Beyond Equi-joins: Ranking, Enumeration and Factorization

Nikolaos Tziavelis
tziavelis.n@northeastern.edu
Northeastern University
Boston, Massachusetts, USA

Wolfgang Gatterbauer
w.gatterbauer@northeastern.edu
Northeastern University
Boston, Massachusetts, USA

Mirek Riedewald
m.riedewald@northeastern.edu
Northeastern University
Boston, Massachusetts, USA

ABSTRACT

We study full acyclic join queries with *general join predicates* that involve conjunctions and disjunctions of inequalities, focusing on *ranked enumeration* where the answers are returned incrementally in an order dictated by a given ranking function. Our approach offers strong time and space complexity guarantees in the standard RAM model of computation, getting surprisingly close to those of equi-joins. With n denoting the number of tuples in the database, we guarantee that for *every* value of k, the k top-ranked answers are returned in $O(n \operatorname{polylog} n + k \log k)$ time and $O(n \operatorname{polylog} n + k)$ space. This is within a polylogarithmic factor of the best-known guarantee for equi-joins and even $O(n + k)$, the time it takes to look at the input and output k answers. The key ingredient is an $O(n \operatorname{polylog} n)$-size *factorized* representation of the query output, which is constructed on-the-fly for a given query and database. As a side benefit, our techniques are also applicable to *unranked* enumeration (where answers can be returned in *any* order) for joins with inequalities, returning k answers in $O(n \operatorname{polylog} n + k)$. This guarantee improves over the state of the art for large values of k. In an experimental study, we show that our ranked-enumeration approach is not only theoretically interesting, but also fast and memory-efficient in practice.

1 INTRODUCTION

Join result enumeration. Join processing is one of the most fundamental topics in database research. While queries with joins have been optimized for decades, they have received renewed attention from an algorithmic perspective [51, 61, 64, 65]. Such efforts provide asymptotic guarantees that shield against *large intermediate results* and ensure predictable performance, no matter the given database instance. Similarly, work on *constant-delay enumeration* [7, 17, 44, 75] strives to pre-process the database for a given query on-the-fly so that the first answer is returned in linear time (in database size), followed by all other answers with constant (i.e., independent of database size) delay between them. (Linear pre-processing and constant delay guarantee that all answers are returned in time linear in input and output size, which is optimal.)

Ranked enumeration. Requiring join answers to be returned in a specific order gives rise to *ranked enumeration* [29, 31, 78, 79, 85, 86], where the main goal is to quickly return the most important answers *without having to materialize and sort the entire output*. Ranked enumeration generalizes the well-known *top-k* paradigm, removing the requirement of having to specify k in advance. Besides, non-trivial complexity guarantees for top-k join algorithms [47] had been derived only for the "middleware" cost model of the celebrated Threshold Algorithm [36], which only accounts for the number of distinct data items accessed [79]. In contrast, work on enumeration, including this paper, accounts for all the steps taken by the algorithm, using the standard RAM model of computation.

Beyond equi-joins. Existing work on ranked enumeration has focused on equi-joins, but big-data analytics often also requires other join conditions [30, 33, 52, 57] such as *inequalities* (e.g. `S.age < T.age`), *non-equalities* (e.g. `S.id ≠ T.id`), and *band* predicates (e.g. `|S.time - T.time| < ε`). Handling these more general predicates efficiently is challenging. If one batch-produces and sorts the full join output, then for a join of ℓ relations of size $O(n)$, sorting alone takes $O(\ell n^\ell \log n)$ time (because output size is $O(n^\ell)$). As we discuss later, a direct $O(n^2)$ reduction from theta-joins to equi-joins allows us to leverage equi-join enumeration techniques [78]. That approach delivers the top-ranked answer in $O(n^2)$ for acyclic join queries. *Our goal is to reduce this to* $O(n \operatorname{polylog} n)$ (see Fig. 1).

Example 1. *Consider an ornithologist studying interactions between bird species using a bird observation dataset* `B(Species, Family, ObsCount, Latitude, Longitude)`. *For her analysis, she decides to extract pairs of observations for birds of different species from the same larger family that have been spotted in the same region. Pairs with higher* `ObsCount` *should also appear first:*

```
SELECT   *, B1.ObsCount + B2.ObsCount as Weight
FROM     B B1, B B2
WHERE    B1.Family = B2.Family
  AND    ABS(B1.Latitude - B2.Latitude) < 1
  AND    ABS(B1.Longitude - B2.Longitude) < 1
  AND    B1.Species <> B2.Species
ORDER BY Weight DESC LIMIT 1000
```

*With n denoting the number of tuples in B, no existing approach can guarantee to return the top-*1000 *results in sub-quadratic time complexity* $o(n^2)$. *In this paper, we show how to achieve* $O(n \log^3 n)$ *even if the size of the output is* $O(n^2)$. *After returning the top-*1000 *answers, our approach is also capable of returning more answers in order without having to restart the query. The exponent of the logarithm is determined by the number of join predicates that are not equalities (3 here). Interestingly, this guarantee is not affected by the number of relations joined, e.g., if we look for triplets of bird observations, because the complexity is determined only by the* pairwise *join with the most predicates that are not equalities.*

This work. We provide the first comprehensive study on enumeration for joins with conditions that go beyond equality: (C) *general theta-join conditions*, ($C1$) *inequality*, ($C2$) *non-equality*, ($C3$) *band conditions*, as well as ($C4$) general expressions thereof as a *DNF* formula. We focus on *acyclic* join queries, which are the most common in practice. Our time and space complexity results are stated in the standard RAM model of computation and we give non-trivial guarantees for space complexity and *Time-to-kth* answer (TT(k)). Following common practice, we treat query size—intuitively, the length of the SQL string—as a constant. This corresponds to the classic notion of data complexity [81], where one is interested in

Join Condition	Example	Time $\mathcal{P}(n)$	Space $\mathcal{S}(n)$
(C) Theta	booleanUDF(S.A, T.C)	$O(n^2)$	$O(n^2)$
($C1$) Inequality	S.A < T.B	$O(n \log n)$	$O(n \log \log n)$
($C2$) Non-equality	S.A ≠ T.B		
($C3$) Band	|S.A − T.B| < ε		
($C4$) DNF of ($C1$), ($C2$), ($C3$)	(S.A<T.B ∧ S.A<T.C) ∨(S.A≠T.D)	$O(n \text{ polylog } n)$	$O(n \text{ polylog } n)$

Figure 1: Preprocessing time and space complexity of our approach for various join types. Ranked and unranked enumeration return the first k results in time $O(\mathcal{P}(n) + k \log k)$ and $O(\mathcal{P}(n) + k)$, respectively, using $O(\mathcal{S}(n) + k)$ space.

scalability in the size of the input data, not the size of the query (because users do not write arbitrarily large query expressions).

Key novelty. We propose a *Tuple-Level Factorization Graph* (TLFG) to compactly represent the results of acyclic theta-joins in general, and of joins with inequality conditions in particular. Even though a join of ℓ relations of size n can have n^ℓ answers, TLFG size is guaranteed to be within a polylogarithmic factor of input size and it is *amenable to (ranked) enumeration*. We achieve this by reducing redundancy in the join output, which is similar in motivation to factorized databases [69]. In fact, a TLFG can be directly interpreted as union-product formulas, i.e., an algebraic representation. Since work on factorized databases had mostly been concerned with equality predicates, our approach complements and extends that line of work.

Contributions. Our main contributions (see also Figure 1) are:

(1) We propose TLFG, a *succinct factorized representation* of the output of any acyclic theta-join, where the join condition can be any Boolean function over the attributes of a pair of relations. For the join of ℓ input relations with $O(n)$ tuples each, it has size and construction time $O(n^2)$, even if the output is of size n^ℓ.
(2) We propose a specialized TLFG for join conditions that are a DNF of inequalities ($C4$), achieving space and construction time $O(n \text{ polylog } n)$. For non-equalities ($C2$) and bands ($C3$), we show how to further improve upon the generic DNF guarantees. As an additional optimization, we reduce space complexity for joins with only one predicate ($C1$-$C3$).
(3) We use the TLFG for *ranked enumeration* of the results of acyclic joins. For every value of k, we return the k top-ranked results in $\text{TT}(k) = O(n^2 + k \log k)$ and $\text{TT}(k) = O(n \text{ polylog } n + k \log k)$ for general theta-joins and DNF of inequality conditions, respectively. The latter is within a polylogarithmic factor of the equi-join case where the optimal $\text{TT}(k)$ is $O(n + k \log k)$ [78]. Note that TLFG can also be used for unranked enumeration, where it achieves $\text{TT}(k) = O(n \text{ polylog } n + k)$. This yields an asymptotic improvement for large values of k over any approach that relies on indexes for range search, such as range trees [27], which give $\text{TT}(k) = O(n \text{ polylog } n + k \text{ polylog } n)$. (For sufficiently large k, e.g., $k = n^2$, the second term dominates.)
(4) Our experiments demonstrate the practical feasibility of our ranked enumeration, improving over the competition by orders of magnitude on synthetic and real datasets.

Due to space constraints, formal proofs and several details of improvements to our core techniques (Section 6) are in the appendix.

2 FORMAL SETUP

2.1 Queries

Let $[m]$ denote the set of integers $\{1, \ldots, m\}$. Instead of SQL, we use the more concise Datalog notation to express joins. A *theta-join* query is a formula of the type

$$Q(\mathbf{Z}) = R_1(\mathbf{X}_1) \wedge \ldots \wedge R_\ell(\mathbf{X}_\ell) \wedge \theta_1(\mathbf{Y}_1) \wedge \ldots \wedge \theta_q(\mathbf{Y}_q)$$

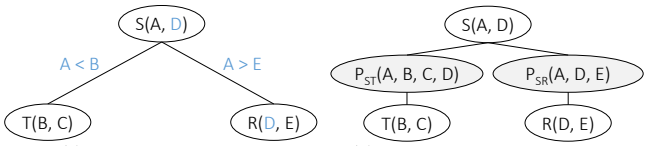
where R_i are relational symbols, $\mathbf{X}_i$ are lists of variables (or attributes), $\mathbf{Z}, \mathbf{Y}_i$ are subsets of $\bigcup \mathbf{X}_i$, $i \in [\ell]$, $j \in [q]$, and θ_j are Boolean formulas called *join predicates*. The terms $R_i(\mathbf{X}_i)$ are called the atoms of the query. Repeated occurrences of a variable in different atoms encode equality predicates. If no predicates θ_j are present, then Q is an *equi-join*. The size of the query $|Q|$ is equal to the number of symbols in the formula. We use $\text{ar}(R_i)$ to denote the arity of the relational symbol R_i and $R_i.A$ for one of its attributes A.

Query Evaluation. Join queries are evaluated over a database that consists of finite relations (or tables) R_i. These draw values from a domain which we assume to be $\mathbb{R}$ for simplicity.[1] Thus, each relation R_i is a set of tuples $r \in \mathbb{R}^{\text{ar}(R_i)}$ which assign values to the attributes of R_i. The total number of tuples in the database is n. We write $r.A$ to reference the value of a specific attribute A in tuple r. For the purposes of this paper and without loss of generality, we assume that relational symbols in different atoms are distinct since self-joins can be handled with linear overhead by copying a relation to a new one. Conceptually, the *query results* (or *answers*) are found by taking the Cartesian product between the relations, then selecting based on equi-join conditions and join predicates θ_j, and finally projecting on the attributes $\mathbf{Z}$. In this paper, we consider only *full* queries, i.e., $\mathbf{Z} = \bigcup \mathbf{X}_i$. This means that the query has no projections and returns the assigned values of all variables. Consequently, a query result can be represented as a valid combination of input tuples, one from each table R_i. While our approach can handle queries with projections, the stronger guarantees we prove only hold for full queries. Yet note that it is straightforward to extend the guarantees we prove to the class of free-connex queries [7, 10, 15] (which support certain projections), but the details are beyond the scope of this paper. We assume there are no predicates on individual relations since they can be removed in linear time by filtering the corresponding input tables.

Atomic Join Predicates. We define the following types of predicates between attributes $S.A$ and $T.B$: an *inequality* is $S.A < T.B$, $S.A > T.B$, $S.A \leq T.B$, or $S.A \geq T.B$, a *non-equality* is $S.A \neq T.B$ and a *band* is $|S.A - T.B| < \varepsilon$ for some $\varepsilon > 0$. Our approach also supports numerical expressions over input tuples, e.g., $f(S.A_1, \ldots, S.A_\alpha) < g(T.B_1, \ldots, T.B_\beta)$, with f and g arbitrary $O(1)$-time computable functions that map to $\mathbb{R}$. We say that pair $(s \in S, t \in T)$ satisfies predicate θ if $\theta(s, t) = \texttt{true}$.

Join Trees. Extending the usual definition of (alpha-)acyclicity [40, 76, 88] from equi-joins to theta-joins, we say that a join query is *acyclic* if we can construct a *join tree* with the atoms (relations) as the nodes where (1) for every attribute A, the nodes containing A form a connected subtree and (2) every join predicate θ_j is assigned to one parent-child pair of nodes (S, T) such that S and T contain

[1] Our approach naturally extends to other domains such as strings or vectors, as long as the corresponding join predicates are well-defined and computable in $O(1)$ for a pair of input tuples.



(a) Join tree of query Q. (b) Join tree of the equi-join Q_E.

Figure 2: Examples 2 and 4: The join tree of the query $Q(A,B,C,D,E) = R(D,E) \wedge S(A,D) \wedge T(B,C) \wedge (A < B) \wedge (A > E)$ before (a) and after (b) applying QuadEqui.

all the attributes referenced in θ_j. Notice that condition (1) alone is the standard definition of a join tree for equi-joins [15]. For parent S and child T we write $S \bowtie_\theta T$, where θ is a conjunction of all predicates θ_j assigned to the pair (S,T) and one equality predicate $S.A = T.A$ for every attribute A that appears in both S and T. We call θ the join condition between S and T. Since we only consider acyclic queries, the join tree is given as the input to our algorithm.

Example 2. *Consider the query $Q(A,B,C,D,E) = R(D,E) \wedge S(A,D) \wedge T(B,C) \wedge (A < B) \wedge (A > E)$.[a] This query is acyclic since we can construct a join tree (Fig. 2a). Notice that the nodes containing the same attribute D are connected and each inequality has been assigned to a parent-child pair that contains all the referenced attributes. For example, $A < B$ has been assigned to (S,T) (S contains A and T contains B). We write $S \bowtie_{\theta_1} T$ with $\theta_1 = S.A < T.B$ and $S \bowtie_{\theta_2} R$ with $\theta_2 = S.A > R.E \wedge S.D = T.D$.*

[a] SELECT * FROM R, S, T WHERE R.D = S.D AND S.A < T.B AND S.A > R.E

2.2 Enumeration

Enumeration is a process that produces the solutions to a (usually combinatorial) problem one by one without duplicates.

Unranked Enumeration. In *unranked enumeration*, a *preprocessing phase* builds data structures which then allow the *enumeration phase* to return query results. Full acyclic equi-joins admit linear preprocessing and constant delay between results [7, 10].

