indicating that i lies in a trivial interval I returned by $\text{RRAccess}^{Q,\varphi}$, then the entire interval I is inserted into B. This process ensures that each result tuple is output exactly once. Moreover, at each step, the output tuple (if exists) is uniformly sampled from the set of the unenumerated join results.

Algorithm 2: REnum

Input: Q
Output: $Res(Q)$ in a random order

$B\leftarrow\emptyset$, $N_{\text{ban}}\leftarrow 0$;
while $N_{\text{ban}}<N$ **do**
  $i\leftarrow B.pick(N)$;
  $res\leftarrow \text{RRAccess}^{Q,\varphi^*}(i)$;
  if *res is a tuple in* $Res(Q)$ **then**
    output *res*;
    $B.ban([i,i])$, $N_{\text{ban}}\leftarrow N_{\text{ban}}+1$;
  else $B.ban(res)$, $N_{\text{ban}}\leftarrow N_{\text{ban}}+|res|$;

Lemma 1. *For any $Q\in\mathcal{Q}$, if $\log N\le O(\log|Q|)$ and there exists a relaxed random-access algorithm $\text{RRAccess}^{Q,\varphi}$ running in $O(\log^2|Q|)$ worst-case time and $O(\log|Q|)$ amortized time, then Algorithm 2 enumerates the tuples in $Res(Q)$ in random order with expected $O(\frac{N}{|Res(Q)|+1}\log^2|Q|)$ delay and $O(N\log|Q|)$ total time.*

In the case where $N\le \text{AGM}(Q)$, that is, $\text{RRAccess}^{Q,\varphi}$ in Lemma 1 satisfies $\forall i>\text{AGM}(Q), \varphi(i)=\perp$, Algorithm 2 enumerates the join results in random order with expected $O(\frac{\text{AGM}(Q)}{|Res(Q)|+1}\log^2|Q|)$ delay and $O(\text{AGM}(Q)\log|Q|)$ total running time, where ρ^* is the fractional edge cover number of G_Q.

In addition, by [13], the $O(\text{AGM}(Q)\log|Q|)$ total running time is nearly optimal in the worst case, unless the combinatorial k-clique hypothesis is false. Moreover, since the first enumerated result tuple is a uniform sample from $Res(Q)$, the lower bound of the expected running time of join sampling algorithms is also a lower bound of the expected delay of random-order enumeration algorithms. Formally, the following theorem holds.

Theorem 2. *Under the combinatorial k-clique hypothesis, for any $\varepsilon>0$, there is no random-order enumeration algorithm for join queries with expected $\tilde{O}(\frac{|Q|^{\rho^*-\varepsilon}}{|Res(Q)|+1})$ delay after a $\tilde{O}(|Q|)$-time preprocessing, where ρ^* is the minimal fractional edge cover number of G_Q.*

This theorem implies that, after a $\tilde{O}(|Q|)$-time preprocessing phase, the expected $O(\frac{\text{AGM}(Q)}{|Res(Q)|+1}\log^2|Q|)\le O(\frac{|Q|^{\rho^*}}{|Res(Q)|+1}\log^2|Q|)$ delay is nearly optimal.

4 THE RELAXED RANDOM-ACCESS

In this section, we present an efficient implementation of RRAccess, which achieves $O(\log^2|Q|)$ worst-case time and $O(\log|Q|)$ amortized time. We then analyze the resulting REnum algorithm and introduce techniques to reduce its space usage. To achieve these guarantees, RRAccess is built on a conceptual tree structure called *relaxed random-access tree (RRATree)*, denoted by $\tilde{T}_Q$.

4.1 Overview of RRATree and RRAccess

Intuitively, the RRATree is a conceptual tree that recursively divides the active domain of the join query.The root represents the full active domain, each node corresponds to a subset of the domain characterized by a filter, and each join result corresponds to a leaf. Formally, the RRATree is defined as follows:

Definition 1 (RRATree). *Let Q be an arbitrary join query. The RRATree of Q, denoted by $\tilde{T}_Q$, is a rooted tree that satisfies:*

1. *each node $u\in\tilde{T}_Q$ corresponds to a filter ψ_u,*
2. *let r be the root node of $\tilde{T}_Q$, then the filter ψ_r can be computed in $O(|Q|)$ time and satisfies $Q=Q|_{\psi_r}$,*
3. *there is an upper-bound algorithm upp which takes a filter ψ_u with $u\in\tilde{T}_Q$ as input and returns a positive integer in $O(\log|Q|)$ time, such that*
   (a) *$\forall u\in\tilde{T}_Q, upp(\psi_u)\ge|Res(Q|_{\psi_u})|$,*
   (b) *for any filter ψ_u with $u\in\tilde{T}_Q$ and $upp(\psi_u)\le 1$, $Res(Q|_{\psi_u})$ can be computed in $O(\log|Q|)$ time,*
4. *there is a children exploration algorithm $children$, such that for each node $u\in\tilde{T}_Q$, $children(\psi_u)$ returns at most a constant number of filters $\psi_{v_1},\dots,\psi_{v_k}$ in $O(\log|Q|)$ time, such that*
   (a) *$\forall u\in\tilde{T}_Q, children(\psi_u)=\emptyset$ iff $upp(\psi_u)\le 1$,*
   (b) *$\bigcup_{i=1}^{k}Res(Q|_{\psi_{v_i}})=Res(Q|_{\psi_u})$,*
   (c) *$\forall 1\le i<j\le k, Res(Q|_{\psi_{v_i}})\cap Res(Q|_{\psi_{v_j}})=\emptyset$,*
   (d) *$\sum_{i=1}^{k}upp(\psi_{v_i})\le upp(\psi_u)$,*
   (e) *$\forall 1\le i\le k, upp(\psi_{v_i})\le\frac{1}{2}upp(\psi_u)$.*

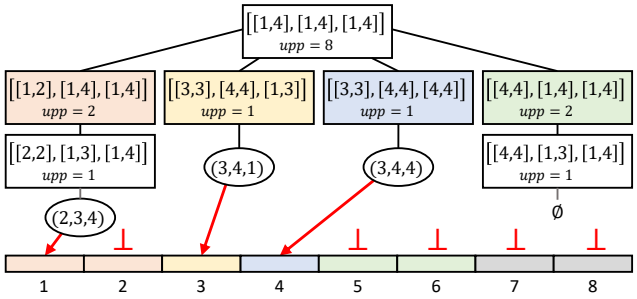


Figure 1: $\tilde{T}_{Q_\Delta}$ **when** $upp(\psi) = \lfloor \mathrm{AGM}_{c^*}(Q_\Delta|_\psi) \rfloor$

Figure 1 shows the RRATree of Q_Δ, where the root node represents the entire active domain of Q_Δ, and its four children represents four disjoint subsets of the domain, each defined by a filter.

Based on the RRATree, we define the function family φ^* with upp and develop the algorithm RRAccess. Intuitively, in Figure 1, as the upper bound of the root node is 8, it is associated with the interval $\mathbb{N}[1, 8]$. And its four children are associated with four disjoint subintervals whose lengths correspond to their respective upper bounds, distinguished by the colors in the figure. This approach recurses until every result tuple is assigned to a unique integer, which in turn maps to that tuple in $\varphi^*_{Q_\Delta}$; the remaining integers are then mapped to $\perp$. Formally, given a join query Q and fix a partial order relation $\prec$ in the filter set, for each integer $i > 0$, if there is a tuple $t \in Res(Q)$ such that

$$i = \sum_{j=1}^{h-1} \sum_{\psi \in S_j} upp(\psi) + 1, \tag{1}$$

where $S_j = \{\psi \in \text{children}(\psi_{u_j}) \mid \psi \prec \psi_{u_{j+1}}\}$, $u_1, \ldots, u_h$ is the path from the root r to the leaf u_t ($u_1 = r$ and $u_h = u_t$), and $u_t \in \tilde{T}_Q$ is the leaf node such that $t \in Res(Q|_{\psi_{u_t}})$, then $\varphi^*_Q(i) = t$. Otherwise, if there does not exist such a tuple, we define $\varphi^*_Q(i) = \perp$. Then the following lemma holds.

Lemma 2. $\forall i > upp(\psi_r)$, $\varphi^*_Q(i) = \perp$.

