ABSTRACT
We study theta-joins in general and join predicates with conjunctions and disjunctions of inequalities in particular, focusing on ranked enumeration where the answers are returned incrementally in an order dictated by a given ranking function. Our approach achieves strong time and space complexity properties: with $n$ denoting the number of tuples in the database, we guarantee for acyclic full join queries with inequality conditions that for every value of $k$, the $k$ top-ranked answers are returned in $O(n \log n + k \log k)$ time. This is within a polylogarithmic factor of $O(n + k \log k)$, i.e., the best known complexity for equi-joins, and even of $O(n + k)$, i.e., the time it takes to look at the input and return $k$ answers in any order. Our guarantees extend to join queries with selections and many types of projections (namely those called “free-connex” queries and those that use bag semantics). Remarkably, they hold even when the number of join results is $n^k$ for a join of $t$ relations. The key ingredient is a novel $O(n \log n)$-size factorized representation of the query output, which is constructed on-the-fly for a given query and database. In addition to providing the first non-trivial theoretical guarantees beyond equi-joins, we show in an experimental study that our ranked-enumeration approach is also memory-efficient and fast in practice, beating the running time of state-of-the-art database systems by orders of magnitude.

1 INTRODUCTION
Join processing is one of the most fundamental topics in database research, with recent work aiming at strong asymptotic guarantees [49, 60, 63, 64]. Work on constant-delay (unranked) enumeration [10, 19, 44, 76] 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 delay (i.e., independent of database size) between them. Together, 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. Ranked enumeration [80] generalizes the heavily studied top-$k$ paradigm [37, 47] by continuously returning join answers in ranking order. This enables the output consumer to select the cut-off $k$ on-the-fly while observing the answers. For top-$k$, the value of $k$ must be chosen in advance, before seeing any query answer. Unfortunately, non-trivial complexity guarantees of previous top-$k$ techniques, including the celebrated Threshold Algorithm [37], are limited to the “middleware” cost model, which only accounts for the number of distinct data items accessed [80]. While some of those top-$k$ algorithms can be applied to joins with general predicates, they do not provide non-trivial guarantees in the standard RAM model of computation, and their time complexity for a join of $t$ relations can be $O(n^t)$.

The goal of this paper is to design ranked-enumeration algorithms for general theta joins with strong space and time guarantees in the standard RAM model of computation. Tight upper complexity bounds are essential for ensuring predictable performance, no matter the given database instance (e.g., in terms of data skew) or the query’s total output size. Notice that it already takes $O(n + k)$ time to simply look at $n$ input tuples as well as create and return $k$ output tuples. Since polylogarithmic factors are generally considered small or even negligible for asymptotic analysis [5, 27], we aim for time bounds that are within such polylogarithmic factors of $O(n + k)$. At the same time, we want space complexity to be reasonable, e.g., for small $k$ to be within a polylogarithmic factor of $O(n)$, which is the required space to hold the input.

While state-of-the-art commercial and open-source DBMSs do not yet support ranked enumeration, it is worth taking a closer look at their implementation of top-$k$ join queries. (Here $k$ is specified in a SQL clause like FETCH FIRST or LIMIT.) While we tried a large variety of inputs, indexes on the input relations, join queries, and values of $k$, the optimizer of PostgreSQL and two other widely used commercial DBMSs always chose to execute the join before applying the ranking and top-$k$ condition on the join results. This implies that their overall time complexity to return even the top-1 result cannot be better than the worst-case join output size, which can be $O(n^k)$ for a join of $t$ relations.

Beyond equi-joins. Recent work on ranked enumeration [32, 34, 79, 19, 80] achieves much stronger worst-case guarantees, but only considers equi-joins. However, big-data analysis often also requires other join conditions [33, 36, 50, 54] such as inequalities (e.g., $S$.age $<$ T.age), non-equalities (e.g., $S$.id $\neq$ T.id), and band predicates (e.g., $|S$.time $-$ T.time $|$ $<$ $\varepsilon$). For these joins, two
major challenges must be addressed. First, the join itself must be computed efficiently in the presence of complex conditions, possibly consisting of conjunctions and disjunctions of such predicates. Second, to avoid having to produce the entire output, ranking has to be pushed deep into the join itself.