Ranked Enumeration. In *ranked enumeration* there is also a given *ranking function* that imposes a total order on the query answers. In general, the complexity of the problem depends on the ranking function. We focus on the case where tuple weights are combined and compared with the two operators of a *selective dioid* [38], which is a semiring with an additional ordering property that is known to be monotonic [59]. This includes ranking based on the sum of real-valued weights in ascending order and lexicographic ordering. For these ranking functions, the results of full acyclic equi-joins can be enumerated in ranked order with linear preprocessing and logarithmic delay between results [78].

2.3 Complexity Measures

We analyze all algorithms in the Random Access Machine (RAM) model with uniform cost measure in terms of their *data complexity*, i.e., we assume the query size $|Q|$ to be constant. In line with previous work [12, 20, 39], we assume that it is possible to create in linear time an index that supports tuple lookups in constant time. In practice, hashing achieves those guarantees in an expected, amortized sense. We include all index construction times and index sizes in our analysis.

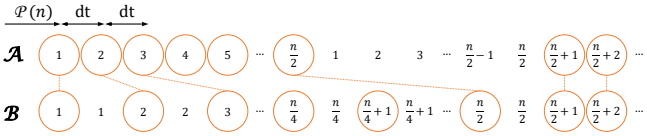


Figure 3: Enumeration algorithms with $\mathrm{TT}(k) = O(\mathcal{P}(n)+k)$.

For the time complexity of enumeration algorithms, we measure the time until the k^{th} result is returned ($\mathrm{TT}(k)$) for all values of k. For unranked enumeration, our goal is to achieve $\mathrm{TT}(k) = O(\mathcal{P}(n)+k)$ with the lowest possible preprocessing time $\mathcal{P}(n)$. The majority of papers on enumeration [7, 17, 45, 75] have traditionally focused instead on *constant delay* after $\mathcal{P}(n)$ preprocessing. This is desirable because it implies the same guarantee $\mathrm{TT}(k) = O(\mathcal{P}(n)) + k \cdot O(1) = O(\mathcal{P}(n)+k)$. However, setting constant delay as the goal can lead to misjudgments about practical performance, as we illustrate next:

Example 3. *Consider an enumeration problem where the output consists of the integers $1, 2, \ldots, n$, but algorithms produce duplicates that have to be filtered out on-the-fly. Assume that two algorithms $\mathcal{A}$ and $\mathcal{B}$ spend preprocessing $\mathcal{P}(n)$, then generate a sequence of results with constant delay. For $\mathcal{A}$, let this sequence be $1, 2, \ldots, n/2, 1, 2, \ldots, n/2, n/2+1, \ldots$ and for $\mathcal{B}$ it is $1, 1, 2, 2, \ldots, n/2, n/2, n/2+1, \ldots$ (see Fig. 3). Even though both achieve $\mathrm{TT}(k) = O(\mathcal{P}(n)+k)$, due to duplicate filtering the worst-case delay of $\mathcal{A}$ is $O(n)$ (between $n/2$ and $n/2+1$), while $\mathcal{B}$ has $O(1)$ delay. However, B is clearly* slower *than A by a factor of 2 for all $k \in [n/2]$. Since $\mathcal{A}$ outputs all these values earlier than $\mathcal{B}$, we could make $\mathcal{A}$* simulate the delay *of $\mathcal{B}$ for $k \in [n/2]$ by storing the computed values on even iterations and returning them later.*

As the example illustrates, for a preprocessing cost of $O(\mathcal{P}(n))$, the ultimate goal is to guarantee $\mathrm{TT}(k) = O(\mathcal{P}(n)+k)$. Constant-delay enumeration is a sufficient condition for achieving this goal, but not necessary. Similarly, for ranked enumeration, we aim for $\mathrm{TT}(k) = O(\mathcal{P}(n) + k \log k)$. Since we do not assume any given indexes, a trivial lower bound on $\mathcal{P}(n)$ is $O(n)$, since the input has to be read at least once. Our algorithms achieve that lower bound up to a polylogarithmic factor. For space complexity, we use $\mathrm{MEM}(k)$ to denote the required memory until the k^{th} result is returned.

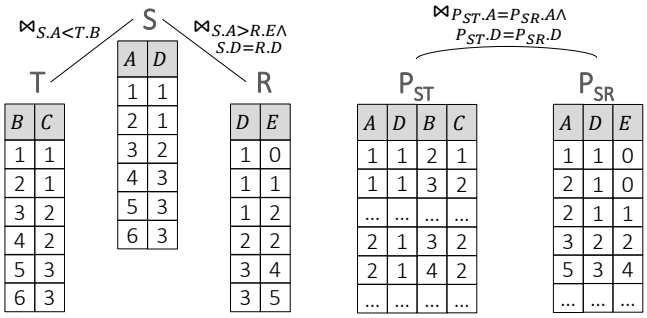
3 THETA-JOIN REPRESENTATIONS

We begin by describing a direct $O(n^2)$ reduction from a theta-join to an equi-join and then introduce a graph representation that will enable more efficient algorithms for inequality conditions.

3.1 Direct Reduction from Theta- to Equi-joins

We can transform any acyclic theta-join query into an *equivalent acyclic equi-join query* with additional "predicate relations" that encode all general predicates besides equality. For every parent-child join $S \bowtie_\theta T$ in the join tree, we materialize a new relation $P_{S,T} = \{(s,t) \mid s \in S \wedge t \in T \wedge \theta(s,t)\}$ that contains all pairs of S- and T-tuples that satisfy the join condition. Clearly, the size of $P_{S,T}$ is $O(n^2)$ and it can be computed in that time.

Theta-join $S \bowtie_\theta T$ is then replaced by the equivalent equi-join $S \bowtie P_{S,T} \bowtie T$. Intuitively, this introduces a new node between S and T in the join tree. (S and T could now even be deleted from



(a) Original database and join conditions according to the join tree of Fig. 2a.

(b) Quadratically large relations that resolve all the predicates that are not equalities.

Figure 4: Example 4: Direct transformation from a theta-join to an equi-join by the QuadEqui approach.

the join tree, but this does not affect asymptotic cost.) We proceed analogously with the remaining join-tree edges that have theta-join conditions. Once the entire join tree has been reduced to equi-joins, we can apply existing (ranked) enumeration algorithms [78]. Due to its quadratic (in input size) time and space complexity for preprocessing, we refer to this reduction algorithm as QuadEqui.

Example 4 (Example 2 continued). *Figure 2b depicts our example query Q after transforming it to an equi-join Q_E by the QuadEqui approach. There are two new "predicate relations" P_{ST} and P_{SR}, one for each child-parent pair in the original join tree. As an optimization, we remove from the query the original relations R, S, and T and connect P_{ST} directly to P_{SR}, obtaining the same query results. Figure 4b shows this direct connection, for the example database of Fig. 4. The tuples in P_{SR} are the $O(n^2)$ pairs of tuples from S and R that satisfy the join condition, i.e., they have the same D value and $S.A > R.E$. Notice that after materializing these relations, all join conditions are equalities.*

To improve over QuadEqui, we need to devise an approach where the $O(n^2)$ joining pairs between parent-child relations in the join tree are represented more compactly.

3.2 Factorization Graphs

To find an efficient representation that also admits efficient enumeration, we look back at the well-studied case of equi-joins. Specifically, we build upon our recent approach [78] that encodes the joining pairs as paths in a directed graph.

The Equality Case. An equi-join between two relations of size n can output n^2 pairs, but it can still be represented in $O(n)$ space. The key insight is that tuples with the same join value can be grouped together, associating the input tuples with the corresponding "group token". This is a fundamental insight that underlies many efficient equi-join processing techniques, from a hash partitioning to the elimination of redundancy in factorized databases [69]. In terms of the graph representation, we introduce artificial "nodes in the middle," one for each join value [78]. This creates a *shared structure*, where different source-target paths use common edges.

Example 5. *Figure 5a depicts an example constructed from relations S and T of Fig. 4a where the join condition is $S.D = T.C$. For simplicity, node labels show only the relevant join values (of attributes D, C) instead of the entire tuple. Directly connecting all the joining pairs creates $O(n^2)$ edges. A more careful construction is shown in Fig. 5b, where three intermediate nodes encode the three join values. The size of that graph is $O(n)$ but the pairs of S and T tuples connected by a path are the same as before.*

We next formalize the graph construction so that we can apply it to other join conditions:

Definition 6. *A* Tuple-Level Factorization Graph (TLFG) *of a theta-join $S \bowtie_\theta T$ of two relations S, T is a directed acyclic graph $G(V, E)$ where:*

(1) V contains a distinct node v_s for each tuple $s \in S$ and a distinct node v_t for each tuple $t \in T$ and
(2) for each $s \in S, t \in T$, there exists a path from v_s to v_t in G if and only if s and t satisfy join condition θ.

The *size* of a TLFG $G(V, E)$ is $|V| + |E|$ and its *depth* is the maximum length of any path in G.

Enumeration with TLFGs. Enumeration algorithms run directly on a TLFG representation. Unranked enumeration can be achieved with a DFS-type traversal on the graph. For example, in Fig. 5a we start from the first S tuple $s_1 = \{1, 1\}$ and in two hops (through intermediate node v_1) reach the first T tuple $t_1 = \{1, 1\}$. Since they are connected, these tuples are joinable according to the equality condition. To complete a join result, we continue in the subtree of T (according to the join tree) if more relations exist. Thereafter, the enumeration proceeds with the next T tuple $t_2 = \{2, 1\}$ that is reachable from v_1.

Ranked enumeration requires prioritizing the paths [78], naturally incurring a logarithmic factor in the traversal. Intuitively, the size of the TLFG has an impact on preprocessing (to construct it), while depth, i.e., the length of the longest path from any S tuple to any T tuple, affects the time complexity of the enumeration phase (when traversing it). As we will see in Section 6, there is often a tradeoff between the two measures. Ideally, we want to find TLFGs of *constant depth* such that the delay between results is independent of the database size.

Duplicates. Our TLFG definition does not require a one-to-one correspondence between paths and join results, i.e., there could be multiple paths between an S tuple and a T tuple it joins with. This leads to duplicate query answers in the enumeration phase. For certain join conditions, it might not be possible to find a representation that is both efficient in terms of size and depth, and also free of duplicate paths. Among the join conditions examined in this paper, this only happens for disjunctions of predicates (discussed in Section 4.3) where each answer is duplicated only $O(1)$ times, thus our complexity results are unaffected by this complication. We refer to the maximum number of paths from v_s to v_t among all $s \in S, t \in T$ pairs as the *duplication factor* of the TLFG. Ideally, the duplication factor is 1, meaning the TLFG is *duplicate-free*.

Direct TLFGs. For any theta-join, a naive way to construct a TLFG is to directly connect each source node with all its matching target nodes, as in Fig. 5c. This $O(n^2)$-size TLFG is equivalent to the QuadEqui approach in that it materializes all joining pairs. Intuitively, the edges of the TLFG are the entries of the relations introduced by QuadEqui for theta-join conditions (e.g., see Fig. 4b).

Our goal is to reduce this size by replacing the direct edges with slightly longer paths that can be composed from fewer edges.

Factorization Formulas. Typically, factorization refers to the process of compacting an algebraic formula by factoring out common sub-expressions using the distributivity property [24]. Under that perspective, factorized databases [69] represent the results of an equi-join efficiently, treating them as a formula built with product and union. Besides distributivity, d-representations [72] replace shared sub-expressions with variables, further improving succinctness through memoization [26]. Our TLFGs directly give a representation of that nature, complementing known results on join factorization. (Note that in addition to supporting joins with non-equality conditions, in TLFG the atomic unit of the formulas is a database tuple (hence Tuple-Level), while in previous work on factorized databases it is an attribute value.) We illustrate this with Example 7 below.

Example 7. *Consider the inequality join $S \bowtie_{A<B} T$ with the relations S, T of Fig. 4a. A naive TLFG is shown in Fig. 5c. The join results can be expressed with the "flat" representation:*

$$\Phi = (1 \times 2) \cup (1 \times 3) \cup \ldots \cup (1 \times 6) \cup (2 \times 3) \cup \ldots \cup (3 \times 4) \cup \ldots$$

where for convenience we refer to tuples by their A or B value, and $\times$ and $\cup$ denote Cartesian product and union respectively. The flat representation has one term for each query result, separated by the union operator. In terms of the TLFG, $\times$ corresponds to path concatenation, and $\cup$ to branching. To make the formula more compact, we can factor out tuples that appear multiple times and reuse common subexpressions by giving them a variable name. Equivalently, the size of the TLFG can be reduced if we introduce intermediate nodes, making the different paths share the same edges. Such a factorized representation is shown in Fig. 5d. We can write the corresponding algebraic formula by defining new variables $v_i, i \in [5]$ for the intermediate nodes:

$$\Phi_3 = (1 \times v_1) \cup (2 \times v_2) \cup (2 \times v_3) \cup (3 \times v_3), \ldots, (5 \times v_5)$$
$$v_1 = (2 \cup 3), v_2 = (3), v_3 = (4 \cup 5 \cup 6), \ldots, v_5 = (6).$$

We will explain how to construct this TLFG in detail in Section 4.1. Notice that the total size of these formulas is asymptotically the same as the TLFG size.

4 FACTORIZATION OF INEQUALITIES

We now show how to construct TLFGs that have $O(n \operatorname{polylog} n)$ size and $O(1)$ depth when the join condition θ in a join $S \bowtie_\theta T$ is a DNF[2] of inequalities (and also equalities). First, we consider the simple case of a single inequality and present a simple partitioning approach. Then we generalize it to conjunctions and finally, show how to combine the conjuncts of a DNF formula.

4.1 Single Inequality Condition

Equality conditions naturally group the tuples into *disjoint* equivalence classes. That is the property that allowed us to derive efficient TLFGs for equi-joins. However, it is missing for *inequality* conditions, making the problem more challenging. In this section, we develop an approach for inequalities that achieves *almost the same asymptotic guarantees* as the prior equality approach.

Baseline. A naive approach is to materialize all paths explicitly as edges. This is roughly equivalent to the QuadEqui approach which materializes all matching tuple pairs. Consider the example of Figure 5c where we have $O(n^2)$ joining tuples: each edge between a source node and a target node represents one joining pair of tuples. Our goal is to find a representation of paths that leverages the shared structure of the inequality.