Our relaxed random-access algorithm is implemented as Algorithm 3. The algorithm starts from ψ_r, which can be computed in $O(1)$ time after an $O(|Q| \log |Q|)$-time index construction phase. Then in lines 3-8, the algorithm traverses $\tilde{T}_Q$ from top to down to find the path on which the filters may contain $\varphi^*_Q(i)$. Since for each filter ψ and its children $children(\psi) = \psi_1, \ldots, \psi_k$, we have $upp(\psi_i) \leq \frac{1}{2} upp(\psi)$ for any $1 \leq i \leq k$, which implies that the depth of $\tilde{T}_Q$ is at most $O(\log upp(\psi_r))$ and the number of nodes in $\tilde{T}_Q$ is $O(upp(\psi_r))$. Moreover, with the use of a caching mechanism that stores the filters, upper bounds and children of all traversed nodes in $\tilde{T}_Q$ (thus avoiding redundant computation), in the case where $upp(\psi_r) \leq \mathrm{AGM}(Q)$, the following lemma holds.

Lemma 3. *If ψ_r can be computed in $O(1)$ time, and there exist an algorithm upp and an algorithm children satisfying the properties in Definition 1 and running in $O(\log |Q|)$ time, then if $upp(\psi_r) \leq \mathrm{AGM}(Q)$, Algorithm 3 is a relaxed random-access algorithm running in $O(\log^2 |Q|)$ worst-case time and $O(\log |Q|)$ amortized time.*

Algorithm 3: RRAccess$^{Q,\varphi^*}$

Input: i
Output: $\varphi^*_Q(i)$ if $\varphi^*_Q(i) \neq \perp$, otherwise a trivial interval containing i

1 $offset \leftarrow 0$;
2 $\psi \leftarrow \psi_r$ where r is the root node of $\tilde{T}_Q$;
3 **while** $upp(\psi) \geq 2$ **do**
4 $\psi_1, \ldots, \psi_k \leftarrow children(\psi)$;
5 **if** $offset + \sum_{j=1}^{k} upp(\psi_k) < i$ **then return** $[i, i]$;
6 $r^* \leftarrow \min\{r \in \mathbb{N}[1, k] | offset + \sum_{j=1}^{r} upp(\psi_j) \geq i\}$;
7 $offset \leftarrow offset + \sum_{j=1}^{r^*-1} upp(\psi_j)$;
8 $\psi \leftarrow \psi_{r^*}$;
9 **if** $Res(Q|_\psi) \neq \emptyset$ **then**
10 **return** the tuple in $Res(Q|_\psi)$
11 **else return** $[i, i]$;

Remarks. For simplicity, the RRAccess algorithm in this section only implements the trivial interval discovery of a naïve method. Specifically, when $\varphi_Q(i) = \perp$, the algorithm returns the singleton interval $[i, i]$. In Section 5, we will introduce techniques that efficiently discover larger trivial intervals within the running of RRAccess, thereby enhancing the practical efficiency of the enumeration.

In the following, we introduce the proper filters of the tree nodes, the $O(\log |Q|)$-time upper-bound algorithm and the $O(\log |Q|)$-time children exploration algorithm.

4.2 The Filters of the Tree Nodes

In this paper, all filters corresponding to the nodes in $\tilde{T}_Q$ are defined to be range filters. Formally, for any node $u \in \tilde{T}_Q$, the filter ψ_u can be expressed as a list of intervals: $\psi_u = [[l_{u,1}, h_{u,1}], \ldots, [l_{u,n}, h_{u,n}]]$. We further say that ψ_u is an *empty* range filter (or an *empty range* for short) if and only if there exists $1 \leq i \leq n$ such that $l_{u,i} > h_{u,i}$. A tuple $t \in R$ satisfies ψ_u if and only if $x_i \in [l_{u,i}, h_{u,i}]$ holds for each $x_i \in \mathrm{att}(R)$. In Example 1, let $\psi = [[1, 2], [1, 4], [1, 4]]$, we have $R|_\psi = \{(1, 2), (2, 3)\}$, $S|_\psi = S$, $T|_\psi = \{(4, 2)\}$, and then $Res(Q_\Delta|_\psi) = \{(2, 3, 4)\}$. Moreover, the filter of the root node r is $\psi_r = \big[[\min_Q(x_1), \max_Q(x_1)], \ldots, [\min_Q(x_n), \max_Q(x_n)]\big]$, where $\max_Q(x_i) = \max \mathrm{dom}_Q(x_i)$ and $\min_Q(x_i) = \min \mathrm{dom}_Q(x_i)$ for $1 \leq i \leq n$. These bounds are computed during index construction in $O(|Q|)$ time. Once the index is constructed, ψ_r can be obtained in $O(1)$ time, and $Q|_{\psi_r} = Q$ as all tuples in $\bigcup_{R \in Q} R$ satisfy ψ_r.

We further define an important class of range filters, say the "*prefix range filters*", on which the upper-bound algorithm and the children exploration algorithm can be calculated efficiently.

Definition 2. *For any range filter $\psi = [[l_1, h_1], \ldots, [l_n, h_n]]$, if there exists an integer $1 \leq s \leq n$ such that ψ satisfies (1) $\forall 1 \leq i < s$, $l_i = h_i$, (2) $l_s \leq h_s$, and (3) $\forall s \leq i \leq n$, $l_i = \min_Q(x_i), h_i = \max_Q(x_i)$, then ψ is a* prefix range filter *with a* split position s. *Moreover, if $s + 1$ is not a split position of ψ, then s is the* maximum split position *of ψ.*

It is clear that ψ_r is a prefix range filter with split position 1. We will later discuss the advantage properties of the prefix range filters and prove that all filters in the RRATree are prefix range filters.

4.3 The Upper-Bound Algorithm

Let G_Q be the schema graph of Q, given a fractional edge cover $c \in EC(G_Q)$, for the sake of ease in theoretical analysis, we initially define $upp(\psi) = \lfloor \mathrm{AGM}_c(Q|_\psi) \rfloor$ for any range filter ψ. Since all the sizes of relations are integers, the floor of the AGM bound still serves as an upper-bound for the size of the join result. Then it holds immediately that $upp(\psi) \geq |Res(Q|_\psi)|$. In Section 5, we introduce other upper bounds of the join result size. Tighter upper bounds will enhance the practical efficiency of the algorithm, *i.e.*, reducing both the enumeration delay and the total running time.

Deng et al. [13] build a range tree as an index for each relation R with $|\mathrm{att}(R)| = d$ in $O(|R| \log^{d-1} |R|)$ time in the index construction phase, then for any range filter ψ, $|R|_\psi|$ can be computed in $O(\log^{d-1} |R|)$ time. We now demonstrate that, for any prefix range filter ψ, the AGM bound of the filtered query $Q|_\psi$ can be computed in $O(\log |Q|)$ time. Since the AGM bound can be computed in $O(1)$ time once the cardinalities $|R|_\psi|$ for each $R \in Q$ are obtained, it suffices to show that these cardinalities can be computed within the stated time bounds.

For each $R \in Q$ with attributes $\mathrm{att}(R) = \{x_{r_1}, \ldots, x_{r_d}\}$ where $d = |\mathrm{att}(R)|$ and $r_1, \ldots, r_d \in \mathbb{N}[1, n]$. Assume that all tuples in R are ordered lexicographically. In particular, $\forall t, t' \in R, t \prec t'$ iff $\big(t(x_{r_1}), \ldots, t(x_{r_d})\big) \prec \big(t'(x_{r_1}), \ldots, t'(x_{r_d})\big)$ in lexicographical order. Then we define the functions $R.lower : \mathbb{Z}^d \to \mathbb{N}$ and $R.upper : \mathbb{Z}^d \to \mathbb{N}$. For any d-ary tuple t (which may not necessarily be a member of R), $R.lower(t)$ returns the index of the first tuple $t' \in R$ such that $t' \succeq t$, and $R.upper(t)$ returns the index of the first tuple $t' \in R$ such that $t' \succ t$. Define $R[t_l, t_h] = \{t \in R | t_l \preceq t \preceq t_h\}$ and $R.cnt(\psi) = |R[t_l^\psi, t_h^\psi]| = R.upper(t_h^\psi) - R.lower(t_l^\psi)$, where $\psi = [[l_1, h_1], \ldots, [l_n, h_n]]$, $t_l^\psi = (l_{r_1}, \ldots, l_{r_d})$ and $t_h^\psi = (h_{r_1}, \ldots, h_{r_d})$, then the following lemma holds.