**Example 1.** A concrete application of ranked enumeration for inequality joins concerns graph-based approaches for detecting “lateral movement” between infected computers in a network [35]. By modeling computers as nodes and connections as timestamped edges, these approaches search for anomalous access patterns that take the form of paths (or more general subgraphs) ranked by the probability of occurrence according to historical data. The inequalities arise from a time constraint: the timestamps of two consecutive edges need to be in ascending order. Concretely, consider the relation $G(\text{From}, \text{To}, \text{Time}, \text{Prob})$. Valid 2-hop paths can be computed with a self-join (where $G_1$, $G_2$ are aliases of $G$) where the join condition is an equality $G_1.\text{To} = G_2.\text{From}$ and an inequality $G_1.\text{Time} < G_2.\text{Time}$, while the score of a path is $G_1.\text{Prob} \cdot G_2.\text{Prob}$. Existing approaches are severely limited computationally in terms of the length of the pattern, since the number of paths in a graph can be extremely large. Thus, they usually resort to a search over very small paths (e.g., only 2-hop). With the techniques developed in this paper, patterns of much larger size can be retrieved efficiently in ranked order without considering all possible instantiations of the pattern.

**Main contributions.** We provide the first comprehensive study on ranked enumeration for joins with conditions other than equality, notably general theta-joins and conjunctions and disjunctions of inequalities and equalities. While such joins are expensive to compute [50, 54], we show that for many of them the top-ranked answers can be computed efficiently in ranked order without loss of generality, we assume that relational symbols in different $R_i$ are aliases of $R_i$ (Section 4), we propose concrete TLFGs with space and construction-time complexity $O(n \log n + k \log k)$, using $\text{MEM}(k) = O(S(n) + k)$ space.

(2) For join conditions that are a DNF of inequalities (Section 4), we propose concrete TLFGs with space and construction-time complexity $O(n \log n)$. Using them for acyclic joins, our algorithm guarantees $TT(k) = O(n \log n + k \log k)$, which is within a polylogarithmic factor of the equi-join case, where $TT(k) = O(n + k \log k)$ [79], and even the lower bound of $O(n + k)$.

(3) Our experiments (Section 6) on synthetic and real datasets show orders-of-magnitude improvements over highly optimized top-k implementations in state-of-the-art DBMSs, as well as over an idealized competitor that is not charged for any join-related cost. Due to space constraints, formal proofs and several details of improvements to our core techniques (Section 5) are in the full version of this paper [81]. Our project website contains more information including source code: https://northeastern-datalab.github.io/anyk/.

## 2 PRELIMINARIES

### 2.1 Queries

Let $[m]$ denote the set of integers $\{1, \ldots, m\}$. A theta-join query in Datalog notation is a formula of the type

$$Q(Z) \leftarrow R_1(X_1), \ldots, R_t(X_t), \theta_1(Y_1), \ldots, \theta_q(Y_q)$$

where $R_i$ are relational symbols, $X_i$ are lists of variables (or attributes), $Z, Y_j$ are subsets of $X = \bigcup X_i$, $i \in [t]$, $j \in [q]$, and $\theta_j$ are Boolean formulas called join predicates. The terms $R_i(X_i)$ are called the atoms of the query. Equality predicates are encoded by repeat occurrences of the same variable in different atoms; all other join predicates are encoded in the corresponding $\theta_j$. If no predicates $\theta_j$ are present, then $Q$ is an equi-join. The size $|Q|$ of the query is equal to the number of symbols in the formula.

**Query semantics.** Join queries are evaluated over a database that associates with each $R_i$ a finite relation (or table) that draws values from a domain that we assume to be $\mathbb{R}$, for simplicity. Without loss of generality, we assume that relational symbols in different $R_i$ are distinct since self-joins can be handled with linear overhead by copying a relation to a new one. The maximum number of tuples in an input relation is denoted by $n$. We write $\cdot A$ for an attribute $A$ of relation $R$ and $r.A$ for the value of $A$ in tuple $r \in R$. The semantics of a theta-join query is to (i) create the Cartesian product of the $t$ relations, (ii) select the tuples that satisfy the equi-join conditions and $\theta_j$ predicates, and (iii) project on the $Z$ attributes.