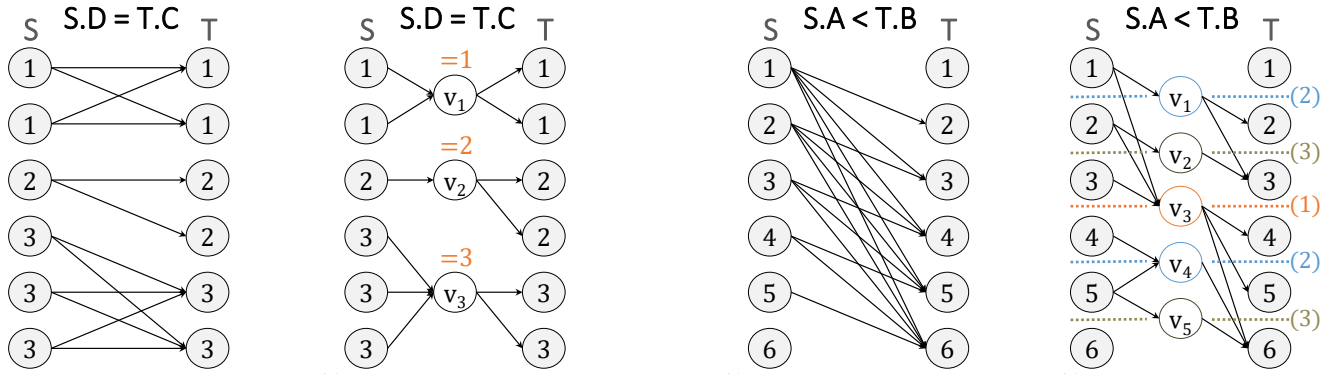
Binary partitioning. We now develop a representation that is inspired by the way quicksort partitions an array based on a pivot element [41]. We call this approach *binary partitioning*. Suppose that we have a less-than condition $S.A < T.B$. We pick a pivot value v and then partition both relations S and T s.t. $s.A < v$ for $s \in S_1$ and $s.A \geq v$ for $s \in S_2$, and similarly $t.B < v$ for $t \in T_1$ and $t.B \geq v$ for $t \in T_2$. Thereby, we know that all values in S_1 are strictly less than those in T_2. Thus, we connect them in the graph via a single intermediate node. Then, we continue on the two horizontal partitions *recursively*. Since S_2 tuples cannot join with T_1 tuples by construction, we do not miss any joining pairs. Importantly, the intermediate node we create will never be used again in subsequent recursive calls, therefore the depth of the TLFG will be 2.

In all recursive steps we pick the *median of the distinct values* as our pivot. For multiset $\{1, 1, 1, 1, 2, 3, 3\}$ the set of distinct values is $\{1, 2, 3\}$ and hence the median is 2. This pivot is easy to find in $O(n)$ if the relations have been sorted on the appropriate attributes beforehand. If all tuples contain different values, then partitioning with the median creates two roughly even partitions of sizes $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$. Thus, each recursive step cuts the input by half and with $O(\log n)$ recursive steps we reach the base case of just one input tuple. However, if the same attribute value appears in multiple input tuples, the two partitions we create might be *uneven*. Still, *the number of distinct values d drops by half* in each recursive call. The number of steps needed to reach the base case of a single distinct value ($d = 1$) is then $O(\log d) = O(\log n)$ because $d \leq n$. When that happens with a strictly less-than inequality ($<$), we stop because all the tuples share the same value. Overall, the time and size of this approach is $O(n \log n)$, and the depth is 2.

Example 8. *Figure 5d illustrates the approach for the same example as before, with dotted lines showing how the relations are partitioned. Initially, we create partitions $\{1, 2, 3\}$ and $\{4, 5, 6\}$. The source nodes of the first partition are connected to the target nodes of the second partition via the intermediate node v_3. The first partition is then recursively split into $\{1\}$ and $\{2, 3\}$. Even though these new partitions are uneven with 2 and 4 nodes respectively, they contain roughly the same number of distinct values (plus or minus one).*

Other inequality types. Our approach for less-than ($<$) is straightforward to generalize to greater-than ($>$), since it is exactly symmetrical: We simply connect the partitions in the opposite direction, i.e., S_2 connects to T_1 (instead of S_1 to T_2). For inequality predicates with equality ($\leq, \geq$), only a minor change in the base case of the algorithm is needed: Instead of simply returning from the recursive call when only 1 distinct value remains, we connect all the source-target nodes that contain that (equal) value. This modification does not affect any of our guarantees.

[2] Converting an arbitrary formula to DNF may increase query size exponentially. This does not affect data complexity, because query size is still a constant.



(a) Equality: naive construction with edges between all joining pairs. $O(n^2)$ size.

(b) Equality: grouping tuples with common join values together. $O(n)$ size.

(c) Inequality: naive construction with edges between all joining pairs. $O(n^2)$ size.

(d) Inequality: binary partitioning. Dotted lines indicate partitioning steps. $O(n \log n)$ size.

Figure 5: Factorization of Equality and Inequality conditions with our TLFGs. The S and T node labels indicate the values of the joining attributes. All TLFGs shown here have $O(1)$ depth.

Lemma 9. *Let θ be an inequality predicate for relations S, T of total size n. A duplicate-free TLFG of $S \bowtie_\theta T$ of size $O(n \log n)$ and depth 2 can be constructed in $O(n \log n)$ time.*

Proof. Correctness is easy to establish by induction: each recursive step connects precisely the joining pairs between the two partitions and the graph within each partition is correct inductively. For the running time, we begin by sorting the relations in $O(n \log n)$. We analyze the recursion in terms of its recursion tree. Each recursive step with size $|S| + |T| = n$ requires $O(n)$ to partition the sorted relations. Then, we materialize one intermediate node and for each source and target node at most one edge. We then invoke 2 recursive calls with sizes $n_1 + n_2 = n$. Therefore, in every level of the recursion tree, the sizes of all the subproblems add up to n. Since we spend linear time per recursive step, the total work per level of the recursion tree is $O(n)$. We always cut the distinct values (roughly) in half, thus the height δ of the tree is $O(\log d) = O(\log n)$. Overall, the time spent on the recursion is $O(n\delta) = O(n \log n)$, which also bounds the size of the TLFG. Across all recursive steps, edges are created either from source nodes to intermediate nodes or from intermediate nodes to target nodes. Thus, the length of all paths from source to target nodes is 2. The invariant property which ensures that the TLFG is duplicate-free is that whenever a recursive step is called on a set of S', T' tuples, no path exists between $v_{s'}$ and $v_{t'}$ for $s' \in S', t' \in T'$. □

4.2 Conjunctions

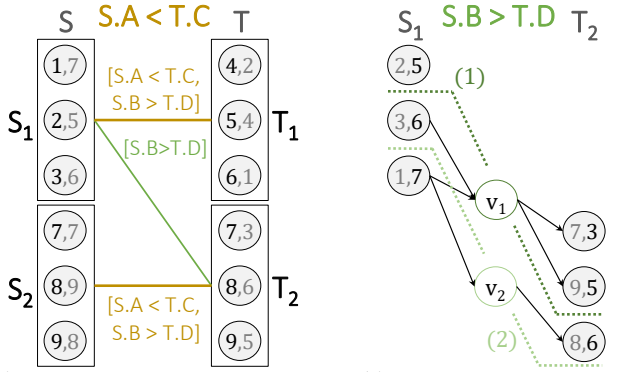
For a conjunction of predicates, we have to make sure that each path in the TLFG satisfies *all predicates*. If we were to construct a TLFG for each predicate individually, then it would be hard to combine the graphs into a single one that has that property. Instead, we propose an approach that handles the conjunction by considering the predicates in sequence: Whenever a set of source nodes would be connected to a set of target nodes according to one predicate, we demand that they additionally satisfy the remaining predicates. Thus, each predicate acts as a filter that keeps only certain pairs of source-target nodes and passes them on to the next predicate. When no predicates remain, then we simply connect the two sets.

Equalities. First, we show that all equality predicates in the conjunction can be treated with virtually no overhead. The property that we rely on is that equalities (irrespective of their number) create disjoint partitions of tuples (see Fig. 5b). Since these partitions are independent in the graph, we simply create a TLFG for each partition separately for all remaining predicates.

Example 10. *Suppose $\theta \equiv (S.D = T.C) \wedge (S.A < T.B)$. We first process the equality predicate and then the inequality. For the example of Fig. 5b, the equality creates three disjoint partitions: (S_1, T_1) with value 1, (S_2, T_2) with value 2, and (S_3, T_3) with value 3. Rather than connecting the source-target nodes within each partition via an intermediate node as in the case where we only have equalities, we now have three inequality subproblems, one for each partition: For (S_i, T_i), we construct a TLFG with the algorithm of Section 4.1. Source-target nodes connected by the latter will satisfy both predicates since they belong to the same equality partition.*

Lemma 11. *Let θ be a conjunction of predicates between relations S, T of total size n, and θ' be that conjunction with all the equality predicates removed. If for S', T' with $|S'|+|T'| = n'$ we can construct a TLFG of the join $S' \bowtie_{\theta'} T'$ of size $O(f(n'))$, depth λ, and duplication factor u in time $O(g(n'))$, and f, g are superadditive functions, then we can construct a TLFG of the join $S \bowtie_\theta T$ of size $O(f(n))$, depth λ, and duplication factor u in time $O(g(n) + n)$.*

Inequalities. We generalize the same idea to the case of multiple inequality predicates. To handle each inequality, we use the binary partitioning approach we developed in Section 4.1. Algorithm 1 shows the pseudocode of our approach. The two partitions are connected via an intermediate node only when no other predicates remain (Lines 15 to 17), otherwise they are passed on to the next predicate (Line 20). Overall, we perform two recursions simultaneously. In one direction, we make recursive calls on smaller partitions of the data and the same set of predicates (Lines 22 and 23). In the other direction, when the current predicate is satisfied for some source-target nodes, nextPredicate() is called with one less predicate (Line 20). The recursion stops either when we are left with 1 join value (base case for binary partitioning) or we exhaust the predicate list (base case for conjunction). Finally, notice that each



(a) Binary partitioning and recursions. (b) Handling the next predicate.

Figure 6: Example 12: Steps of the conjunction algorithm for two inequality predicates on $S(A, B), T(C, D)$. Node labels depict A, B values (left) or C, D values (right).

Algorithm 1: Factorizing a conjunction of inequalities

```
1  Input: Relations S, T, nodes v_s, v_t for s ∈ S, t ∈ T,
2         Conjunction θ as list of conditions [θ_1 | θ_L]
3  Output: A TLFG of the join S ⋈_θ T
4  nextPredicate (S, T, θ)
5  Procedure nextPredicate(S, T, θ = [θ_1 | θ_L])
6      S', T' = S, T sorted by the attributes of θ_1
7      if (θ_1 == S.A < T.B) then
8          partIneqBinary (S', T', [θ_1 | θ_L])
9  Procedure partIneqBinary(S, T, [θ_1 = S.A < T.B | θ_L])
10     d = vals(S ∪ T)//Number of distinct A, B values
11     if d == 1 then return //Base case for binary partitioning
12     Partition (S ∪ T) into (S_1 ∪ T_1), (S_2 ∪ T_2) with median distinct
         value as pivot
13     if θ_L == [ ] then
14         //Base case for #predicates: connect S_1 to T_2
15         Materialize intermediate node x
16         foreach s in S_1 do Create edge v_s ⟶ x
17         foreach t in T_2 do Create edge x ⟶ v_t
18     else
19         //Check S_1 ⟶ T_2 against the rest of the predicates
20         nextPredicate (S_1, T_2, θ_L)
21     //Recursive calls on horizontal partitions, same predicates
22     partIneqBinary (S_1, T_1, [θ_1 | θ_L])
23     partIneqBinary (S_2, T_2, [θ_1 | θ_L])
```

time a new predicate is encountered, the nodes have to be sorted according to the new attributes (Line 6).

Example 12. *Consider two inequalities $S.A < T.C \wedge S.B > T.D$ for relations $S(A, B), T(C, D)$ as shown in Fig. 6a. The algorithm initially processes the first inequality and splits the relations into $(S_1, T_1), (S_2, T_2)$ as per the binary partitioning method (see Section 4.1). The recursive calls on these two partitions (depicted with horizontal edges) are made with the same list of predicates. While for one inequality S_1 and T_2 would be connected via intermediate node, we now make a third recursive call (depicted with a diagonal edge) that will process the next inequality $S.B > T.D$. The result of this recursive call is shown in Fig. 6b. Only some pairs of these nodes satisfy this second predicate and are eventually connected. Also notice that inside this recursive call, we had to sort on attributes B, D before using binary partitioning.*

Lemma 13. *Let θ be a conjunction of p inequality and any number of equality predicates for relations S, T of total size n. A duplicate-free TLFG of $S \bowtie_\theta T$ of size $O(n \log^p n)$ and depth 2 can be constructed in $O(n \log^p n)$ time.*

4.3 Disjunctions

We now consider disjunctions so that we can combine the TLFGs constructed from each of the conjunctions of a DNF formula. As long as we can construct a TLFG for each term of the disjunction, we can then put them together so that the final TLFG contains the union of the TLFG paths. This approach may create duplicate paths since the predicates in the disjunction may be satisfied by the same pairs of tuples. However, the number of these duplicates is bounded by the number of different TLFGs we assemble, which in turn depends only on the size of the query.

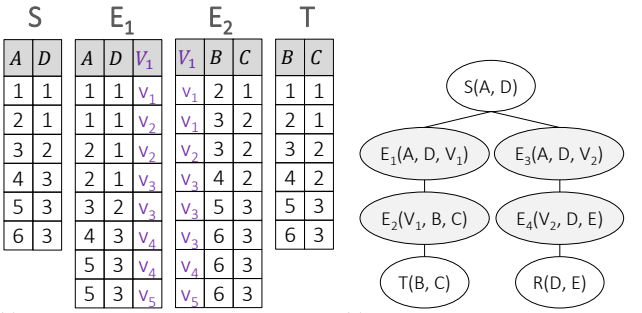
Lemma 14. *Let θ be a disjunction of predicates $\theta_1, \ldots, \theta_p$ for relations S, T. If for each $\theta_i, i \in [p]$ we can construct a duplicate-free TLFG of $S \bowtie_{\theta_i} T$ of size $O(\mathcal{S}_i)$ and depth λ_i in $O(\mathcal{T}_i)$ time, then we can construct a TLFG of $S \bowtie_\theta T$ of size $O(\sum_i \mathcal{S}_i)$ and depth $\max_i \lambda_i$ in $O(\sum_i O(\mathcal{T}_i))$ time. The duplication factor of the latter is at most p.*

5 ENUMERATION FOR ACYCLIC QUERIES

We now apply our techniques on enumeration problems for acylic queries that contain inequalities. A baseline approach is to apply QuadEqui to transform the query into an equi-join (see Section 2.1). This allows us to then use any enumeration algorithm designed for equi-joins. However, QuadEqui materializes relations that are quadratically large, hence the preprocessing time (and memory) for ranked or unranked enumeration will be $\mathcal{P}(n) = O(n^2)$. Instead, we leverage our factorizations to derive enumeration algorithms whose preprocessing time is only $\mathcal{P}(n) = O(n \text{ polylog } n)$.