Lemma 4. *For any prefix range filter ψ, $|R|_\psi| = R.cnt(\psi)$.*

We now present the index for each $R \in Q$ that supports $O(\log |R|)$-time computation of $R.cnt$. A straightforward approach to calculate $R.cnt$ is to maintain a lexicographically sorted array of all tuples in R. Given such an array, for any prefix range filter ψ, both $R.lower(t_l^\psi)$ and $R.upper(t_h^\psi)$ can be calculated in $O(\log |R|)$ time via a binary search, then $R.cnt(\psi)$ can be calculated in $O(\log |R|)$ time.

However, this approach is inefficient for updates, as each insertion or deletion may require $O(|R|)$ element shifts. To support both efficient query and update operations, we adopt a self-balancing binary search tree (BST), *e.g.*, an AVL tree, as the underlying index structure. Each node of the tree corresponds one-to-one to a tuple in the relation table and maintains the binary search property: the tuple in each node is lexicographically greater (less) than that in its left (right) child, if such children exist. Furthermore, each node u stores the size of the subtree rooted in itself, denoted as $u.size$, which enables efficient counting of tuples within a specified range in the relation. As both tree balancing and subtree size maintenance can be performed in $O(\log |R|)$ time per insertion or deletion, the index structure efficiently supports dynamic update with logarithmic overhead. Using the BST structure and the stored sizes, $R.lower(t)$ and $R.upper(t)$ can be calculated in $O(\log |R|)$ time for any tuple $t \in \mathbb{Z}^d$ by performing a root-to-leaf traversal. That is, for any prefix range filter ψ, $R.cnt(\psi) = R.upper(t_h^\psi) - R.lower(t_l^\psi)$ can be calculated in $O(\log |R|)$ time. Then we have

Lemma 5. *For any prefix range filter ψ and any fractional edge cover $c \in EC(G_Q)$, both $\mathrm{AGM}_c(Q|_\psi)$ and $\mathrm{AGM}(Q|_\psi)$ can be calculated in $O(\log |Q|)$ time.*

Therefore, for any prefix range filter ψ, $upp(\psi) = \lfloor \mathrm{AGM}_c(Q|_\psi) \rfloor$ can be computed in $O(\log |Q|)$ time.

We now aim to prove that when $upp(\psi) \leq 1$, the join result $Res(Q|_\psi)$ can be computed in $O(\log |Q|)$ time. To this end, we first define a property of upp, called super-additivity, and prove that when upp satisfies the property, $Res(Q|_\psi)$ can be computed in $O(\log |Q|)$ time for any ψ satisfying $upp(\psi) \leq 1$.

Property 1 (super-additivity). *Given any range filter $\psi = [[l_1, h_1], \ldots, [l_n, h_n]]$ and $1 \leq p \leq n$, for any partition of the interval $[l_p, h_p]$ into k disjoint sub-intervals $I_1, \ldots, I_k$ such that $[l_p, h_p] = \bigcup_{i=1}^k I_i$ and $I_i \cap I_j = \emptyset$ for $i \neq j$, the inequality $\sum_{i=1}^k upp([[l_1, h_1], \ldots, I_i, \ldots, [l_n, h_n]]) \leq upp(\psi)$ holds.*

If an upper-bound algorithm satisfies Property 1, it is referred to as *super-additive*. Then the following lemma holds.

Lemma 6. *If upp is super-additive, then for any prefix range filter $\psi = [[l_1, h_1], \ldots, [l_n, h_n]]$ such that $upp(\psi) \leq 1$, $Res(Q|_\psi)$ can be computed in $O(\log |Q|)$ time.*

Moreover, the upper-bound algorithm proposed in this section is super-additive. This can be readily derived from the AGM split theorem [13]. Hence, for any prefix range filter ψ with $upp(\psi) \leq 1$, $Res(Q|_\psi)$ can be computed in $O(\log |Q|)$ time.

4.4 The Children Exploration Algorithm

During the path-finding process of RRAccess, in order to calculate the children of each visited node in $O(\log |Q|)$ time, we develop an efficient children exploration algorithm, which takes a prefix range filter ψ as input and outputs at most a constant number of children filters $\psi_1, \ldots, \psi_k$ as required in Definition 1. Our approach follows the active domain partitioning strategy proposed by Deng et al. [13]. We improve their method by introducing a novel partitioning algorithm and more efficient data structures, which together reduce the computational complexity from $O(\log^d |Q|)$ to $O(\log |Q|)$, where $d = \max_{R \in Q} \mathrm{att}(R)$ denotes the maximum arity of the relations.

The divide operation. Specifically, define an operation "divide" that takes a range filter $\psi = [[l_1, h_1], \ldots, [l_n, h_n]]$ with $s = \min\{i | l_i \neq h_i\}$ as input and outputs three range filters $\psi_{\text{left}}, \psi_{\text{mid}}$ and ψ_{right} of the form $[[l_1, h_1], \ldots, [l'_s, r'_s], \ldots, [l_n, h_n]]$, where $[l'_s, r'_s]$ is $[l_s, p-1]$, $[p, p]$ and $[p+1, h_s]$ respectively, such that

(1) $upp(\psi_{\text{left}}) + upp(\psi_{\text{mid}}) + upp(\psi_{\text{right}}) \leq upp(\psi)$,
(2) $upp(\psi_{\text{left}}) \leq \frac{1}{2} upp(\psi), upp(\psi_{\text{right}}) \leq \frac{1}{2} upp(\psi)$.

Assuming the upper-bound algorithm is super-additive, then (1) holds for any $p \in \mathbb{N}[l_s, h_s]$. Deng et al. [13] calculate the proper division point p that satisfies (2) by a binary search for the minimized integer in $[l_s, h_s]$ such that $upp(\psi_{\text{left}}) \geq \frac{1}{2} upp(\psi)$. In each iteration of their binary search process, a count oracle implemented by a range tree is employed to calculate the cardinality of each filtered relation. This results in a time complexity of $O(\log^d |Q|)$ for the divide operation, where $d = \max_{R \in Q} |\mathrm{att}(R)|$. In contrast, we show

that when dividing a prefix range filter, the division point p can be computed in $O(\log |Q|)$ time. We achieve this by first demonstrating that s is a valid split position of ψ, then reducing the problem to an equivalent optimization problem, and finally designing an efficient $O(\log |Q|)$-time algorithm to solve it.

Lemma 7. *Let $\psi = [[l_1, h_1], \ldots, [l_n, h_n]]$ be a prefix range filter, then $s = \min\{i | l_i \neq h_i\}$ is a split position of ψ.*

For each relation $R_i \in Q$ satisfying $x_s \in \text{att}(R_i)$, define $A_i = \left[t_1(x_s), \ldots, t_{n_i}(x_s)\right]$, where $n_i = |R_i|_\psi|$, $R|_\psi = \{t_1, \ldots, t_{n_i}\}$ and $t_1 \preceq \cdots \preceq t_{n_i}$. We observe that the task of "calculating the division point p" in the divide operation can be cast as an instance of the following optimization problem:

Input: a constant number of sorted integer arrays $A_1, \ldots, A_k$ and an integer T,

Output: the maximum integer p^* such that $F(p^*) \leq T$,

where $N_i(p) = |\{x \in A_i | x < p\}|$, and $F : \mathbb{N}^k \to \mathbb{N}$ is an k-ary non-decreasing function (For simplicity, we use $F(p)$ as the shorthand for $F(N_1(p), \ldots, N_k(p))$). The reduction is obvious since *upp* can be viewed as a non-decreasing function over the cardinalities of the filtered relations R_{left} for each relation $R \in Q$ with $x_s \in \text{att}(R)$.