---

\footnote{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.}
Consequently, each individual query answer can be represented as a combination of joining input tuples, one from each table $R_i$.

**Projections.** In this paper, we focus on full queries, i.e., join queries without projections ($Z = X$). While our approach can handle projections by applying them in the end, the strong asymptotic $TT(k)$ guarantee may not hold any more. The reason is that a projection could map multiple distinct output tuples to the same projected answer. In the strict relational model where relations are sets, those “duplicates” would have to be eliminated, creating larger gaps between consecutive answers returned to the user. Fortunately, our strong guarantees still hold for arbitrary projections in the presence of bag semantics, which is what DBMSs use when the SQL query has a SELECT clause instead of SELECT DISTINCT. Even for set semantics and SELECT DISTINCT queries, it is straightforward to extend our strong guarantees to non-full queries that are free-connex [10, 13, 17, 45].

**Join trees for equi-joins.** An equi-join query is (alpha-)acyclic [41, 77, 91] if it admits a join tree. A join tree is a tree with the atoms (relations) as the nodes where for every attribute $A$ appearing in an atom, all nodes containing $A$ form a connected subtree. The GYO reduction [91] compiles such a join tree for equi-joins.

**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-inequality is $S.A \neq T.B$ and a band is $[S.A-T.B] < \epsilon$ for some $\epsilon > 0$. Our approach also supports numerical expressions over input tuples, e.g., $f(S.A_1, S.A_2, \ldots) < g(T.B_1, T.B_2, \ldots)$, with $f$ and $g$ arbitrary $O(1)$-time computable functions that map to $\mathbb{R}$. The join predicates $\theta_j$ are built with conjunctions and disjunctions of such atomic predicates. We assume there are no predicates on individual relations since they can be removed in linear time by filtering the corresponding input tables.

### 2.2 Ranked Enumeration

**Ranked enumeration** [80] returns distinct join answers one-at-a-time, in the order dictated by a given ranking function on the output tuples. Since this paradigm generalizes top-$k$ (top-$k$ for “any $k$” value, or “anytime top-$k$”), it is also called any-$k$ [79, 88]. An obvious solution is to compute the entire join output, and then either batch-sort it or insert it into a heap data structure. Our goal is to find more efficient solutions for appropriate ranking functions.

For simplicity, in this paper we only discuss ranking by increasing sum-of-weights, where each input tuple has a real-valued weight and the weight of an output tuple is the sum of the weights of the input tuples that were joined to derive it. Ranked enumeration returns the join answers in increasing order of output-tuple weight. It is straightforward to generalize our approach to any ranking function that can be interpreted as a *selective dioid* [79]. Intuitively, a selective dioid [39] is a semiring that also establishes a total order on the domain. It has two operators (min and + for sum-of-weights) where one *distributes* over the other (+ distributes over min). These structures include even less obvious cases such as lexicographic ordering by relation attributes.

### 2.3 Complexity Measures

We consider in-memory computation and analyze all algorithms in the standard Random Access Machine (RAM) model with uniform cost measure. Following common practice, we treat query size $|Q|$—intuitively, the length of the SQL string—as a constant. This corresponds to the classic notion of data complexity [82], where one is interested in scalability in the size of the input data, and not of the query (because users do not write arbitrarily large queries).

In line with previous work [15, 22, 40], 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.

For the time complexity of enumeration algorithms, we measure the time until the $k^\text{th}$ result is returned ($TT(k)$) for all values of $k$. In the full version [81], we further discuss the relationship of $TT(k)$ to enumeration delay as complexity measures. Since we do not assume any given indexes, a trivial lower bound is $TT(k) = O(n + k)$: the time to inspect each input tuple at least once and to return $k$ output tuples. Our algorithms achieve that lower bound up to a polylogarithmic factor. For space complexity, we use $MEM(k)$ to denote the required memory until the $k^\text{th}$ result is returned.