The idea is to represent the matching tuples *for each parent-child pair* of relations S, T in the join tree with our TLFGs. Consequently, matching tuples across the entire tree are connected and each matching combination corresponds to one query result. As discussed in Section 3.2, enumeration algorithms operate directly on that structure, e.g., with the any-k Tree-DP framework we have proposed [78]. Since the details of that approach are beyond the limits of this paper, we instead present our results here via an alternative route. From the factorized graph representation we will create a relational representation by transforming the graph into tables. Thus, the inequality-join is transformed into an *equi-join* over $O(n \text{ polylog } n)$ relations. Then, we apply equi-join enumeration to the new query as a black-box procedure. This approach is similar to QuadEqui (or the equivalent direct TLFG) but more efficient, since the materialized relations are asymptotically smaller. This highlights the generality of our approach for contexts besides enumeration: *it can be used to reduce an acyclic query with general join conditions to an equi-join on relations that are only $O(\text{polylog } n)$ larger.*

Example 15 (Example 4 continued). *For the join of Fig. 4a, we construct a TLFG for $S - T$ and $S - R$ respectively. The former is depicted in Fig. 5d. We transform the TLFG into a relational representation by adding new domain values for the intermediate nodes. More specifically, we introduce a new attribute V_1 with domain values $v_1 - v_5$ and store the edges from source nodes to*



(a) Relations constructed from the TLFG of Fig. 5d.

(b) The join tree of the equi-join Q_F that we obtain with our method.

Figure 7: Example 15: Using our TLFGs to transform the inequality join of Fig. 2a into an equi-join.

V_1 nodes in a relation E_1, shown in Fig. 7a. Similarly, a relation E_2 contains the edges from V_1 nodes to target nodes. The relations are now joined with equi-join conditions only via the new attribute V_1. After repeating the process for relations S, R, we get a new join tree shown in Fig. 7b that corresponds to the equi-join query $Q_F(A, B, C, D, E, V_1, V_2) = R(D, E) \wedge S(A, D) \wedge T(B, C) \wedge E_1(A, D, V_1) \wedge E_2(V_1, B, C) \wedge E_3(A, D, V_2) \wedge E_4(V_2, D, E)$. Notice that the size of the new relations E_1, E_2, E_3, E_4 is only $O(n \log n)$ by the binary partitioning method.

By applying this transformation and using known techniques for the enumeration of equi-joins, we establish the following:

Theorem 16. *Let Q be a full acyclic theta-join query over a database D of size n where all the join conditions are DNF formulas of equality and inequality predicates. Let p be the maximum number of inequalities in a conjunction in every join condition θ of the join tree. Ranked enumeration of the answers to Q over D can be performed with $\mathrm{TT}(k) = O(n \log^p n + k \log k)$, while unranked enumeration can be performed with $\mathrm{TT}(k) = O(n \log^p n + k)$. The space requirement in both cases is $\mathrm{MEM}(k) = O(n \log^p n + k)$.*

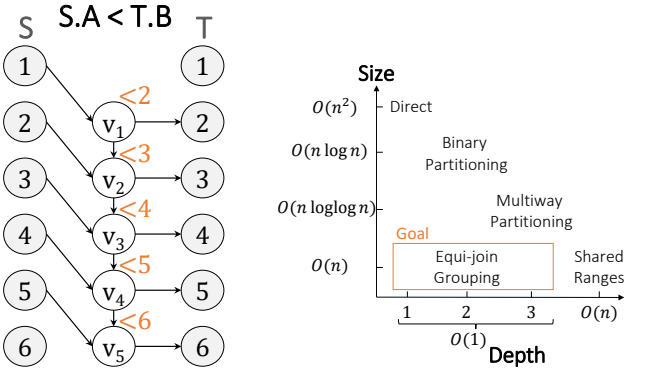
6 FURTHER IMPROVEMENTS

We now work on improving the efficiency of our approach. In particular, we study alternative factorization methods (beyond binary partitioning) and then specialize our treatment of common inequality-type predicates, namely non-equalities and bands. These allow us to then strengthen the guarantees of Theorem 16. Due to space constraints, we only present the high-level ideas, but all the details can be found in the full version [80].

6.1 Factorization Methods

In this section, we explore different factorization methods for inequalities to asymptotically improve the size of the TLFG.

Shared ranges. A different idea than partitioning that is based on *shared ranges* is depicted in Figure 8a. This method creates a hierarchy of intermediate nodes, each one representing a range of values. Each range is entirely contained in all those that are higher in the hierarchy, thus we connect the intermediate nodes in a chain. Even though this approach yields $O(n)$ size for the TLFG, the depth is unfortunately $O(n)$ (e.g. see the path from left node 1 to right node 6). This is particularly undesirable for enumeration algorithms



(a) Inequality: shared ranges. Middle nodes indicate a range. $O(n)$ size, $O(n)$ depth.

(b) Tradeoff between size and depth of the TLFG for a single inequality. Ideally, we want to achieve $O(n)$ size and $O(1)$ depth, which is possible for equi-joins.

Figure 8: Different factorization methods.

that traverse the TLFG since the depth affects the enumeration delay. To achieve guarantees of the type $\mathrm{TT}(k) = O(\mathcal{P}(n) + k \log k)$ (see Section 2.3), we want instead TLFGs of $O(1)$ depth. Moreover, it is unclear how to generalize this idea to a conjunction of inequalities.

Multiway partitioning. When only one inequality predicate is present, it is possible to improve the partitioning method so that the size of the representation becomes $O(n \log \log n)$ instead of $O(n \log n)$. The idea is to split the tuples into multiple partitions per step instead of two, hence we call this improved approach *multiway partitioning*. Intuitively, it does more work in each recursive step (but still linear) so that the total number of steps decreases. One complication of this approach is that we need 2 layers of intermediate nodes to appropriately connect the partitions, hence the depth is 3 (instead of 2 of the binary partitioning method). This tradeoff between size and depth of the TLFG is depicted in Fig. 8b. We also emphasize that this improvement is only possible for the special case of one predicate, while for conjunctions of inequalities we still rely on binary partitioning.

6.2 Non-Equality and Band Predicates

We now consider the case of non-equality and band predicates in the join condition. These predicates can be translated into a DNF of inequalities that we already handle with our techniques: A non-equality is a disjunction and a band is a conjunction of 2 inequalities. However, a specialized construction that leverages the specific structure of these predicates gives a more efficient representation. Specifically, we show that these two predicates can be handled as efficiently as an inequality predicate.

6.3 Putting Everything Together

Combining the multiway partitioning method with the predicate-specific techniques gives us the following result:

Lemma 17. *Let θ be an inequality, non-equality, or band predicate for relations S, T of total size n. A duplicate-free TLFG of $S \bowtie_\theta T$ of size $O(n \log \log n)$ and depth 3 can be constructed in $O(n \log n)$ time.*

We are now in position to state our main result. Compared to Theorem 16, the theorem below relies on multiway partitioning

for the base case of the conjunction algorithm (when one predicate remains) and the specialized TLFGs for non-equalities and bands.

Theorem 18 (Main Result). *Let Q be a full acyclic theta-join query over a database D of size n where all the join conditions are DNF formulas of equality, inequality, non-equality, or band predicates. Let p be the maximum number of predicates excluding equalities in a conjunction in every join condition θ of the join tree. Ranked enumeration of the answers to Q over D can be performed with* $\mathrm{TT}(k) = O(n \log^p n + k \log k)$, *while unranked enumeration can be performed with* $\mathrm{TT}(k) = O(n \log^p n + k)$. *The space requirement in both cases is* $\mathrm{MEM}(k) = O(n \log^{p-1} n \cdot \log\log n + k)$.

7 EXPERIMENTS

We now validate the effectiveness of our approach for ranked enumeration in practice. In particular, we are interested in the time taken by various approaches to return the k^{th} ranked result for different queries and data.

Algorithms. We compare 4 approaches for ranked enumeration of join queries with inequality-type predicates: ① Factorized Any-K is our factorization approach. ② QuadEqui is the approach described in Section 3.1 which materializes quadratically large relations that resolve the inequality predicates. Given the many possible ways to handle the inequalities, we elect to compare against an *idealistic implementation* that cannot be surpassed by any possible real-world implementation. In our experiments, QuadEqui receives its materialized tables without measuring the time required for their computation. We thus obtain a *lower bound* on the running time. ③ Batch produces the entire output of the query and then sorts it. It serves as a yardstick for any approach that materializes all results. Batch is given access to all the query results without having to compute them; we only measure the time it takes to sort and thus again obtain a *lower bound* on the running time. We note that for $\ell = 2$ relations, the difference between QuadEqui and Batch is that the former uses a priority queue. ④ PSQL uses the open-source PostgreSQL system. Even though it is developed in a different language and offers many other functionalities, it still provides a point of reference to the performance of existing systems.

We also compare our different factorization methods against each other (see Section 6.1). (1a) Binary Partitioning is the most general and works for all join conditions we consider. (1b) Multiway Partitioning is our asymptotic optimization for a single inequality-type predicate per pairwise join. (1c) Shared Ranges is based on the idea of transitivity, which lacks in terms of TLFG depth (hence enumeration delay). and can – to the best of our knowledge – only be applied to a single inequality-type predicate.

Data. Ⓢ Our synthetic data generator creates relations $S_i(A_i, A_{i+1}, W)$, $i \geq 1$ by drawing values for A_i from a fixed domain of integers $[0 \ldots 10^4 - 1]$ uniformly at random while discarding duplicates. The weights W are reals drawn from $[0, 10^4)$.

Ⓡ For real data, we use a temporal graph RedditTitles [55] where the $\sim 286k$ edges are posts from a source community to a target community identified by a hyperlink in the post title. The schema is `Reddit(From, To, Timestamp, Sentiment, Readability)`. Ⓑ We also use OceaniaBirds [1], a dataset of bird observations from the Oceania continent. The schema is `Birds(ID, Latitude, Longitude, IndivCount)`. We keep only the $\sim 452k$ observations that have a non-empty `IndivCount` attribute.

Queries. We test queries with various join conditions and sizes. For each type of query, we give below the corresponding SQL query that would produce the entire sorted result for a binary join. We parameterize the queries with the number of relations ℓ. For longer queries $\ell > 2$, we organize the relations in a chain and repeat the given join conditions between the i^{th} and $(i+1)^{\text{th}}$ relations.

Query Q_{S1} joins our synthetic tables with a single inequality.

```
SELECT   *, S1.W + S2.W as Weight
FROM     S1, S2
WHERE    S1.A2 < S2.A3
ORDER BY Weight ASC                                  Q_S1
```

Query Q_{S2} has a more complicated join condition that is a conjunction of a band and a non-equality.

```
SELECT   *, S1.W + S2.W as Weight
FROM     S1, S2
WHERE    ABS(S1.A2 - S2.A3) < 50 AND S1.A1 <> S2.A4
ORDER BY Weight ASC                                  Q_S2
```

Query Q_{R1} computes temporal paths [83] on RedditTitles, and ranks them by a measure of sentiment such that sequences of negative posts are retrieved first.

```
SELECT   *, R1.Sentiment + R2.Sentiment as Weight
FROM     Reddit R1, Reddit R2
WHERE    R2.Timestamp > R1.Timestamp
ORDER BY Weight ASC                                  Q_R1
```

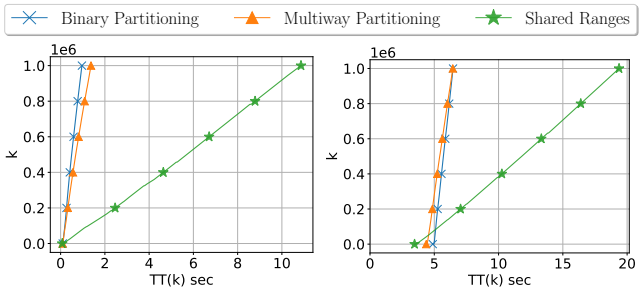
Query Q_{R2} uses the sentiment in the join condition, keeping only paths along which the negative sentiment increases. For ranking, we use a measure of readability to focus on posts of higher quality.

```
SELECT   *, R1.Readability + R2.Readability as Weight
FROM     Reddit R1, Reddit R2
WHERE    R2.Timestamp > R1.Timestamp AND
         R2.Sentiment < R1.Sentiment
ORDER BY Weight DESC                                 Q_R2
```

Last, Q_B is a spatial band join on OceaniaBirds that finds pairs of populous bird sightings that are close based on proximity. The parameter ε is given as a parameter.

```
SELECT   *, B1.IndivCount + R2.IndivCount as Weight
FROM     Birds B1, Birds B2
WHERE    ABS(B2.Latitude - B1.Latitude) < ε AND
         ABS(B2.Longitude - B1.Longitude) < ε
ORDER BY Weight DESC                                 Q_B
```

Details. The implementation of our algorithms is in Java 8 and the experiments are conducted on an Intel Xeon E5-2643 CPU running Ubuntu Linux. The query execution is in main memory, and the Java VM is allocated 100GB of RAM. If that is exceeded, we indicate it with an Out-Of-Memory (OOM) annotation. For Factorized Any-K and QuadEqui that require an equi-join ranked enumeration algorithm, we use any-k Lazy [21, 78] which was found to outperform others in previous work. The version of PostgreSQL is 9.5.24. We set its parameters such that it is optimized for main-memory execution and system overhead related to logging or concurrency is minimized, as it is standard in the literature [9, 78]. To enable input caching for PSQL, each execution is performed twice and we only



(a) Query Q_{S1}, $\ell = 4$, $n = 2^{10}$. (b) Query Q_{R1}, $\ell = 4$.

Figure 9: First 10^6 results for our different factorization methods on queries with one single inequality-type predicate. (Recall that only Binary Partitioning generalizes to arbitrary inequality conjunctions).

	Q_{S1}, $n = 2^{10}$	Q_{S1}, $n = 2^{20}$	Q_{R1}
Multiway Partitioning	35.433k	27.007M	9.455M
Binary Partitioning	38.967k	47.307M	10.255M
Shared Ranges	14.306k	11.593M	4.868M

Figure 10: Representation size measured as the sum of nodes and edges of the TLFG.

time the second one. Additionally, we create appropriate indexes on the input relations beforehand, while our methods do not receive these indexes. Even though the task is ranked enumeration, we still give PSQL a LIMIT clause whenever we measure a specific TT(k), and thus allow it to leverage the k value. All data points we show are the median of 5 measurements.

7.1 Comparison of our 3 Methods

First, we compare the performance of ranked enumeration using the three different factorization methods we proposed. Since only Binary Partitioning is applicable to all the types of join conditions considered in this paper, we test the different methods on the queries that have only one inequality-type predicate (Q_{S1}, Q_{R1}). Figure 9 depicts TT(k) for the first 10^6 results. Shared Ranges yields an enumeration delay that is linear in the database size, hence quickly deteriorates as the number of returned results k increases. However, its TLFG is constructed in a single pass (after sorting), which makes preprocessing slightly faster than the partitioning approaches. Specifically, for Q_{R1} (Fig. 9b) it starts returning results after 3.5 sec, compared to 4.4 of Multiway Partitioning and 4.9 of Binary Partitioning. Still, we note that this is a constant-factor advantage since the asymptotic cost is dominated by sorting for all three. Multiway Partitioning is faster than Binary Partitioning in preprocessing but slower in terms of delay, hence the latter catches up and eventually overtakes it as k increases. This is a consequence of the size-depth tradeoff of the TLFGs (Fig. 8b).