Algorithm 4: Multi Head Binary Search (MHBS)

Input: $A_1, \ldots, A_k, T$
Output: maximum p^* such that $F(p^*) \leq T$

if $F(|A_1|, \ldots, |A_k|) \leq T$ **then return** $+\infty$;
for $1 \leq i \leq k$ **do**
  if $r_i - l_i \leq 1$ **then**
    if $F(A_i[l_i] + 1) > T$ **then return** $A_i[l_i]$;
    else $m_i \leftarrow r_i$;
  else $l_i \leftarrow 0, r_i \leftarrow |A_i|, m_i \leftarrow \lfloor \frac{l_i + r_i}{2} \rfloor$;
while $\exists i, r_i - l_i > 1$ **do**
  $i_{\min} \leftarrow \arg\min_{r_i - l_i > 1} A_i[m_i]$, $i_{\max} \leftarrow \arg\max_{r_i - l_i > 1} A_i[m_i]$;
  if $F(m_1, \ldots, m_k) \leq T$ **then** $l_{i_{\min}} \leftarrow m_{i_{\min}}$;
  else $r_{i_{\max}} \leftarrow m_{i_{\max}}$;
  for $i \in \{i_{\min}, i_{\max}\}$ **do** $m_i \leftarrow \lfloor \frac{l_i + r_i}{2} \rfloor$;
  if $\exists i \in \{i_{\min}, i_{\max}\}, r_i - l_i \leq 1$ **then**
    if $F(A_i[l_i] + 1) > T$ **then return** $A_i[l_i]$;
    else $m_i \leftarrow r_i$;
return $\min_{1 \leq i \leq k} A_i[r_i]$;

Assuming for each relation table $R \in Q$, all tuples in R are sorted lexicographically, we propose the Multi-Head Binary Search (MHBS) algorithm (see Algorithm 4), which solves the above problem in $O(\log \sum_{i=1}^k |A_i|)$ time. Intuitively, the MHBS algorithm maintains a binary search interval for each array and iteratively halves the interval of one array chosen by a selection criterion. The process continues until all the intervals converge to the optimal solution.

Lemma 8. *Algorithm 4 returns the maximum integer p^* such that $F(p^*) \leq T$ in $O(\log \sum_{i=1}^k |A_i|)$ time.*

Since $\sum_{i=1}^k |A_i| \leq \sum_{i=1}^k |R_i| \leq |Q|$, it follows immediately that

Corollary 1. *The divide operation on any prefix range filter can be performed in $O(\log |Q|)$ time.*

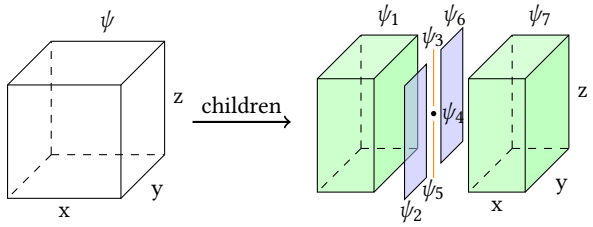


Figure 2: Devision of ψ of Q_Δ.

Remarks. The assumption that all tuples in each relation table are sorted lexicographically is made for analytical convenience and does not entail any loss of generality, since other hierarchical indexing structures (such as balanced trees, B-trees, skip lists, and tries) can be supported with minor implementation modifications in a manner similar to that discussed in Section 4.3.

Algorithm 5: *children*

Input: ψ
Output: a list of filters

if $upp(\psi) \leq 1$ **then return** $\emptyset$;
$res \leftarrow \emptyset$;
divide ψ into $\psi_{\text{left}}, \psi_{\text{mid}}, \psi_{\text{right}}$;
if $upp(\psi_{\text{left}}) > 0$ *and* ψ_{left} *is not empty* **then**
  $res \leftarrow res \cup \{\psi_{\text{left}}\}$
if $upp(\psi_{\text{mid}}) = 1$ **then** $res \leftarrow res \cup \{\psi_{\text{mid}}\}$;
else $res \leftarrow res \cup children(\psi_{\text{mid}})$;
if $upp(\psi_{\text{right}}) > 0$ *and* ψ_{right} *is not empty* **then**
  $res \leftarrow res \cup \{\psi_{\text{right}}\}$
return res;

Then for any prefix range filter ψ, $children(\psi)$ can be calculated in a recursive way in $O(\log |Q|)$ time, as shown in Algorithm 5. Since the recursive depth is at most d, Algorithm 5 returns at most $2d + 1 \leq O(1)$ filters. In Example 1, a prefix range filter ψ can be split into at most 7 filters, as illustrated in Figure 2. Let $upp(\psi) = \lfloor \text{AGM}_{c^*}(Q|_\psi) \rfloor$ for any filter ψ where $c^*(R) = c^*(S) = c^*(T) = \frac{1}{2}$, $\tilde{T}_{Q_\Delta}$ is finally constructed as shown in Figure 1. Moreover, by the definition of the division operation and Property 1, it is easy to prove that the returned filters satisfy the properties in Definition 1.

Finally, we establish that all filters in $\tilde{T}_Q$ are prefix range filters.

Lemma 9. *For any prefix range filter ψ, let ψ_{left}, ψ_{mid} and ψ_{right} denote the three filters obtained by dividing ψ, then all non-empty ranges among these filters are prefix range filters.*

Since the filter of the root node (*i.e.*, ψ_r) is a prefix range filter, it follows by induction that

Corollary 2. *For any node $u \in \tilde{T}_Q$, ψ_u is a prefix range filter.*

4.5 The Near-Optimal REnum Algorithm.

The above implementations provide a theoretical guarantee of the performance of REnum. Then the following theorem can be proven by analyzing Algorithm 2 under the setting where $N = upp(\psi_r)$ and $upp(\psi) = \left\lfloor \text{AGM}_{c^*}(Q|_\psi) \right\rfloor$ for any range filter ψ.

Theorem 3. *There exists a constructive random-order enumeration algorithm for join queries with expected* $O(\frac{\mathrm{AGM}(Q)}{|Res(Q)|+1}\log^2|Q|)$ *delay and* $O(\mathrm{AGM}(Q)\log|Q|)$ *total running time, after an* $O(|Q|\log|Q|)$*-time index construction phase.*

Theorem 2 and the results from [13] demonstrate that, under the combinatorial k-clique hypothesis, both the expected delay and the total running time of Algorithm 2 are nearly worst-case optimal, after an $O(|Q|\log|Q|)$-time index construction phase.

Remarks. Our algorithm is not only applicable to static relations, but also efficiently supports evolving environments where data are frequently inserted, deleted, or updated. As discussed in Sections 4.3 and 4.4, the index components of our framework can be implemented using dynamic data structures such as balanced trees, B-trees, skip lists, and tries. With such dynamic indexes, the proposed algorithm supports logarithmic-time updates, making it well suited for modern data systems with hybrid workloads.

4.6 Space Usage.

The space complexity of our algorithm is dominated by two components: (1) the ban-pick tree and (2) the cached RRATree structure. Since both the ban-pick tree and the RRATree contain at most $O(\mathrm{AGM}(Q))$ nodes, and each node occupies $O(1)$ space on average, the overall space complexity is $O(\mathrm{AGM}(Q))$. In practice, we propose three techniques for reducing space overhead while largely preserving the performance benefits of the caching mechanism. All these techniques are implemented and evaluated in Section 6.

Merging banned intervals. When banning a new interval I, *B.ban* traverses the ban-pick tree from root to leaf to locate its position. During this process, if I can be merged with the interval of an existing node, we update the node with the merged interval, thus avoiding the need to create a new node for I. This strategy preserves the correctness of the tree while reducing space usage.

On-demand filter release. Once an RRATree node has been processed by the children exploration algorithm, its associated filter will no longer be required in the subsequent process. Therefore, the memory occupied by these filters can be released, reducing the overall space overhead.

Depth-bounded caching. In the enumeration process, the nodes in RRATree with smaller depths are accessed much more frequently than those with larger depths, and the number of nodes increases rapidly with depth. Therefore, under limited memory budgets, we cache only the nodes within a certain depth threshold, so that the memory can be utilized most effectively to store frequently accessed nodes, thus reducing redundant recomputation as much as possible.

5 SPEED UP THE ENUMERATION

Evidently, large $\mathrm{AGM}(Q)$ and $\frac{\mathrm{AGM}(Q)}{|Res(Q)|+1}$ lead to long enumeration delay and total running time of REnum. To address this bottleneck, based on our enumeration framework, we propose two speed-up techniques that significantly improve its efficiency in practice.