### 3 GRAPH FRAMEWORK FOR JOINS

We summarize our recent work on ranked enumeration for equi-joins, then show our novel generalization to theta-joins.

#### 3.1 Previous Work: Any-$k$ for Equi-joins

Any-$k$ algorithms [79] for acyclic equi-joins reduce ranked enumeration to the problem of finding the $k^\text{th}$-lightest trees in a layered DAG, which we call the *enumeration graph*. Its structure depends on the join tree of the given query; an example is depicted in Fig. 2a. The enumeration graph is a layered DAG in the sense that we associate it with a particular topological sort: (1) Conceptually, each node is labeled with a layer ID (not shown in the figure to avoid clutter). A layer is a set of nodes that share the same layer ID (depicted with rounded rectangles). (2) Each edge is directed, going from lower to higher layer ID. (3) All tuples from an input relation appear as (black-shaded) nodes in the same layer, called a *relation layer*. Each relation layer has a unique ID and for each join-tree edge $(S, T)$, $S$ has a lower layer ID than $T$.

(4) If only if only two relations are adjacent in the join tree, then their layers are connected via a *connection layer* that contains (blue-shaded) nodes representing their join-attribute values. (5) The edges from a relation layer to a connection layer connect the tuples with their corresponding join-attribute values and vice-versa.

The enumeration graph is constructed on-the-fly and bottom-up, according to a join tree of the query (starting from $U$ and $T$ in the example). This phase essentially performs a bottom-up semi-join reduction that also creates the edges and join-attribute-value nodes. A tree solution is a tree that starts from the root layer and contains exactly 1 node from each relation layer. By construction, every tree solution corresponds to a query answer, and vice versa.

The any-$k$ algorithm then goes through two phases on the enumeration graph. The first is a Dynamic Programming computation, where every graph node records for each of its outgoing edges the lowest weight among all subtrees that contain 1 node from each relation layer below. The minimum-subtree and input-tuple weights
are not shown in Figure 2a to avoid clutter. For instance, the outgoing edge for R-node (2, 3) would store the smaller of the weights of U-tuples (2, 1) and (2, 2). Similarly, the left edge from S-node (2, 1) would store the sum of the weight of R-tuple (2, 3) and the minimum subtree weight from R-node (2, 3). The minimum-subtree weight for a node’s outgoing edge is obtained at a constant cost by pushing the minimum weight over all outgoing edges up to the node’s parent. Afterwards, enumeration is done in a second phase, where the enumeration graph is traversed top-down (from S in the example), with the traversal order determined by the layer IDs and minimum-subtree weights on a node’s outgoing edges.

The size of the enumeration graph and its number of layers determine space and time complexity of the any-κ algorithm. The following lemma summarizes the main result from our previous work [79]. We restate it here in terms of data complexity (where query size ℓ is a constant) and using λ for the number of layers.3

**Lemma 2 ([79]).** Given an enumeration graph with |E| edges and λ layers, ranked enumeration of the k-lightest tree solutions can be performed with T(k) = O(|E| + k log k + kl) and MEM(k) = O(|E| + kl).

To extend the any-κ framework beyond equi-joins, we generalize first the definition of a join tree and then the enumeration graph with an abstraction that is sensitive to the join conditions.

### 3.2 Theta-Join Tree

The join tree is essential for generating the enumeration graph. In contrast to equi-joins, for general join conditions there is no established methodology for how to define or find a join tree. We generalize the join tree definition as follows:

**Definition 3 (Theta-Join Tree).** A theta-join tree for a theta-join query Q is a join tree for the equi-join Q′ that has all the θj predicates of Q removed, and every θj is assigned to an edge (S, T) of the tree such that S and T contain all the attributes referenced in θj.

We call a theta-join query acyclic if it admits a theta-join tree. In the theta-join tree, edge (S, T) represents the join S θ T, where join condition θ is the conjunction of all predicates θj assigned to the edge, as well as the equality predicates S.A = T.A for every attribute A that appears in both S and T.