We also report the size of the constructed TLFGs (Fig. 10). In line with our analysis, Shared Ranges is significantly more compact (by a factor of 2.1 up to 4.1) than Binary Partitioning across all tested queries. Multiway Partitioning also succeeds to a lesser extent in reducing the size of Binary Partitioning (by a factor of 1.1 up to 1.8). Because of the asymptotic differences, the factors of improvement are expected to increase for larger databases.

7.2 Comparison Against Alternatives

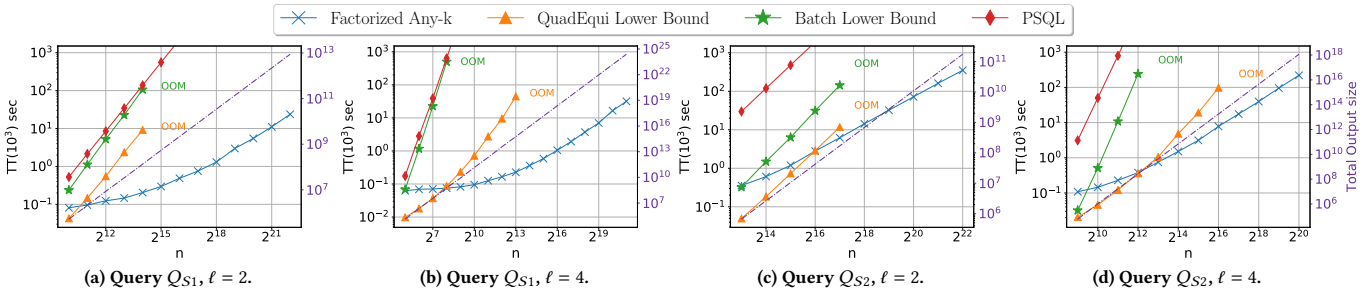
In the following, we set (1b) Multiway Partitioning as the factorization method of (1) Factorized Any-K for the single-predicate cases and (1a) Binary Partitioning for all others. We will show that our approach has a significant advantage over the competition when the size of the output is sufficiently large. We test three distinct scenarios for which large output can occur: (1) the size of the database grows, (2) the length of the query increases, and (3) the parameter of a band join increases.

Summary. *(1) Factorized Any-K is superior when the total output size is large, even when compared against a lower bound of the running time of the other methods. (2) QuadEqui and (3) Batch require significantly more memory and are infeasible for a variety of queries. (4) PSQL does not support ranked enumeration and produces the entire output which is very costly.*

7.2.1 Effect of Data Size. We run queries Q_{S1}, Q_{S2} for different input sizes n and two distinct query lengths.[3] Figure 11 depicts the time to return the top $k = 10^3$ results. The value of k is chosen to achieve a balance between returning a few top results and the special case of a top-1 computation. We also plot how the size of the output grows with increasing n. Even though QuadEqui and Batch are given precomputed results and do not even have to resolve complicated join predicates, they still require a large amount of memory to store those. Thus, they quickly run out of memory even for relatively small inputs (Figure 11a). PSQL does not face a memory problem because it can resort to secondary storage, yet becomes unacceptably slow. In contrast, our Factorized Any-K approach scales smoothly across all tests and requires much less memory. For instance, in Figure 11b QuadEqui fails after $\sim 8k$ input size, while we can easily handle $\sim 2M$ and even larger than that. For very small input sizes, the lower bounds of QuadEqui and Batch are sometimes lower, but their real running times can be much higher than that. Q_{S2} has more join predicates and thus, a more restricted output size (Figures 11c and 11d). Our advantage is smaller in this case, yet still significant for large values of n.

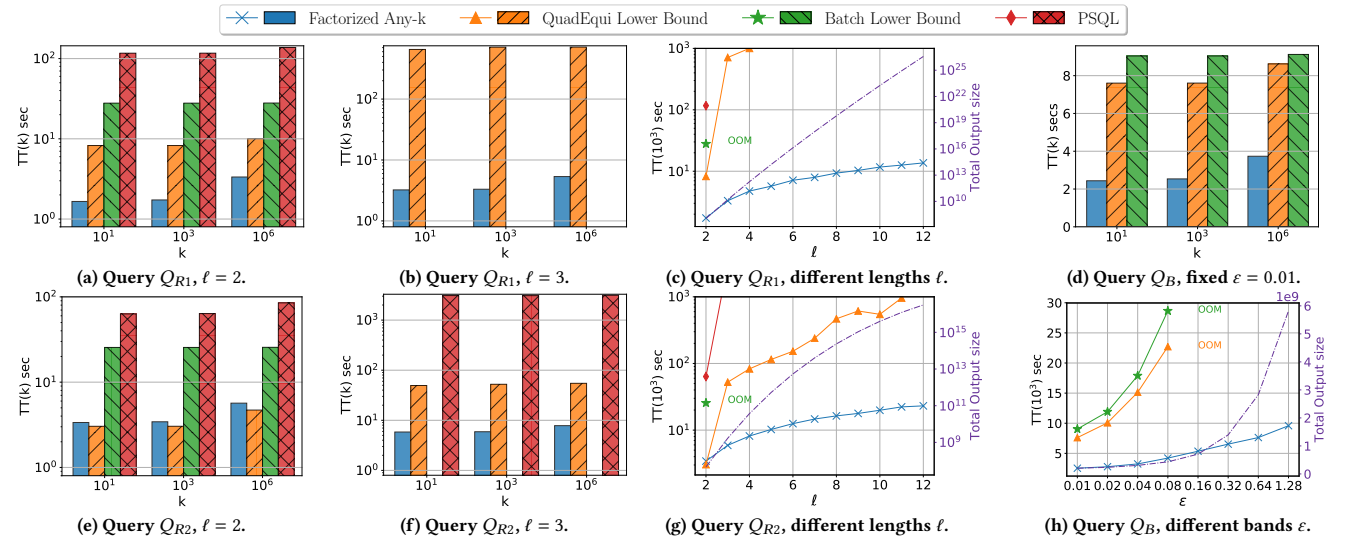
7.2.2 Effect of Query Length. Next, we test the effect of query length on RedditTitles. We plot TT(k) for three distinct values ($k = 10, 10^3, 10^6$) when the length is small ($\ell = 2, 3$) and one value ($k = 10^3$) for longer queries. Figure 12 depicts our results for queries Q_{R1}, Q_{R2}. Increasing the value of k makes the enumeration phase longer, but the relative standing of the approaches is not affected. For a binary-join Q_{R1}, our Factorized Any-K is faster than the lower bounds of the other methods (Figure 12a), and its advantage increases for longer queries, since the output also grows (Figures 12b and 12c). Batch runs out of memory for $\ell = 3$, while PSQL did not terminate after 3 hours. Query Q_{R2} has an additional join predicate, hence its output size is more restricted. Thus, the lower bound of QuadEqui is slightly better than our approach for $\ell = 2$ (Figure 12e), but we expect that this would not happen when the cost of materializing the quadratic tables is taken into account. Still, for $\ell \geq 3$ (Figure 12g), our approach dominates even when

[3] We note that n in this section refers to the size of one relation, in contrast to the entire database size as in previous sections.



(a) Query Q_{S1}, $\ell = 2$. (b) Query Q_{S1}, $\ell = 4$. (c) Query Q_{S2}, $\ell = 2$. (d) Query Q_{S2}, $\ell = 4$.

Figure 11: Section 7.2.1: Synthetic data with a growing database size n. While all three alternative methods either run out of memory or exceed a reasonable running time, our method scales quasilinearly ($O(n \operatorname{polylog} n)$) with n.



(a) Query Q_{R1}, $\ell = 2$. (b) Query Q_{R1}, $\ell = 3$. (c) Query Q_{R1}, different lengths ℓ. (d) Query Q_B, fixed $\varepsilon = 0.01$.

(e) Query Q_{R2}, $\ell = 2$. (f) Query Q_{R2}, $\ell = 3$. (g) Query Q_{R2}, different lengths ℓ. (h) Query Q_B, different bands ε.

Figure 12: Section 7.2.2: a,b,c,e,f,g: Section 7.2.3: d, h: Temporal paths of different lengths on RedditTitles (left), and spatial band-join on OceaniaBirds (right). Our method is robust to increasing query sizes or band-join ranges.

compared against these lower bounds. PSQL manages to terminate for $\ell = 3$ (Figure 12f) but is orders of magnitude slower.

7.2.3 Effect of Band Parameter. We now test the band-join Q_B on the OceaniaBirds dataset with various band parameters ε. Figure 12d shows that Factorized Any-K is superior for all tested k values when we fix $\varepsilon = 0.01$. Increasing the band parameter yields more joining pairs and causes the size of the output to grow (Figure 12h). Hence, QuadEqui and Batch consume more and more memory and cannot handle $\varepsilon \geq 0.16$. On the other hand, the performance of Factorized Any-K is mildly affected by increasing ε. PSQL did not terminate after 6 hours, even for the smallest ε. We found that band joins are poorly handled in that system, resulting in a nested loop execution even when an index is available.

8 RELATED WORK

(Ranked) Enumeration of Query Results. If projections are involved, unranked enumeration for equi-joins can be performed with linear preprocessing and constant delay for the class of free-connex acyclic queries [7]. In fact, no other equi-join query (excluding those with self-joins) admits such an enumeration algorithm, under fine-grained complexity assumptions [7, 14]. Similar dichotomies have been pursued by later works in more general settings, by allowing updates on the database [11], unions of queries [17], or functional dependencies [18]. Multiple surveys exist on this topic [10, 32, 75]. If we require a specific order on the output, then the more demanding task of ranked enumeration requires logarithmic delay, which is unavoidable [29] assuming the $X + Y$ conjecture [16]. Drawing and extending ideas from k shortest paths [34, 42, 48], Dynamic Programming [13, 26], and k-best enumeration [35, 56], we recently presented [78] two competing algorithmic approaches to this problem. Focusing on binary equi-joins, Ding et al. [31] approach ranked enumeration from a mainly practical perspective. The related problems of enumerating in provably random order [20] and directly accessing any result via its rank [19] have also been considered, but are limited to equi-joins.

Non-Equality Predicates ($\neq$). Works that target *non-equality*[4] predicates mainly rely on *color-coding* [5], a technique that was originally developed for subgraph isomorphism. Papadimitriou and Yannakakis [73] apply color-coding to conjunctive queries, establishing that non-equalities can be removed by paying an $O(\log^2 n)$ factor (in data complexity), even if they create cycles. The same core idea is leveraged by the (unranked) enumeration algorithm of

[4] A *non-equality* is sometimes referred to as an *inequality* [53] or *disequality* [7] in the literature.

Bagan et al. [7]. and by Koutris et al. [53] who offer batch algorithms for non-equalities (which compute the entire set of results). Queries with negation can be answered by rewriting them with *not-all-equal-predicates* [50], a generalization of non-equality. Compared to the color-coding technique, our approach (1) offers a unified perspective for a wide range of predicates and not just non-equalities, (2) gives a factorization of the query results, and (3) is asymptotically faster by $O(\log n)$ in the case of one non-equality predicate (and arbitrary equalities) per pairwise join.

Inequality Predicates (<). Khayatt et al. [52] provide optimized and distributed *batch* algorithms for up to two inequalities per join. In general, inequality predicates can be resolved by building data structures (indexes) that return the matching (right) tuples when probed with a (left) tuple [45]. Accessing these data structures requires time that is $O(\log n)$ or higher (e.g. $O(\log^{p-1} n)$ for range trees [27] with $p > 1$ inequalities). In the same spirit, Khamis et al. [2] rely on Chazelle's data structures [23] for aggregate computation under inequality predicates. *Unranked enumeration* for conjunctions of inequalities is possible with these data structures, yet the delay between results is (poly)logarithmic. Idris et al. [45] also report an $O(\log n)$ delay for unranked enumeration by sorting lexicographically on the inequality attributes[5]. In contrast, our approach achieves *constant delay* for conjunctions of inequalities (because of the constant-depth TLFGs). For *ranked enumeration*, it is not clear how any of these prior data structures can be used other than the QuadEqui approach, which is our baseline.

Factorized Databases. Factorized representations of query results [8, 69] have been proposed for *equi-joins* as a way to eliminate redundancies while still being useful for other tasks. These include result enumeration [71, 72], aggregate computation [8], and even machine learning [3, 54, 68, 74]. We provide factorized representations for general join predicates and leverage them for (ranked) enumeration. Factorizations have also been proposed for (equi-join) provenance polynomials [70, 71] where the atomic unit of the factorization is a tuple, similarly to our work. Other representation schemes are also being explored [28, 49]. For probabilistic databases, factorization of non-equalities [66] and inequalities [67] is possible with OBDDs. Although these are designed for a different purpose, the latter exploits the transitivity of inequality, similarly to our "shared ranges" approach (Figure 8a). We remind the reader that the latter is lacking in terms of enumeration delay and it is unclear how to generalize it to conjunctions of inequalities.

Top-k Joins. Top-k queries are a special case of ranked enumeration where the value of k is given explicitly and its knowledge can be exploited by the algorithm. Fagin et al. [36] present the Threshold Algorithm, which has surprisingly strong optimality properties in terms of the number of tuples that are fetched from some external source, also known as the "middleware" cost model. Since this algorithm assumes restricted key-to-key joins, later works generalize it to more general joins [37, 46, 58, 84], which may even involve theta-joins [60]. However, they retain the middleware cost model, hence do not provide any non-trivial guarantees when the actual join cost is taken into account [79]. Ilyas et al. [47] survey some of these approaches, along with some related ones such as building top-k indexes [22, 77] or views [25, 43].

Optimal Join Algorithms. A significant progress has been made towards join algorithms that achieve asymptotic guarantees as close as possible to some lower bound. Acyclic equi-joins are evaluated optimally in $O(n + |\text{out}|)$ by the Yannakakis algorithm [87], where $|\text{out}|$ is the output size. This bound is unattainable for cyclic queries [64], thus worst-case optimal join (WCOJ) algorithms [61, 64, 65, 82] settle for the AGM bound [6], which is the worst-case output size. Improvements over WCOJ algorithms have been made by applying (hyper)tree decompositions to get output-sensitive guarantees [4, 39], while a geometric perspective has led to even stronger notions of optimality [51, 63]. Ngo [62] recounts the development of these ideas. That line of work focuses on *producing all the query results* or Boolean queries, while our work focuses on (ranked) enumeration. The enumeration guarantees we achieve for acyclic queries are within logarithmic factors (which are often ignored in this area) of the lower bound $\text{TT}(k) = O(n + k)$.

9 CONCLUSIONS AND FUTURE WORK

We developed $O(n \operatorname{polylog} n)$-size representations for acyclic join queries with inequality-type predicates. We then leveraged them for enumeration algorithms with or without a ranking. While for a theta-join with a black-box join condition it seems unavoidable to avoid the $O(n^2)$ cost, it would be interesting to identify other types of joins that admit efficient representations. Moreover, it remains open whether our "shared ranges" factorization (which is more compact but leads to higher delay) can be generalized to conjunctions of inequalities. Last, we leave as future work the application of our techniques to cyclic joins.