5.1 Larger Trivial Interval Discovery

We observe that trivial intervals often appear as consecutive sequences and thus can be grouped into larger intervals, particularly when $\mathrm{AGM}(Q) \gg |Res(Q)|$. To avoid the frequent picking of such trivial integers during enumeration, we propose the Larger Trivial Interval discovery (LTI) technique. In contrast to the previous implementation of RRAccess, which returns only a single-point interval $[i,i]$ when $\varphi_Q(i)=\perp$, LTI discovers larger trivial intervals, so that more trivial integers are prevented from being picked in the follow-up steps, thereby speeding up the enumeration.

The trivial intervals of filters. Observe that when $\varphi^*_Q(i)=\perp$, Algorithm 3 may return a trivial interval only at line 5 or line 11. By the definition of $\tilde{T}_Q$, for each filter ψ satisfying $children(\psi)=\psi_1,\ldots,\psi_k$, we have $upp(\psi)\geq\sum_{i=1}^{k}upp(\psi_i)$. Then if an integer i (after removing the offset) falls within $(\sum_{i=1}^{k}upp(\psi_i), upp(\psi)]$, it follows that $\varphi^*_Q(i)=\perp$. That is, such intervals should be discovered and banned in the follow-up steps. Then for line 5, since for any integer $offset+\sum_{j=1}^{k}upp(\psi_k)<t\leq offset+upp(\psi)$ we have $\varphi^*_Q(t)=\perp$, we define the trivial interval of the filter ψ as

$$I_\psi=\left[offset+\sum_{j=1}^{k}upp(\psi_k)+1,\ offset+upp(\psi)\right], \tag{2}$$

and let $\mathrm{RRAccess}^{Q,\varphi^*}$ return I_ψ instead of $[i,i]$. For line 11, since the algorithm has already reached a leaf node of $\tilde{T}_Q$ (*i.e.* $upp(\psi)=1$), and the leaf node corresponds to no result tuple, we define the trivial interval of ψ as $I_\psi=[i,i]$ and let $\mathrm{RRAccess}^{Q,\varphi^*}$ return it.

In Example 1, when we call $\mathrm{RRAccess}^{Q_\Delta,\varphi^*}(7)$ in the enumeration process, the algorithm first computes $C_r\leftarrow children(\psi_r)$. As shown in Figure 1, $\sum_{\psi\in C_r}upp(\psi)=6<7$, thus it returns $I_{\psi_r}=\left[\sum_{\psi\in C_r}upp(\psi)+1, upp(\psi_r)\right]=[7,8]$. Then the Ban-Pick tree bans $[7,8]$ and stops picking the integers in it during the follow-up steps, thereby eliminating the need to call $\mathrm{RRAccess}^{Q_\Delta,\varphi^*}(8)$.

Notice that each trivial interval $[l,h]$ is produced by $\mathrm{RRAccess}^{Q,\varphi^*}$ at most once during the enumeration process. Otherwise, some integer $i\in[l,h]$ would be selected after $[l,h]$ has been banned, which leads to a contradiction. Moreover, by the definition of $\tilde{T}_Q$ and φ^*, the trivial intervals of the filters in $\tilde{T}_Q$ are disjoint from each other. We evaluate this basic LTI technique in Section 6.

Merging contiguous trivial intervals. We observe that many trivial intervals are contiguous and can be merged to form a larger interval. Specifically, by the definition of $\tilde{T}_Q$ and φ^*, for any filter ψ in $\tilde{T}_Q$, let $\psi'\in children(\psi)$ be the last child of ψ, then the intervals I_ψ and $I_{\psi'}$ are contiguous. For example, in $\tilde{T}_{Q_\Delta}$ (Figure 1), the last child of the root filter ψ_r is $\psi'_r=[[4,4],[1,4],[1,4]]$, then the trivial interval $I_{\psi'_r}=[6,6]$ is contiguous with $I_{\psi_r}=[7,8]$. Moreover, once we obtain a trivial interval I_ψ of a filter ψ in the recursive process of RRAccess, to compute such contiguous trivial intervals that can be merged with I_ψ, one can recursively visit the last child of the current filter, and determine whether they can be merged with their parents' trivial intervals upon the recursive return (for instance, if ψ is the last child of its parent ψ^*, then the trivial interval I_{ψ^*} can be merged with I_ψ). Then we get the merged trivial interval in $O(\log|Q|)$ time, without calling RRAccess too many times. As a result, the number of calls to RRAccess is reduced during the enumeration, leading to better practical performance. Moreover, it can be proven that during the enumeration process, no two merged

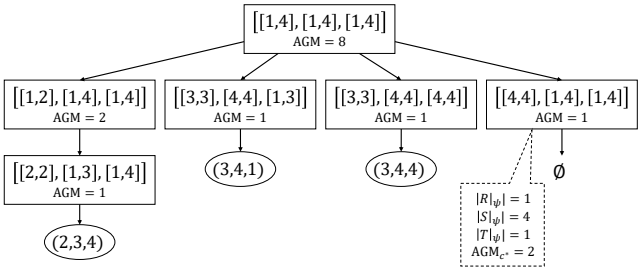


Figure 3: $\tilde{T}_{Q_\Delta}$ when $upp(\psi) = \lfloor \text{AGM}(Q_\Delta|_\psi) \rfloor$

trivial intervals overlap. We evaluate this variant of basic LTI (say merging trivial intervals, MTI) in Section 6.

Batch Trivial Interval Discovery. To further reduce the number of calls to RRAccess, we enhance the MTI technique to discover and report multiple trivial intervals within a single run of RRAccess. Specifically, along the top-to-down traversal path of the RRATree, we compute the trivial intervals of each visited node and recursively explore the chain of its last child. All the discovered trivial intervals along this path are then merged (when possible) and reported to the Ban-Pick Tree. In this way, although the enumeration delay increases to $O(\frac{\text{AGM}(Q)}{|Res(Q)|} \log^3 |Q|)$, the total running time remains $O(\text{AGM}(Q) \log |Q|)$, and the number of calls to RRAccess is reduced during the enumeration. We evaluate this variant of MTI (say batch trivial interval discovery, BTI) in Section 6.

5.2 Tighter Upper-Bound Estimation

This subsection introduces the Tighter Upper-bound estimation (TU) technique, which enhances the effectiveness of LTI. We first illustrate its impact through a motivating example.

Let $upp(\psi) = \lfloor \text{AGM}(Q|_\psi) \rfloor$ instead of $\lfloor \text{AGM}_{c^*}(Q|_\psi) \rfloor$ for any ψ, then $\tilde{T}_{Q_\Delta}$ is shown in Figure 3. For $\psi = [[4,4],[1,4],[1,4]]$, we have $|R|_\psi| = |T|_\psi| = 1$ and $|S|_\psi| = 4$. Then $upp(\psi) = \lfloor \text{AGM}(Q_\Delta|_\psi) \rfloor = 1 < \lfloor \text{AGM}_{c^*}(Q_\Delta|_\psi) \rfloor = 2$. This tighter upper bound increases the size of the trivial interval of ψ_r, yielding $I_{\psi_r} = [6,8]$ instead of $[7,8]$. Namely, TU expands the trivial intervals of filters in the lower levels of the RRATree, making more trivial integers to be discovered and banned earlier, thus enhancing the effectiveness of LTI.

In this subsection, we introduce two different upper-bound algorithms. It can be verified that both of them satisfy the properties in Definition 1 and are super-additive. In practice, for each prefix range filter ψ, we compute the minimum value among these upper-bound algorithms as the estimated upper bound of $|Res(Q|_\psi)|$, say $upp^*(\psi)$. In this way, it is clear that upp^* satisfies the properties defined in Definition 1. Moreover, it is super-additive, as formalized in the following lemma.

Lemma 10. *If the upper-bound algorithms $upp_1, \ldots, upp_c$ are super-additive, then the upper-bound algorithm upp^*, where $\forall\psi$, $upp^*(\psi) = \min_{i=1}^{c} upp_i(\psi)$, is also super-additive.*

Therefore, upp^* can be used as a replacement for the upper-bound algorithm presented in Section 4.