Acknowledgements. This work was supported in part by the National Science Foundation (NSF) under award numbers CAREER IIS-1762268 and IIS-1956096.

REFERENCES

[1] 2020. Bird Occurrences in Oceania. https://doi.org/10.15468/dl.d6u6tj From https://www.gbif.org/.
[2] Mahmoud Abo Khamis, Ryan R. Curtin, Benjamin Moseley, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2019. On Functional Aggregate Queries with Additive Inequalities. In *PODS*. 414–431. https://doi.org/10.1145/3294052.3319694
[3] Mahmoud Abo Khamis, Hung Q Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2018. In-database learning with sparse tensors. In *PODS*. 325–340. https://doi.org/10.1145/3196959.3196960
[4] Mahmoud Abo Khamis, Hung Q Ngo, and Dan Suciu. 2017. What do Shannon-type Inequalities, Submodular Width, and Disjunctive Datalog have to do with one another?. In *PODS*. 429–444. https://doi.org/10.1145/3034786.3056105
[5] Noga Alon, Raphael Yuster, and Uri Zwick. 1995. Color-coding. *J. ACM* 42, 4 (1995), 844–856. https://doi.org/10.1145/210332.210337
[6] Albert Atserias, Martin Grohe, and Dániel Marx. 2013. Size Bounds and Query Plans for Relational Joins. *SIAM J. Comput.* 42, 4 (2013), 1737–1767. https://doi.org/10.1137/110859440
[7] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. 2007. On acyclic conjunctive queries and constant delay enumeration. In *International Workshop on Computer Science Logic (CSL)*. 208–222. https://doi.org/10.1007/978-3-540-74915-8_18
[8] Nurzhan Bakibayev, Tomáš Kočiský, Dan Olteanu, and Jakub Závodný. 2013. Aggregation and Ordering in Factorised Databases. *PVLDB* 6, 14 (2013), 1990–2001. https://doi.org/10.14778/2556549.2556579
[9] Nurzhan Bakibayev, Dan Olteanu, and Jakub Závodný. 2012. FDB: A Query Engine for Factorised Relational Databases. *PVLDB* 5, 11 (2012), 1232–1243. https://doi.org/10.14778/2350229.2350242
[10] Christoph Berkholz, Fabian Gerhardt, and Nicole Schweikardt. 2020. Constant Delay Enumeration for Conjunctive Queries: A Tutorial. *ACM SIGLOG News* 7, 1

[5] According to our understanding of their method, the number of binary searches needed for each result is actually $O(n)$, hence the delay is $O(n \log n)$ instead of $O(\log n)$ in the worst case.

(2020), 4–33. https://doi.org/10.1145/3385634.3385636
[11] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. 2017. Answering Conjunctive Queries Under Updates. In *PODS*. 303–318. https://doi.org/10.1145/3034786.3034789
[12] Christoph Berkholz and Nicole Schweikardt. 2019. Constant Delay Enumeration with FPT-Preprocessing for Conjunctive Queries of Bounded Submodular Width. In *44th International Symposium on Mathematical Foundations of Computer Science (MFCS) (LIPIcs)*, Vol. 138. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 58:1–58:15. https://doi.org/10.4230/LIPIcs.MFCS.2019.58
[13] Dimitri P. Bertsekas. 2005. *Dynamic Programming and Optimal Control* (3rd ed.). Vol. I. Athena Scientific. http://www.athenasc.com/dpbook.html
[14] Johann Brault-Baron. 2013. *De la pertinence de l'énumération: complexité en logiques propositionnelle et du premier ordre*. Ph.D. Dissertation. Université de Caen. https://hal.archives-ouvertes.fr/tel-01081392
[15] Johann Brault-Baron. 2016. Hypergraph Acyclicity Revisited. *ACM Comput. Surv.* 49, 3, Article 54 (Dec. 2016), 26 pages. https://doi.org/10.1145/2983573
[16] David Bremner, Timothy M Chan, Erik D Demaine, Jeff Erickson, Ferran Hurtado, John Iacono, Stefan Langerman, and Perouz Taslakian. 2006. Necklaces, convolutions, and X+ Y. In *European Symposium on Algorithms*. Springer, 160–171. https://doi.org/10.1007/s00453-012-9734-3
[17] Nofar Carmeli and Markus Kröll. 2019. On the Enumeration Complexity of Unions of Conjunctive Queries. In *PODS*. 134–148. https://doi.org/10.1145/3294052.3319700
[18] Nofar Carmeli and Markus Kröll. 2020. Enumeration Complexity of Conjunctive Queries with Functional Dependencies. *Theory Comput. Syst.* 64, 5 (2020), 828–860. https://doi.org/10.1007/s00224-019-09937-9
[19] Nofar Carmeli, Nikolaos Tziavelis, Wolfgang Gatterbauer, Benny Kimelfeld, and Mirek Riedewald. 2020. Tractable Orders for Direct Access to Ranked Answers of Conjunctive Queries. *CoRR* abs/2012.11965 (2020). arXiv:2012.11965
[20] Nofar Carmeli, Shai Zeevi, Christoph Berkholz, Benny Kimelfeld, and Nicole Schweikardt. 2020. Answering (Unions of) Conjunctive Queries Using Random Access and Random-Order Enumeration. In *PODS*. 393–409. https://doi.org/10.1145/3375395.3387662
[21] Lijun Chang, Xuemin Lin, Wenjie Zhang, Jeffrey Xu Yu, Ying Zhang, and Lu Qin. 2015. Optimal enumeration: Efficient top-k tree matching. *PVLDB* 8, 5 (2015), 533–544. https://doi.org/10.14778/2735479.2735486
[22] Yuan-Chi Chang, Lawrence Bergman, Vittorio Castelli, Chung-Sheng Li, Ming-Ling Lo, and John R Smith. 2000. The onion technique: indexing for linear optimization queries. In *SIGMOD*. 391–402. https://doi.org/10.1145/342009.335433
[23] Bernard Chazelle. 1988. Functional approach to data structures and its use in multidimensional searching. *SIAM J. Comput.* 17, 3 (1988), 427–462. https://doi.org/10.1137/0217026
[24] Yves Crama and Peter L Hammer. 2011. *Boolean functions: Theory, algorithms, and applications*. Cambridge University Press. https://doi.org/10.1017/CBO9780511852008
[25] Gautam Das, Dimitrios Gunopulos, Nick Koudas, and Dimitris Tsirogiannis. 2006. Answering top-k queries using views. In *VLDB*. 451–462. https://dl.acm.org/doi/10.5555/1182635.1164167
[26] Sanjoy Dasgupta, Christos H Papadimitriou, and Umesh Virkumar Vazirani. 2008. *Algorithms*. McGraw-Hill Higher Education. https://dl.acm.org/doi/book/10.5555/1177299
[27] Mark De Berg, Marc Van Kreveld, Mark Overmars, and Otfried Schwarzkopf. 1997. Computational geometry. In *Computational geometry*. Springer, 1–17. https://doi.org/10.1007/978-3-540-77974-2
[28] Shaleen Deep and Paraschos Koutris. 2018. Compressed representations of conjunctive query results. In *PODS*. 307–322. https://doi.org/10.1145/3196959.3196979
[29] Shaleen Deep and Paraschos Koutris. 2021. Ranked Enumeration of Conjunctive Query Results. In *ICDT*. http://arxiv.org/abs/1902.02698
[30] David J. DeWitt, Jeffrey F. Naughton, and Donovan A. Schneider. 1991. An Evaluation of Non-Equijoin Algorithms. In *VLDB*. 443–452. https://dl.acm.org/doi/10.5555/645917.672320
[31] Mengsu Ding, Shimin Chen, Nantia Makrynioti, and Stefan Manegold. 2021. Progressive Join Algorithms Considering User Preference. In *CIDR*. https://ir.cwi.nl/pub/30501/30501.pdf
[32] Arnaud Durand. 2020. Fine-Grained Complexity Analysis of Queries: From Decision to Counting and Enumeration. In *PODS*. 331–346. https://doi.org/10.1145/3375395.3389130
[33] Jost Enderle, Matthias Hampel, and Thomas Seidl. 2004. Joining Interval Data in Relational Databases. In *SIGMOD*. 683–694. https://doi.org/10.1145/1007568.1007645
[34] David Eppstein. 1998. Finding the k shortest paths. *SIAM J. Comput.* 28, 2 (1998), 652–673. https://doi.org/10.1137/S0097539795290477
[35] David Eppstein. 2016. *k-Best Enumeration*. Springer, Encyclopedia of Algorithms, 1003–1006. https://doi.org/10.1007/978-1-4939-2864-4_733
[36] Ronald Fagin, Amnon Lotem, and Moni Naor. 2003. Optimal aggregation algorithms for middleware. *J. Comput. System Sci.* 66, 4 (2003), 614–656. https://doi.org/10.1016/S0022-0000(03)00026-6
[37] Jonathan Finger and Neoklis Polyzotis. 2009. Robust and efficient algorithms for rank join evaluation. In *SIGMOD*. 415–428. https://doi.org/10.1145/1559845.1559890
[38] Michel Gondran and Michel Minoux. 2008. *Graphs, Dioids and Semirings: New Models and Algorithms (Operations Research/Computer Science Interfaces Series)*. Springer. https://doi.org/10.1007/978-0-387-75450-5
[39] Georg Gottlob, Gianluigi Greco, Nicola Leone, and Francesco Scarcello. 2016. Hypertree Decompositions: Questions and Answers. In *PODS*. 57–74. https://doi.org/10.1145/2902251.2902309
[40] M.H. Graham. 1979. *On the universal relation*. Technical Report. Univ. of Toronto.
[41] C. A. R. Hoare. 1962. Quicksort. *Comput. J.* 5, 1 (01 1962), 10–16. https://doi.org/10.1093/comjnl/5.1.10
[42] Walter Hoffman and Richard Pavley. 1959. A Method for the Solution of the Nth Best Path Problem. *J. ACM* 6, 4 (1959), 506–514. https://doi.org/10.1145/320998.321004
[43] Vagelis Hristidis, Nick Koudas, and Yannis Papakonstantinou. 2001. PREFER: A system for the efficient execution of multi-parametric ranked queries. *SIGMOD Record* 30, 2 (2001), 259–270. https://doi.org/10.1145/375663.375690
[44] Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. 2019. Efficient Query Processing for Dynamically Changing Datasets. *SIGMOD Record* 48, 1 (2019), 33–40. https://doi.org/10.1145/3371316.3371325
[45] Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. 2020. General dynamic Yannakakis: conjunctive queries with theta joins under updates. *VLDB J.* 29 (2020), 619–653. https://doi.org/10.1007/s00778-019-00590-9
[46] Ihab F Ilyas, Walid G Aref, and Ahmed K Elmagarmid. 2004. Supporting top-k join queries in relational databases. *VLDB J.* 13, 3 (2004), 207–221. https://doi.org/10.1007/s00778-004-0128-2
[47] Ihab F Ilyas, George Beskales, and Mohamed A Soliman. 2008. A survey of top-k query processing techniques in relational database systems. *Comput. Surveys* 40, 4 (2008), 11. https://doi.org/10.1145/1391729.1391730
[48] Víctor M Jiménez and Andrés Marzal. 1999. Computing the K shortest paths: A new algorithm and an experimental comparison. In *International Workshop on Algorithm Engineering (WAE)*. Springer, 15–29. https://doi.org/10.1007/3-540-48318-7_4
[49] Ahmet Kara and Dan Olteanu. 2018. Covers of Query Results. In *ICDT*. 16:1–16:22. https://doi.org/10.4230/LIPIcs.ICDT.2018.16
[50] Mahmoud Abo Khamis, Hung Q. Ngo, Dan Olteanu, and Dan Suciu. 2019. Boolean Tensor Decomposition for Conjunctive Queries with Negation. In *ICDT*. 21:1–21:19. https://doi.org/10.4230/LIPIcs.ICDT.2019.21
[51] Mahmoud Abo Khamis, Hung Q. Ngo, Christopher Ré, and Atri Rudra. 2016. Joins via Geometric Resolutions: Worst Case and Beyond. *TODS* 41, 4, Article 22 (2016), 45 pages. https://doi.org/10.1145/2967101
[52] Zuhair Khayyat, William Lucia, Meghna Singh, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Panos Kalnis. 2017. Fast and scalable inequality joins. *VLDB J.* 26, 1 (2017), 125–150. https://doi.org/10.1007/s00778-016-0441-6
[53] Paraschos Koutris, Tova Milo, Sudeepa Roy, and Dan Suciu. 2017. Answering Conjunctive Queries with Inequalities. *Theory of Computing Systems* 61, 1 (2017), 2–30. https://doi.org/10.1007/s00224-016-9684-2
[54] Arun Kumar, Jeffrey Naughton, and Jignesh M Patel. 2015. Learning generalized linear models over normalized data. In *SIGMOD*. 1969–1984. https://doi.org/10.1145/2723372.2723713
[55] Srijan Kumar, William L Hamilton, Jure Leskovec, and Dan Jurafsky. 2018. Community interaction and conflict on the web. https://snap.stanford.edu/data/soc-RedditHyperlinks.html. In *WWW*. 933–943.
[56] Eugene L Lawler. 1972. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management science* 18, 7 (1972), 401–405. https://doi.org/10.1287/mnsc.18.7.401
[57] Rundong Li, Wolfgang Gatterbauer, and Mirek Riedewald. 2020. Near-Optimal Distributed Band-Joins through Recursive Partitioning. In *SIGMOD*. 2375–2390. https://doi.org/10.1145/3318464.3389750
[58] Nikos Mamoulis, Man Lung Yiu, Kit Hung Cheng, and David W Cheung. 2007. Efficient top-k aggregation of ranked inputs. *TODS* 32, 3 (2007), 19. https://doi.org/10.1145/1272743.1272749
[59] Mehryar Mohri. 2002. Semiring Frameworks and Algorithms for Shortest-distance Problems. *J. Autom. Lang. Comb.* 7, 3 (Jan. 2002), 321–350. http://dl.acm.org/citation.cfm?id=639508.639512
[60] Apostol Natsev, Yuan-Chi Chang, John R Smith, Chung-Sheng Li, and Jeffrey Scott Vitter. 2001. Supporting incremental join queries on ranked inputs. In *VLDB*. 281–290. http://www.vldb.org/conf/2001/P281.pdf
[61] Gonzalo Navarro, Juan L. Reutter, and Javiel Rojas-Ledesma. 2020. Optimal Joins Using Compact Data Structures. In *ICDT*, Vol. 155. 21:1–21:21. https://doi.org/10.4230/LIPIcs.ICDT.2020.21
[62] Hung Q Ngo. 2018. Worst-case optimal join algorithms: Techniques, results, and open problems. In *PODS*. 111–124. https://doi.org/10.1145/3196959.3196990

[63] Hung Q Ngo, Dung T Nguyen, Christopher Re, and Atri Rudra. 2014. Beyond worst-case analysis for joins with minesweeper. In *PODS*. 234–245. https://doi.org/10.1145/2594538.2594547

[64] Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-case optimal join algorithms. *J. ACM* 65, 3 (2018), 16. https://doi.org/10.1145/3180143

[65] Hung Q Ngo, Christopher Ré, and Atri Rudra. 2014. Skew Strikes Back: New Developments in the Theory of Join Algorithms. *SIGMOD Record* 42, 4 (Feb. 2014), 5–16. https://doi.org/10.1145/2590989.2590991

[66] Dan Olteanu and Jiewen Huang. 2008. Using OBDDs for efficient query evaluation on probabilistic databases. (2008), 326–340. https://doi.org/10.1007/978-3-540-87993-0_26

[67] Dan Olteanu and Jiewen Huang. 2009. Secondary-storage confidence computation for conjunctive queries with inequalities. In *SIGMOD*. 389–402. https://doi.org/10.1145/1559845.1559887

[68] Dan Olteanu and Maximilian Schleich. 2016. F: Regression Models over Factorized Views. *PVLDB* 9, 13 (2016), 1573–1576. https://doi.org/10.14778/3007263.3007312

[69] Dan Olteanu and Maximilian Schleich. 2016. Factorized databases. *SIGMOD Record* 45, 2 (2016). https://doi.org/10.1145/3003665.3003667

[70] Dan Olteanu and Jakub Závodnỳ. 2011. On factorisation of provenance polynomials. In *TaPP*. https://www.usenix.org/conference/tapp11/factorisation-provenance-polynomials

[71] Dan Olteanu and Jakub Závodnỳ. 2012. Factorised representations of query results: size bounds and readability. In *ICDT*. 285–298. https://doi.org/10.1145/2274576.2274607

[72] Dan Olteanu and Jakub Závodnỳ. 2015. Size bounds for factorised representations of query results. *TODS* 40, 1 (2015), 2. https://doi.org/10.1145/2656335

[73] Christos H. Papadimitriou and Mihalis Yannakakis. 1999. On the complexity of database queries. *J. Comput. System Sci.* 58, 3 (1999), 407–427. https://doi.org/10.1006/jcss.1999.1626

[74] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. 2016. Learning linear regression models over factorized joins. In *SIGMOD*. 3–18. https://doi.org/10.1145/2882903.2882939

[75] Luc Segoufin. 2015. Constant Delay Enumeration for Conjunctive Queries. *SIGMOD Record* 44, 1 (2015), 10–17. https://doi.org/10.1145/2783888.2783894

[76] Robert E Tarjan and Mihalis Yannakakis. 1984. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.* 13, 3 (1984), 566–579. https://doi.org/10.1137/0213035

[77] Panayiotis Tsaparas, Themistoklis Palpanas, Yannis Kotidis, Nick Koudas, and Divesh Srivastava. 2003. Ranked join indices. In *ICDE*. IEEE, 277–288. https://doi.org/10.1109/ICDE.2003.1260799

[78] Nikolaos Tziavelis, Deepak Ajwani, Wolfgang Gatterbauer, Mirek Riedewald, and Xiaofeng Yang. 2020. Optimal Algorithms for Ranked Enumeration of Answers to Full Conjunctive Queries. *PVLDB* 13, 9 (2020), 1582–1597. https://doi.org/10.14778/3397230.3397250

[79] Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. 2020. Optimal Join Algorithms Meet Top-k. In *SIGMOD*. 2659–2665. https://doi.org/10.1145/3318464.3383132

[80] Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. 2021. Beyond Equi-joins: Ranking, Enumeration and Factorization. *CoRR* abs/2101.12158 (2021). arXiv:2101.12158

[81] Moshe Y. Vardi. 1982. The Complexity of Relational Query Languages (Extended Abstract). In *STOC*. 137–146. https://doi.org/10.1145/800070.802186

[82] Todd L. Veldhuizen. 2014. Triejoin: A Simple, Worst-Case Optimal Join Algorithm. In *ICDT*. 96–106. https://doi.org/10.5441/002/icdt.2014.13

[83] Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. 2014. Path Problems in Temporal Graphs. *PVLDB* 7, 9 (2014), 721–732. https://doi.org/10.14778/2732939.2732945

[84] Minji Wu, Laure Berti-Equille, Amélie Marian, Cecilia M Procopiuc, and Divesh Srivastava. 2010. Processing top-k join queries. *PVLDB* 3, 1 (2010), 860–870. https://doi.org/10.14778/1920841.1920951

[85] Xiaofeng Yang, Deepak Ajwani, Wolfgang Gatterbauer, Patrick K Nicholson, Mirek Riedewald, and Alessandra Sala. 2018. Any-k: Anytime Top-k Tree Pattern Retrieval in Labeled Graphs. In *WWW*. 489–498. https://doi.org/10.1145/3178876.3186115

[86] Xiaofeng Yang, Mirek Riedewald, Rundong Li, and Wolfgang Gatterbauer. 2018. Any-k Algorithms for Exploratory Analysis with Conjunctive Queries. In *International Workshop on Exploratory Search in Databases and the Web (ExploreDB)*. 1–3. https://doi.org/10.1145/3214708.3214711

[87] Mihalis Yannakakis. 1981. Algorithms for Acyclic Database Schemes. In *VLDB*. 82–94. https://dl.acm.org/doi/10.5555/1286831.1286840

[88] Clement Tak Yu and Meral Z Ozsoyoglu. 1979. An algorithm for tree-query membership of a distributed query. In *COMPSAC*. IEEE, 306–312. https://doi.org/10.1109/CMPSAC.1979.762509

A NOMENCLATURE

Symbol	Definition
Q	Join query
R, S, T	Relations
A, B, C	Attributes
$\mathbf{X}, \mathbf{Y}, \mathbf{Z}$	Lists of attributes
r, s, t	Tuples
θ	Join Predicate
$S \bowtie_\theta T$	Join between S, T on predicate θ
n	Total number of tuples
d	Number of distinct values
ℓ	Number of relations
q	Number of predicates in the query
$G(V, E)$	Graph with nodes V and edges E
v_s, v_t	Nodes corresponding to tuples $s \in S, t \in T$
$\mathcal{S}$	Size of TLFG
λ	Depth of TLFG
u	Duplication factor of TLFG
p	Number of conjuncts or disjuncts
ρ	Number of partitions in equality/inequality factorization
M_i	Partition in inequality factorization
m	Number of groups in band factorization
H_i	Group in band factorization
$\mathrm{TT}(k)$	Time-to-k^{th} result
$\mathrm{MEM}(k)$	Memory until the k^{th} result
$\mathcal{T}$	Time for constructing a TLFG
$\mathcal{P}(n)$	Time for preprocessing
f, g	(Computable) functions

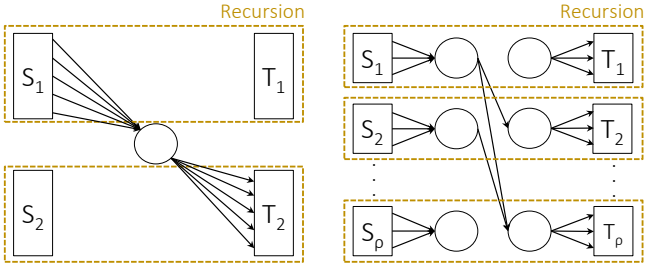
B MULTIWAY PARTITIONING

We provide more details on the multiway partitioning method discussed in Section 6.1. Recall that it constitutes an improvement over the binary partitioning method of Section 4.1 for the case of a single inequality predicate. More specifically, it creates a TLFG of size $O(n \log\log n)$ instead of $O(n \log n)$, while only increasing the depth to 3 from 2 (see Fig. 8b).

The main idea is to create more data partitions per recursive step. In particular, we pick $\rho - 1$ pivots that create ρ partitions of nodes with a roughly equal number of distinct values. Fig. 13b depicts how the partitions are connected for a less-than ($<$) predicate. Each source partition $S_i, i \in [1, \rho - 1]$ is connected to all target partitions $T_j, j \in [i + 1, \rho]$, since all values in S_i are guaranteed to be smaller than all values in T_j. The ideal number of partitions is $\Theta(\sqrt{d})$, so that the connections between them can be built in $O(\sqrt{d}^2) = O(n)$, i.e., the same that binary partitioning needs per recursive step. The advantage of the multiple partitions is that we can reach the base case $d = 1$ faster since each partition is smaller. Algorithm 2 shows the pseudocode of this approach.

Lemma 19. *Let θ be an inequality predicate between relations S, T of total size n. A duplicate-free TLFG of the join $S \bowtie_\theta T$ of size $O(n \log\log n)$ and depth 3 can be constructed in $O(n \log n)$ time.*

Proof. The arguments for correctness and the duplicate-free property are similar to the case of binary partitioning (Lemma 9). For the depth, notice that all the edges we create are either from the source nodes to a layer of x nodes (Line 13) or from x nodes to a layer of y nodes (Line 15) or from y nodes to target nodes(Line 14). Thus, all paths from source to target nodes have a length of 3. The running time is dominated by the $O(n \log n)$ initial sorting of the relations, but the recursion (which bounds the space consumption) is now more efficient than the binary partitioning case. Each recursive step with size $|S| + |T| = n$ requires $O(n)$ to partition the sorted relations. Then, we materialize $O(n)$ edges for source and target nodes, $O(\sqrt{d})$ intermediate nodes and $O(\sqrt{d}^2)$ edges between them. This adds up to $O(n)$ because $d \leq n$. We then invoke $\rho = \lceil \sqrt{d} \rceil = O(\sqrt{n})$ recursive calls with sizes $n_1 + n_2 + \ldots + n_\rho = n$. Therefore, in every level of the recursion tree, the sizes of all the subproblems add up to n. Since we spend linear time per problem, the total work per level of the recursion tree is $O(n)$. The height δ of the tree is the number of times we have to take the square root of d (and then the ceil function) in order to reach $d = 1$, which is $O(\log\log d) = O(\log\log n)$. To see this, observe that $d^{(\frac{1}{2})^\delta} = 2 \Rightarrow (\frac{1}{2})^\delta \log d = 1 \Rightarrow \delta = \log\log d$. Overall, the time spent on the recursion and thus, the size of the TLFG is bounded by $O(n \log\log n)$. □



(a) 2 partitions: depth 2 with one intermediate node. **(b) ρ partitions: depth 3 with 2ρ intermediate nodes.**

Figure 13: Binary vs Multi-way partitioning for inequalities.

Algorithm 2: Multiway partitioning

```
1  Input: Relations S, T, nodes v_s, v_t for s ∈ S, t ∈ T,
2         predicate θ ≡ S.A < T.B
3  Output: A TLFG of the join S ⋈_θ T
4  Sort S, T according to attributes A, B
5  partIneqMulti(S, T, θ)
6  Procedure partIneqMulti(S, T, θ)
7      d = vals(S ∪ T)//Number of distinct A, B values
8      if d == 1 then return //Base case
9      ρ = ⌈√(d)⌉ //Number of partitions
10     Partition (S ∪ T) into (S_1 ∪ T_1), ..., (S_ρ ∪ T_ρ) with
         ρ-quantiles of distinct values as pivots
11     for i ← 1 to ρ do
12         Materialize intermediate nodes x_i, y_i
13         foreach s in S_i do Create edge v_s ⟶ x_i
14         foreach t in T_i do Create edge y_i ⟶ v_t
15         for j ← 1 to i − 1 do Create edge x_j ⟶ y_i
16         partIneqMulti(S_i, T_i, θ) //Recursive call
```

C NON-EQUALITY PREDICATES

A non-equality condition $S.A \neq T.B$ is satisfied if either $S.A < T.B$ or $S.A > T.B$. Even though it can be modeled as a disjunction of two inequalities, we now establish that (in contrast to arbitrary disjunctions), they do not increase the TLFG duplication factor. The main observation is that the pairs which satisfy one of the

inequalities cannot satisfy the other one. Therefore, if we union the two inequality TLFGs no path will be duplicated. The guarantees we obtain are the same as the inequality case by using multiway partitioning (once for each inequality).

Lemma 20. *Let θ be an non-equality predicate between relations S, T of total size n. A duplicate-free TLFG of the join $S \bowtie_\theta T$ of size $O(n \log\log n)$ and depth 3 can be constructed in $O(n \log n)$ time.*

Proof. We sort once in $O(n \log n)$ and then call the inequality multiway partitioning algorithm twice. Thus, we have to spend two times $O(n \log\log n)$ time and space. The depth of the final TLFG is still 3 since the two TLFGs are constructed independently. It also remains duplicate-free since the two inequality conditions cannot hold simultaneously. Suppose that the calls to partIneqMulti(S, T, S.A < T.B) and partIneqMulti(S, T, S.A > T.B) both create a path between v_s and v_t for two tuples $s \in S, t \in T$. Then, the two tuples would have to satisfy $s.A < t.B$ and $s.A > t.B$, which is impossible. □

D BAND PREDICATES

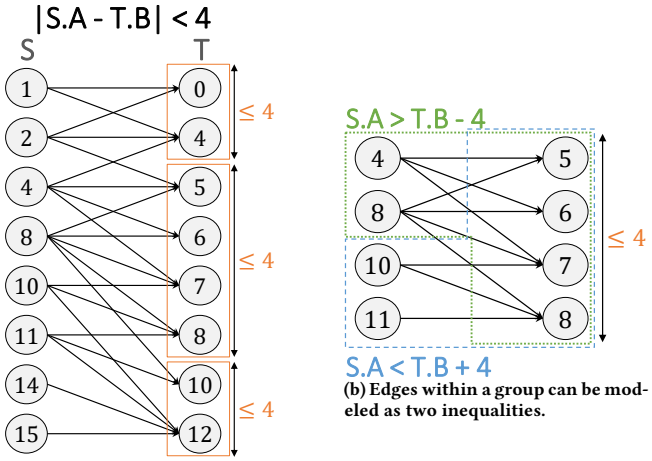
In this section, we target band predicates of the type $|S.A - T.B| < \varepsilon$. We provide an algorithm that leverages the structure of the band to achieve asymptotically the same guarantees as the inequality case. If a band condition is handled as a generic conjunction of inequalities, then the time spent, as well as the TLFG size are higher than our specialized construction.

Our algorithm translates the band problem into a set of inequality problems for smaller groups of tuples, which can then be solved independently. First, we describe the intuition. The band predicate consists of two inequalities ($S.A < T.B+\varepsilon$) and ($S.A > T.B-\varepsilon$) that need to hold simultaneously. If for some source-target tuples we can guarantee that one of the two inequalities is always satisfied, then it suffices to use the inequality algorithm we developed in Section 4.1 for the other one. Therefore, the idea is to create groups of tuples with that property and cover all the possible joining pairs with these groups.