Upper-bound based on minimized AGM bound. Given any join query $Q \in \boldsymbol{Q}$, $|Res(Q)| \le \lfloor \text{AGM}(Q) \rfloor$. During the enumeration process, the minimal AGM bound $\text{AGM}(Q|_\psi)$ can be calculated by solving a linear program in $O(1)$ time after computing the sizes of relations $\{|R|_\psi | R \in Q\}$ in $O(\log |Q|)$ time.

However, solving this linear program incurs significant computational overhead. As a compromise, we heuristically select a small number of representative fractional edge covers and calculate the minimum of their corresponding AGM bounds. This approach effectively presents a tighter upper bound while avoiding the cost of repeatedly solving linear programs.

In Example 1, during the top-down traversal of $\tilde{T}_{Q_\Delta}$, the active domain of each attribute is progressively reduced in order x, y, and z. Since x is the first attribute to be constrained, the cardinalities of the filtered relations involving x (*i.e.*, R and T) will decrease more rapidly in each root-to-leaf path of $\tilde{T}_{Q_\Delta}$. Therefore, higher fractional edge cover weights of R and T lead to a smaller AGM bound. For instance, consider the prefix range filter $\psi = [[4,4],[1,4],[1,4]]$. Let $c \in EC(Q_\Delta)$ with $c(R) = c(T) = 1$ and $c(S) = 0$, then we have $\text{AGM}_c(Q_\Delta|_\psi) = 1 < \text{AGM}_{c^*}(Q_\Delta|_\psi) = 2$. This example demonstrates that assigning larger fractional edge cover weights to relations involving attributes that appear earlier in the domain-reduction order can lead to a tighter AGM bound at lower levels of the RRATree. In practice, we heuristically select a constant number of fractional edge covers based on this principle, and ensure that for every relation $R \in Q$, there exists at least one selected cover $c \in EC(Q)$ such that $c(R) > 0$.

Upper-bound based on acyclic skeleton query. Although the AGM bound is tight in the worst case, it can be extremely large and often much larger than the actual result size in practice. To address this, we propose another upper bound that uses additional table information to obtain a potentially tighter practical estimation. By [27], any cyclic join query can be transformed into an acyclic one by removing a subset of the relations. Specifically, given any cyclic join query Q, it can be decomposed into two subqueries, Q_s and Q_r, such that $Q = Q_s \bowtie Q_r$. Here, Q_s is an acyclic query (called the *skeleton query*) and Q_r is the query consisting of the remaining relations (called the *residual query*). In Example 1, the triangle query Q_Δ can be decomposed into an acyclic skeleton subquery $Q_{\Delta s} = R(x,y) \bowtie S(y,z)$ and a residual subquery $Q_{\Delta r} = T(x,z)$. Based on the above decomposition, the following lemma holds:

Lemma 11. *For any join query Q and filter ψ, if $\text{att}(Q_s) = \text{att}(Q)$, then $|Res(Q|_\psi)| \le |Res(Q_s|_\psi)|$. Otherwise, if $\text{att}(Q_s) \subsetneq \text{att}(Q)$, for any fractional edge cover $c \in EC(G_{Q_r \setminus Q_s})$, $|Res(Q|_\psi)| \le |Res(Q_s|_\psi)| \cdot \text{AGM}_c(Q_r^*|_\psi)$, where $Q_r^* = \{R[\text{att}(Q_r) \setminus \text{att}(Q_s)] | R \in Q_r\}$, and for any attribute set $V \subseteq \text{att}(R)$, $R[V] = \{t[V] | t \in R\}$.*

We build the indexes described in Section 4 on Q_r^* before the enumeration, so that $\text{AGM}_c(Q_r^*|_\psi)$ can be computed in $O(\log |Q|)$ time for any ψ. For $|Res(Q_s|_\psi)|$, we assume that each relation $R = \{t_1, \ldots, t_{|R|}\}$ is ordered lexicographically, and calculate an array A_R and its prefix-sum array for each $R \in Q_s$, such that $\forall 1 \le i \le |R|$, $A_R[i] = |\ltimes_{R' \in T_R} R' \ltimes t_i|$, where T_R denotes the subtree of the join tree of Q_s rooted at R. These arrays can be computed using a dynamic programming approach similar to that described in [27], which takes $O(|Q| \log |Q|)$ time. Then, for each table R and any prefix range filter ψ, the cardinality $|(\ltimes_{R' \in T_R} R') \bowtie R|_\psi|$ can be efficiently computed in $O(\log |R|)$ time with the help of a prefix-sum array of A_R. Subsequently, $|Res(Q_s|\psi)|$ can be computed in $O(1)$ time using these cardinalities for all $R \in Q$. Although this method

requires an $O(|Q| \log |Q|)$-time query-specific preprocessing, it can effectively reduce the enumeration delay for many queries in practice, especially in the early stages of the enumeration.

We implement these upper-bound algorithms, referred to as A (minimized AGM-based) and S (Skeleton-based), and evaluate their effectiveness in speeding up the enumeration in Section 6.

6 EXPERIMENTS

In this section, we present an experimental evaluation of our random-order enumeration algorithm for join queries along with the optimization techniques. We denote the basic random-order enumeration algorithm introduced in Section 3 as REnum. We evaluate the optimized variants of REnum, each combining an LTI technique from Section 5.1 with a TU technique from Section 5.2. These variants are denoted uniformly as $\text{REnum}_{\text{X-Y}}$, where $\text{X} \in \{\text{L}, \text{M}, \text{B}\}$ indicates the use of basic LIT, MTI or BTI, and $Y \in \{\text{A}, \text{AS}\}$ indicates the employed upper-bound algorithm. Unless stated otherwise, all optimized REnum variants run with full caching of the RRATree.

6.1 Experimental Setup

We now describe the setup of our experiments.

Algorithms. We compare our algorithms with sampling-based methods derived from two join sampling algorithms:

(1) **PGMJoin** [23]: the state-of-the-art among all existing implemented algorithms (with preprocessing), which requires $\Omega(|Q|)$-time preprocessing for each query.
(2) **NWCO** [13]: a theoretically nearly worst-case optimal join sampling algorithm without query-specific preprocessing.

We use the implementation of PGMJoin from their public repository. As there is no publicly available implementation of NWCO, we implemented it for our experiments. These algorithms generate uniform samples with replacement. We adapt them to a sampling without replacement (SWOR) process by naively discarding duplicate results that have already been seen. We denote the resulting variants by SWOR_{PGM} and $\text{SWOR}_{\text{NWCO}}$, respectively.

Datasets. In our experiments, we use two datasets: **TPC-DS+** and **Twitter**. TPC-DS+ is an extension of the standard TPC-DS benchmark (with a scale factor of 5), augmented with an additional customer-to-customer relation table with 3×10^5 tuples, that models follow relationships in a social network. The source node of each tuple is sampled uniformly, while its target node is drawn from a normal distribution. The resulting follow graph is treated as undirected by adding reciprocal edges. Twitter is a real-world dataset that refers to follower links and profiles of twitter users, which is also used in [10, 23, 27], and we treat the follow graph as undirected. All relation tables in these datasets are instantiated in memory, with the necessary indexes constructed in advance.

Queries. We evaluate the algorithms and the optimization techniques across three queries. On the TPC-DS+ dataset, we evaluate Q_A, which returns tuples consisting of a pair of customers and 2 items they have both purchased, under the condition that the customers are connected in the social network. On the Twitter dataset, we evaluate queries that return all triangles (Q_Δ) and 4-cliques (Q_S). To avoid excessive running times during the complete performance evaluation, we uniformly sample node subsets and run the queries on their induced subgraphs, with 2M and 1M nodes for Q_Δ and Q_S, respectively, unless stated otherwise. These queries were selected both for their practical relevance (*e.g.* in social network and e-commerce analysis); and for their structural representativeness, as they form key components of many complex cyclic join queries. The SQL statements are provided in the technical report [11].

Platform and Hardware. All experiments are performed on a workstation running 64-bit Ubuntu22.04 LTS, equipped with one Intel Xeon E7-8860@2.2GHz, 384GB DDR4 RAM, and 2TB SSD.

6.2 Experimental Results

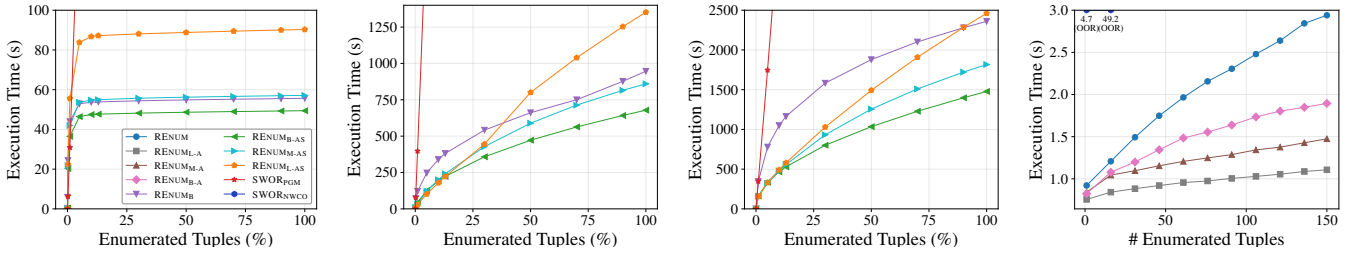
Total enumeration time. To characterize the total enumeration time of our algorithms, we compare it against that of SWOR_{PGM} and $\text{SWOR}_{\text{NWCO}}$ on these queries. We record the elapsed time from the beginning to the moment after the k-th join result is enumerated, with k represents 1% to 100% of the total result size. The experimental results are presented in Figure 4 (a)-(c) with a chart per query. As both $\text{SWOR}_{\text{NWCO}}$ and the basic REnum require prohibitively long time to produce even 1% of the join results, we only record their performance on Q_Δ in the early stages of the enumeration, which is presented in Figure 4 (d).

The results indicate that, with the speed-up techniques, our algorithm significantly outperforms SWOR_{PGM} and $\text{SWOR}_{\text{NWCO}}$. As an ablation study, Figure 4(a)–(c) show that $\text{REnum}_{\text{B-AS}}$ enumerates faster than $\text{REnum}_{\text{M-AS}}$, which in turn performs better than $\text{REnum}_{\text{L-AS}}$, and all of them significantly outperform the basic algorithm REnum. This demonstrates the effectiveness of the basic LTI, MTI and BTI. Similarly, the results of REnum_{B} and $\text{REnum}_{\text{B-AS}}$ verify the effectiveness of TU. In the early stages of the enumeration, we record the total enumeration time of the first 150 enumerated tuples, as shown in Figure 4 (d). Note that all these tuples (except those of $\text{SWOR}_{\text{NWCO}}$) are enumerated before SWOR_{PGM} completing its preprocessing and producing its first result tuple. The results show that our algorithm enumerates faster in the early stages without MTI or BTI, due to the overhead incurred by discovering and merging numerous trivial intervals.

Enumeration delay. Due to its poor performance, $\text{SWOR}_{\text{NWCO}}$ is excluded from the delay analysis, and the delay of REnum is evaluated only during the early stages (the first 150 tuples) of the enumeration on Q_Δ. We record the enumeration delay of each result tuple and analyze the delay distribution by calculating the delays for the first 5% and all results (only the first 5% for SWOR_{PGM}), which are presented in a box plot, as shown in Figure 5 (a)-(c). The delays in the early stages of Q_Δ are shown in Figure 5 (d). Outliers that fell outside the whiskers are not shown, since some are several orders of magnitude larger than the median.

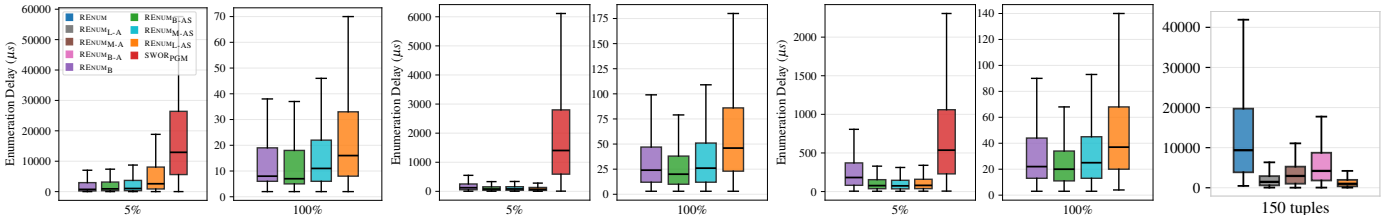
These results show that the enumeration delay of our algorithms (with the speed-up techniques) is much smaller than that of SWOR_{PGM} and $\text{SWOR}_{\text{NWCO}}$. Moreover, while MTI and BTI reduce the overall enumeration delay, they tend to increase the delay in the early stages due to the overhead of trivial interval discovery and merging. In contrast, TU mainly reduces the enumeration delay in the early stages. These findings are consistent with the experimental results on the total running time presented earlier.

Space usage. To evaluate the memory efficiency of our algorithm, we implement the techniques introduced in Section 4.6 and examine the effect of limiting the number of cached layers in the



(a) Q_A (b) Q_Δ (c) Q_S (d) Q_Δ (early stages)

Figure 4: Total enumeration time



(a) Q_A (b) Q_Δ (c) Q_S (d) Q_Δ (early stages)

Figure 5: Enumeration delay

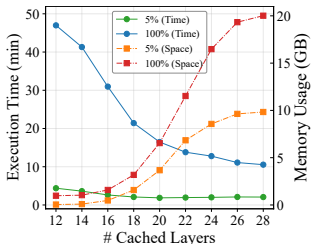


Figure 6: Time-space tradeoff

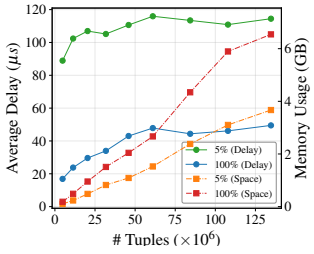


Figure 7: Scalability

RRATree. Specifically, we vary the number of the cached layers and measure both the memory usage[2] and the running time for enumerating the first 5% and all results of Q_Δ.

Figure 6 illustrates a clear trade-off between space usage and execution time. Caching only a few nodes in the upper layers (*e.g.*, up to 12–20 layers) significantly reduces memory usage while keeping the algorithm efficient, particularly in the early stages (first 5%). Notably, our algorithm outperforms sampling-based methods even under stricter memory constraints. With 12 cached layers, it enumerates 5% and 100% of $Res(Q_\Delta)$ with 46MB and 1002MB of memory, respectively. In contrast, the sampling-based methods require at least 84MB and 1690MB to store tuples in an `unordered_set<vector>` for deduplication and runs much slower. These results demonstrate that our algorithm can flexibly adapt to different memory budgets and efficiently utilizes memory to achieve high performance.

Scalability. We further evaluate the time and space scalability of our algorithm (with 20 cached layers). Specifically, we generate datasets of varying sizes by iteratively sampling nodes and extracting their induced subgraphs. On these datasets, we execute Q_Δ and measure the average delay and memory usage for enumerating the first 5% and all result tuples.

As shown in Figure 7, the average enumeration delay grows logarithmically with the database size. The space usage increases approximately linearly with the database size, yet remains far slower than the growth of the total number of results, which typically increases superlinearly with respect to the database size.

7 CONCLUSIONS

This paper demonstrates that the sampling-based random-order enumeration is not the end of the story, by presenting a more efficient random-order enumeration algorithm with a worst-case guarantee on the total running time for join queries, along with two techniques to speed up the enumeration. Our algorithm is efficient and flexible, achieving nearly optimal expected delay and total running time in the worst case under the combinatorial k-clique hypothesis, without large hidden constants or query-specific preprocessing. In our experiments, with the optimization techniques, our algorithm significantly outperforms the state-of-the-art sampling-based methods. This allows the join results to be produced more quickly in random order within data analysis pipelines. An interesting future work direction is to implement and further optimize these algorithms in external-memory and distributed database systems to support queries in large-scale data analysis.