The first step is to sort the input relations and group the tuples of the target relation into maximal ε-intervals. More specifically, we start from the first T tuple and group together all those whose B values are at most ε apart from it. We then repeat the same process starting from the T tuple that is immediately after the group, creating $m \leq n$ groups, whose range of B values is at most ε. A source tuple is assigned to a group if it joins with at least one target tuple in the group. Since the groups represent ε-intervals of target tuples, *each source tuple can be assigned to at most three groups.*

Example 21. *Figure 14 depicts an example with $\varepsilon = 4$. Notice that as the number of tuples grows, the output is $O(n^2)$, e.g., if the domain is fixed or if ε grows together with the domain size. Initially, we group the target tuples by ε intervals (Fig. 14a). Thus, the first group starts with the first T tuple 0, and ends before 5 since $5 - 0 > \varepsilon = 4$. This process creates three groups of target tuples, each one having a range of B values bounded by 4. Then, a source tuple is assigned to a group by comparing its A value with the limits of the group. For instance, tuple 11 is assigned to the middle group because $5 - 4 < 11 < 8 + 4$, hence it joins with at least one target tuple in that group.*

After the assignment of tuples to groups, we work on each group separately. For example, consider the middle group depicted in Fig. 14b. Source tuple 4 joins with the top T tuple 5, which means that the pair (4, 5) satisfies both inequalities. From that we can infer that 4 satisfies the less-than inequality with all the target tuples in the group, *since their B values are at least 5. Thus, we can handle it by using our inequality algorithm for the greater-than condition ($S.A > T.B - \varepsilon$). Conversely, tuple 10 joins with the bottom T tuple 8, thus satisfies the greater-than inequality with all the target tuples in the group. For that tuple, we only have to handle the less-than inequality ($S.A < T.B + \varepsilon$). Notice that all the source tuples in the group are covered by at least one of the above scenarios.*



(a) Edges between all $O(n^2)$ joining pairs and grouping with ε-intervals.

(b) Edges within a group can be modeled as two inequalities.

Figure 14: Example 21: TLFG construction for band conditions.

For each group of source-target tuples we created, there are three cases for the S tuples: (1) those who join with the top target tuple but not the bottom, (2) those who join with the bottom target tuple but not the top, (3) those who join with all the target tuples. These are the only three cases since by construction of the group, the distance between the target tuples is at most ε. Case (1) can be handled as a greater-than TLFG, case (2) as a less-than, and case (3) as either one of them. As Algorithm 3 shows, `partIneqMulti()` is called twice for each group.

Lemma 22. *Let θ be a band predicate between relations S, T of total size n. A duplicate-free TLFG of the join $S \bowtie_\theta T$ of size $O(n \log\log n)$ and depth 3 can be constructed in $O(n \log n)$ time.*

Proof. First, we create disjoint T groups based on ε-intervals and assign each S tuple to all groups where it has joining partners (Lines 9 to 16). This can be done with binary search in $O(n \log n)$. Each T tuple is assigned to a single group. An S tuple cannot be assigned to more than three consecutive groups since their values span a range of at least 2ε. Within each group $H_j = (S_j \cup T_j)$, the correctness of our algorithm follows from the fact that the T_j tuples are at most ε apart on the B attribute. Since all the assigned S_j tuples

Algorithm 3: Handling a band predicate

```
Input: Relations S, T, nodes v_s, v_t for s ∈ S, t ∈ T,
       predicate θ ≡ |S.A − T.B| < ε
Output: A TLFG of the join S ⋈_θ T
Sort S, T according to attributes A, B
foreach (S_b, T_b, θ_b) in bandToIneq (S, T, θ) do
  partIneqMulti(S_b, T_b, θ_b)
Function bandToIneq(S, T, θ)
  ineqs = []
  //Find the limits of the groups on the right
  H_1.start = t_1.B, m = 1
  for i ← 1 to |T| do
    if t[i].B > H_m.start + ε then
      H_m.end = T[i − 1].B
      m++
      H_m.start = T[i].B
  H_m.end = T[i].B
  foreach H_j in [H_1, ..., H_m] do
    //Assign tuples to the group
    S_j = [s ∈ S | H_j.start − ε ≤ s.A ≤ H_j.end + ε]
    T_j = [t ∈ T | H_j.start ≤ t.B ≤ H_j.end]
    //Greater-than inequality
    S_> = [s ∈ S_j | s.A < H_j.start + ε]
    ineqs.add((S_<, T_j, S.A > T.B − ε))
    //Less-than inequality
    ineqs.add((S_j − S_>, T_j, S.A < T.B + ε))
    return ineqs
```

have at least one joining partner in T_j, they have to join either with the first T_j tuple (in sorted B order) or with the last one. Recall that the band condition can be rewritten as $(S.A < T.B + \varepsilon) \wedge (S.A > T.B - \varepsilon)$, i.e., two inequality conditions that both have to be satisfied. In case some $s \in S_j$ joins with the first T_j tuple, then we know that the less-than condition is always satisfied for s within the group H_j. Thus, we just need to connect v_s with all v_t for $t \in T_j$ that satisfy the greater-than condition. We argue similarly for the case when s joins with the last tuple of T_j, where we have to take care only of the less-than condition. Finally, there is also the possibility that s joins with all T_j tuples. In that case, both inequality conditions are satisfied – we assign those tuples to only one of the inequalities which ensures the duplicate-free property. For the running time, the total size of the groups we create is $n_1 + n_2 + \ldots + n_m \leq 3n$. If for a problem of size $|S| + |T| = n$ where the relations have been sorted, $\mathcal{T}_B(n)$ is the time for factorizing a band condition and $\mathcal{T}_I(n)$ for an inequality, we have $\mathcal{T}_B(n) = O(n) + 2\mathcal{T}_I(n_1) + 2\mathcal{T}_I(n_2) + \ldots + 2\mathcal{T}_I(n_m)$, since we call the inequality algorithm twice within each group. For $\mathcal{T}_I(n) = O(n \log \log n)$, we get $\mathcal{T}_B(n) = O(n \log \log n)$, which also bounds the size of the TLFG. Each call to the inequality algorithm involves different S, T pairs, giving us the duplicate-free property and the same depth as the inequality TLFG. □

E ADDITIONAL PROOFS

E.1 Proof of Lemma 11

To construct the TLFG for $S \bowtie_\theta T$, we gather all the equality predicates and use hashing to create partitions of tuples that correspond to equal joining values for the equality predicates. This takes $O(n)$. We then construct the TLFG for each partition independently with the conditions θ' through some algorithm $\mathcal{A}$. If $\mathcal{A}$ elects to connect two nodes, then they satisfy both θ', and also the equalities since they belong to the same partition. Conversely, two nodes that remain disconnected at the end of the process either do not belong to the same equality partition or were not connected by $\mathcal{A}$, thus do not satisfy θ'.

Assume that the number of tuples in each partition is $n_i, i \in [\rho]$ with $n_1 + \ldots + n_\rho = n$. The total time spent on each partition is $O(g(n_1) + \ldots + g(n_\rho))$ which by the superadditivity property of g is $O(g(n_1 + \ldots + n_\rho)) = O(g(n))$. The same argument applies to the size, giving us $O(f(n))$. Since the partitions are disjoint, we cannot create additional duplicate paths apart from the ones created by $\mathcal{A}$, or increase the depth of each TLFG.

E.2 Proof of Lemma 13

As a first step, all the equality predicates are handled by Lemma 11. Since the time and size guarantees we show are $O(n \log^p n)$ and $n \log^p n$ is a superadditive function, they are unaffected by this step. The remaining inequality predicates are handled by Algorithm 1. We denote by $\mathcal{T}_I(n, p)$ the running time for n tuples and p inequality predicates. We proceed by induction on the number of predicates p to show that $\mathcal{T}_I(n, p) \leq f(p) n \log^p n$ for some function f and sufficiently large n. First, assume that all the predicates are inequalities. For the base case $p = 1$, the analysis is the same as in the proof of Lemma 9: Tthe height of the recursion tree is $O(\log n)$ and the total time is $O(n \log n)$ together with sorting once. In other words, we have $\mathcal{T}_I(n, 1) \leq cn \log n$ for sufficiently large n. For the inductive step, we assume that $\mathcal{T}_I(n, p-1) \leq f(p-1) n \log^{p-1} n$. The inequality at the head of the list creates a recursion tree where every node has a subset of the tuples n' and calls the next inequality, thus is computed in $\mathcal{T}_I(n', p-1)$. The problem sizes in some level of the tree add up to $n_1 + \ldots + n_\rho = n$. Thus, the work per level is bounded by $\mathcal{T}_I(n_1, p-1) + \ldots + \mathcal{T}_I(n_t, p-1) \leq f(p-1) n_1 \log^{p-1} n_1 + \ldots + f(p-1) n_t \log^{p-1} n_t \leq f(p-1) n \log^{p-1} n$. The height of the tree is $O(\log n)$, thus the total work in the tree is bounded by $c' \log n f(p-1) n \log^{p-1} n = c' f(p-1) n \log^p n$. We also take into account the time for sorting according to the attributes of the current inequality, which is bounded by $c'' n \log n$. Thus, we get that $\mathcal{T}_I(n, p) \leq c' f(p-1) n \log^p n + c'' n \log n$. If we pick a function f such that $f(1) \geq c$ and $f(p) \geq c' f(p-1) + \frac{c''}{\log^{p-1} n}$, then $\mathcal{T}_I(n, p) \leq f(p) n \log^p n$. This completes the induction, establishing that $\mathcal{T}_I(n, p) = O(n \log^p n)$ in data complexity.

The size of the TLFG cannot exceed the running time, thus it is also $O(n \log^p n)$. The depth is 2 because in all cases we use the binary partitioning method and the duplication factor is 1 because we only connect tuples in the base case of one predicate $p = 1$, which we already proved does not create duplicates (Lemma 9).

E.3 Proof of Lemma 14

Correctness follows from the fact that the paths in the constructed TLFG is the union of the paths in the TLFGs for $S \bowtie_{\theta_i} T$. For the depth, note that each θ_i is processed independently, thus the component TLFGs do not share any nodes or edges other than the endpoints. A path from v_s to v_t for $s \in S, t \in T$ may only be duplicated by different TLFG constructions since each one is duplicate-free.

Thus, the duplication factor cannot exceed the number of predicates p.

E.4 Proof of Theorem 16

For each parent S and child T in the join tree, we construct a TLFG. According to Lemmas 13 and 14, the depth of each TLFG is $\lambda = 2$. For the TLFGs created by the binary partitioning method, there is only one layer of intermediate nodes. We introduce a variable V_1, and create two new relations E_1, E_2. Let $\mathbf{A}_S$, $\mathbf{A}_T$ be the attributes of S and T respectively. Relation E_1 contains attributes $\mathbf{A}_S \cup V_1$, and E_2 contains $V_1 \cup \mathbf{A}_T$. We add to the two new relations the edges from source to intermediate layer and from intermediate to target layer respectively. If an edge (v_s, v_1) exists for $s \in S, v_1 \in V_1$, we add a tuple (s, v_1) to relation E_1. Similarly, if an edge (v_1, v_t) exists for $t \in T, v_1 \in V_1$, we add a tuple (v_1, t) to relation E_2.

The size of the new relations and the time required for the entire construction follow directly from the TLFG guarantees.

We run enumeration on a query Q_F that is created by removing from Q all predicates and adding one atom for each new relation. Each answer to Q_F corresponds to an answer to Q because of the correctness of the TLFGs. The duplication factor is 1, except if we have disjunctions (Lemma 14). Let u_{max} be the maximum duplication factor among the constructed TLFGs. The number of "duplicate" answers we get in Q_F that correspond to the same answer q of Q are bounded by u_{max}^{ℓ}, where ℓ is the number of Q atoms. That depends only on the query size which we consider as constant, thus it is $O(1)$.

We enumerate the answers to Q_F and project out the values of the new V_1 variable as a post-processing step. First assume that we have no duplicates. In that case, the guarantees of this theorem follow immediately from known results on equi-join enumeration [7, 78]. If we have duplicates, then we can filter them by maintaining a lookup table. If the time for each answer without the filtering is $\mathrm{TT}'(k)$, then we have that $\mathrm{TT}(k) = O(\mathrm{TT}'(k \cdot u_{max}^{\ell})) = O(\mathrm{TT}'(k))$, since the number of duplicates per answer is $O(1)$.

E.5 Proof of Lemma 17

Lemmas 19, 20 and 22 together prove Lemma 17.

E.6 Proof of Theorem 18

The proof is the same as that of Theorem 16, but this time we use (1) multiway partitioning for the base case of $p = 1$ in the conjunction algorithm and (2) specialized constructions for non-equalities and bands that make their corresponding TLFGs have the same guarantees as the inequality case. We delineate the additional elements needed for the proof.

First, equalities are removed because of Lemma 11 and the fact that $n \log^p n$ and $n \log^{p-1} n \cdot \log \log n$ are superadditive functions. In the conjunction algorithm, we use multiway partitioning for $p = 1$ and binary partitioning for $p > 1$. Therefore $T_I(n, 1) \leq cn \log \log n$, resulting in $T_I(n, p) = O(n \log^{p-1} n \cdot \log \log n)$ overall. Non-equalities and bands are translated into inequalities by using the techniques we developed in Appendices C and D: a non-equality results into two inequalities on the same sets of nodes, while a band creates multiple inequality subproblems. We use the same arguments as in the proofs of Lemmas 20 and 22. We denote by $\mathcal{T}_I(n, p), \mathcal{T}_N(n, p), \mathcal{T}_B(n, p)$ the running time for n tuples and p predicates when the head of the list of predicates is an inequality, non-equality or band respectively. $\mathcal{T}_N(n, p) = O(\mathcal{T}_I(n, p) + \mathcal{T}_I(n, p))$ and $\mathcal{T}_B(n, p) = O(n) + 2\mathcal{T}_I(n_1, p) + 2\mathcal{T}_I(n_2, p) + \ldots + \mathcal{T}_I(n_m, p)$ for $n_1 + n_2 + \ldots + n_m \leq 3n$. By these formulas, and since $\mathcal{T}_I(n, p) = O(n \log^{p-1} n \cdot \log \log n)$, it is easy to show the same bound for the other two. This proves the space consumption of the TLFGs, thus the space bound of the theorem.

We force all paths in the constructed TLFGs to have length 3. Multiway partitioning already has that property. Whenever we use binary partitioning (that creates source-target paths of length 3), we insert artificial nodes between the intermediate layer and the target nodes. Specifically, for each edge (v_1, v_t) where $t \in T$, we introduce a new node v_2 and replace that edge with two edges $(v_1, v_2), (v_2, v_t)$. It is easy to see that the source-target paths remain the same by this modification and that we can do it time linear in the TLFG size, hence it does not affect any of our guarantees.

By the above, all intermediate nodes in our TLFGs can be assigned to two layers according to their distance from a source node. Our TLFGs are then translated to a relational representation. We introduce variables V_1, V_2, and create new relations E_1, E_2, E_3. Let $\mathbf{A}_S$, $\mathbf{A}_T$ be the attributes of S and T respectively. Relation E_1 contains attributes $\mathbf{A}_S \cup V_1$, E_2 contains $V_1 \cup V_2$, and E_3 contains $\mathbf{V}_2 \cup A_T$. Each relation contains the edges between the corresponding layers of nodes. This gives us an equi-join query Q_F. The arguments for the enumeration guarantees are the same as in the proof of Theorem 16.