8 ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their valuable suggestions. This work was partially supported by the Natural Science Foundation of Heilongjiang Province of China grant HSF20230095 and 2024ZXJ01A04, the National Natural Science Foundation of China grant 62372138, and the China Railway Rolling Stock Corporation.

[2]In this paper, the reported memory usage only accounts for the enumeration phase. The space required for indexing and preprocessing is not included.

REFERENCES

[1] Kaleb Alway, Eric Blais, and Semih Salihoglu. 2021. Box Covers and Domain Orderings for Beyond Worst-Case Join Processing. In *24th International Conference on Database Theory (ICDT 2021) (Leibniz International Proceedings in Informatics (LIPIcs))*, Ke Yi and Zhewei Wei (Eds.), Vol. 186. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 3:1–3:23. https://doi.org/10.4230/LIPIcs.ICDT.2021.3

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

[3] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. 2007. On acyclic conjunctive queries and constant delay enumeration. In *Proceedings of the 21st International Conference, and Proceedings of the 16th Annuall Conference on Computer Science Logic* (Lausanne, Switzerland) *(CSL'07/EACSL'07)*. Springer-Verlag, Berlin, Heidelberg, 208–222.

[4] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. 2018. Answering UCQs under Updates and in the Presence of Integrity Constraints. In *21st International Conference on Database Theory (ICDT 2018) (Leibniz International Proceedings in Informatics (LIPIcs))*, Benny Kimelfeld and Yael Amsterdamer (Eds.), Vol. 98. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 8:1–8:19. https://doi.org/10.4230/LIPIcs.ICDT.2018.8

[5] Johann Brault-Baron. 2013. *De la pertinence de l'énumération : complexité en logiques propositionnelle et du premier ordre.* Theses. Université de Caen. https://hal.science/tel-01081392

[6] Florent Capelli and Yann Strozecki. 2019. Incremental delay enumeration: Space and time. *Discrete Applied Mathematics* 268 (2019), 179–190. https://doi.org/10.1016/j.dam.2018.06.038

[7] Nofar Carmeli and Markus Kröll. 2019. On the Enumeration Complexity of Unions of Conjunctive Queries. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Amsterdam, Netherlands) *(PODS '19)*. Association for Computing Machinery, New York, NY, USA, 134–148. https://doi.org/10.1145/3294052.3319700

[8] Nofar Carmeli and Markus Kröll. 2020. Enumeration complexity of conjunctive queries with functional dependencies. *Theory of Computing Systems* 64, 5 (2020), 828–860.

[9] Nofar Carmeli, Shai Zeevi, Christoph Berkholz, Alessio Conte, Benny Kimelfeld, and Nicole Schweikardt. 2022. Answering (Unions of) Conjunctive Queries using Random Access and Random-Order Enumeration. *ACM Trans. Database Syst.* 47, 3, Article 9 (Aug. 2022), 49 pages. https://doi.org/10.1145/3531055

[10] Meeyoung Cha, Hamed Haddadi, Fabricio Benevenuto, and Krishna Gummadi. 2010. Measuring User Influence in Twitter: The Million Follower Fallacy. *Proceedings of the International AAAI Conference on Web and Social Media* 4, 1 (May 2010), 10–17. https://doi.org/10.1609/icwsm.v4i1.14033

[11] Pengyu Chen, Zizheng Guo, Jianwei Yang, and Dongjing Miao. 2025. *Towards Efficient Random-Order Enumeration for Join Queries.* Technical Report. Harbin Institute of Technology. https://github.com/Chen-Py/JoinREnum/blob/main/TechnicalReport.pdf

[12] Yu Chen and Ke Yi. 2020. Random Sampling and Size Estimation Over Cyclic Joins. In *23rd International Conference on Database Theory (ICDT 2020) (Leibniz International Proceedings in Informatics (LIPIcs))*, Carsten Lutz and Jean Christoph Jung (Eds.), Vol. 155. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 7:1–7:18. https://doi.org/10.4230/LIPIcs.ICDT.2020.7

[13] Shiyuan Deng, Shangqi Lu, and Yufei Tao. 2023. On Join Sampling and the Hardness of Combinatorial Output-Sensitive Join Algorithms. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Seattle, WA, USA) *(PODS '23)*. Association for Computing Machinery, New York, NY, USA, 99–111. https://doi.org/10.1145/3584372.3588666

[14] Mahmoud Abo Khamis, Hung Q. Ngo, Xuanlong Nguyen, Dan Olteanu, and Maximilian Schleich. 2020. Learning Models over Relational Data Using Sparse Tensors and Functional Dependencies. *ACM Trans. Database Syst.* 45, 2, Article 7 (June 2020), 66 pages. https://doi.org/10.1145/3375661

[15] Mahmoud Abo Khamis, Hung Q. Ngo, Christopher Ré, and Atri Rudra. 2016. Joins via Geometric Resolutions: Worst Case and Beyond. *ACM Trans. Database Syst.* 41, 4, Article 22 (Nov. 2016), 45 pages. https://doi.org/10.1145/2967101

[16] Kyoungmin Kim, Jaehyun Ha, George Fletcher, and Wook-Shin Han. 2023. Guaranteeing the Õ(AGM/OUT) Runtime for Uniform Sampling and Size Estimation over Joins. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Seattle, WA, USA) *(PODS '23)*. Association for Computing Machinery, New York, NY, USA, 113–125. https://doi.org/10.1145/3584372.3588676

[17] Benny Kimelfeld and Christopher Ré. 2018. A Relational Framework for Classifier Engineering. *ACM Trans. Database Syst.* 43, 3, Article 11 (Oct. 2018), 36 pages. https://doi.org/10.1145/3268931

[18] Gonzalo Navarro, Juan L. Reutter, and Javiel Rojas-Ledesma. 2020. Optimal Joins Using Compact Data Structures. In *23rd International Conference on Database Theory (ICDT 2020) (Leibniz International Proceedings in Informatics (LIPIcs))*, Carsten Lutz and Jean Christoph Jung (Eds.), Vol. 155. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 21:1–21:21. https://doi.org/10.4230/LIPIcs.ICDT.2020.21

[19] Hung Q. Ngo, Dung T. Nguyen, Christopher Re, and Atri Rudra. 2014. Beyond worst-case analysis for joins with minesweeper. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Snowbird, Utah, USA) *(PODS '14)*. Association for Computing Machinery, New York, NY, USA, 234–245. https://doi.org/10.1145/2594538.2594547

[20] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2012. Worst-case optimal join algorithms: [extended abstract]. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Scottsdale, Arizona, USA) *(PODS '12)*. Association for Computing Machinery, New York, NY, USA, 37–48. https://doi.org/10.1145/2213556.2213565

[21] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-case Optimal Join Algorithms. *J. ACM* 65, 3, Article 16 (March 2018), 40 pages. https://doi.org/10.1145/3180143

[22] Hung Q Ngo, Christopher Ré, and Atri Rudra. 2014. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Rec.* 42, 4 (Feb. 2014), 5–16. https://doi.org/10.1145/2590989.2590991

[23] Ali Mohammadi Shanghooshabad, Meghdad Kurmanji, Qingzhi Ma, Michael Shekelyan, Mehrdad Almasi, and Peter Triantafillou. 2021. PGMJoins: Random Join Sampling with Graphical Models. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1610–1622. https://doi.org/10.1145/3448016.3457302

[24] W. Nick Street and YongSeog Kim. 2001. A Streaming Ensemble Algorithm (SEA) for Large-Scale Classification. In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California) *(KDD '01)*. Association for Computing Machinery, New York, NY, USA, 377–382. https://doi.org/10.1145/502512.502568

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

[26] Ru Wang and Yufei Tao. 2024. Join Sampling Under Acyclic Degree Constraints and (Cyclic) Subgraph Sampling. In *27th International Conference on Database Theory (ICDT 2024) (Leibniz International Proceedings in Informatics (LIPIcs))*, Graham Cormode and Michael Shekelyan (Eds.), Vol. 290. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 23:1–23:20. https://doi.org/10.4230/LIPIcs.ICDT.2024.23

[27] Zhuoyue Zhao, Robert Christensen, Feifei Li, Xiao Hu, and Ke Yi. 2018. Random Sampling over Joins Revisited. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1525–1539. https://doi.org/10.1145/3183713.3183739