**Example 4.** Consider Q(A, B, C, D, E, F) = R(D, E), S(A, D), T(B, C), U(D, F), (A < B), (A > E), (E < F).a This query is acyclic since we can construct the theta-join tree shown in Fig. 2b. Notice that all nodes containing attribute D are connected and each inequality is assigned to an edge whose adjacent nodes together contain all referenced attributes. For example, A < B is assigned to (S, T) (S contains A and T contains B). The join-tree edges represent join predicates θ1 = S.A < T.B (edge (S, T)), θ2 = S.A > T.E (edge (S, R)), and θ3 = R.E < U.D (edge (R, U)).

We can construct the theta-join tree by first removing all θj predicates from the given query Q, turning it into an equi-join Q′. Then an algorithm like the GYO reduction can be used to find a join tree for Q’. For the query in Example 4, this join tree looks like the one in Figure 2b, but without the edge labels. Finally, we attempt to add each θj predicate to a join-tree edge: θj can be assigned to any edge where the two adjacent nodes contain all the attributes referenced in it. Note that there may exist different join trees for Q’, and we may have to try all possible options to obtain a theta-join tree. Fortunately, this computation depends only on the query, thus takes O(1) space and time in data complexity. If either the GYO algorithm fails to find a join tree for Q’ or no join tree allows us to assign the θj predicates to tree edges, then the query is cyclic and can be handled as discussed in Section 3.3. We discuss next how to create the enumeration graph for a given theta-join tree.

### 3.3 Factorized Join Representation

By relying on a join tree similar in structure to the equi-join case, we can establish a similar layered structure for the enumeration graph. In particular, each input relation appears in a separate layer
and each join-tree edge is mapped to a subgraph implementing the join condition between the corresponding relation layers. This is visualized by the blue clouds in Figure 2b. In contrast to the equi-joins, we allow more general connection layers, possibly a single layer with a more complex connection pattern (like the S-to-T connection in the example) or even multiple layers (like the connection between R-node (2, 1) and U-node (2, 2)).

To be able to apply our any-k algorithms to this generalized enumeration graph we must ensure that (1) each “blue cloud” can be mapped to a layered graph and (2) each tree solution corresponds to a join answer, and vice versa (like the one highlighted in Figure 2b which corresponds to joining input tuples \( s = (3, 2), t = (4, 3), r = (2, 1), \) and \( u = (2, 2) \)). For (2) it is sufficient to ensure for each adjacent parent-child pair of relations in the theta-join tree that there exists a path from a node in the parent-relation layer to a node in the child-relation layer i.e. the corresponding input tuples join. In the example, there is a path from S-node (3, 2) via \( v_s \) to T-node (4, 3), because the two tuples satisfy \( A = 3 < B = 4 \). Similarly, since \( s = (5, 3) \) and \( t = (4, 3) \) violate \( A < B \), there is no path from the former to the latter. For (1), it is sufficient to ensure that the “blue cloud” is a DAG with parent-relation nodes only having edges going into the cloud, while all child-relation edges must point out of the cloud. We formalize these properties with the notion of a Tuple-Level Factorization Graph (TLFG).

**Definition 5 (TLFG).** A Tuple-Level Factorization Graph of a theta-join \( S \bowtie T \) of relation \( S \), called the source, and \( T \), called the target, is a directed acyclic graph \( G(V, E) \) where:

1. \( V \) contains a distinct source node \( v_s \) for each tuple \( s \in S \), a distinct target node \( v_t \) for each tuple \( t \in T \), and possibly other intermediate nodes,
2. each source node \( v_s \) has only outgoing edges and each target node \( v_t \) has only incoming edges, and
3. 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 \( \theta \).

The size of a TLFG \( G(V, E) \) is \( |V| + |E| \) and its depth \( d \) is the maximum length of any path in \( G \). The graphs depicted in Fig. 4a and Fig. 4b are valid TLFGs for equi-joins.

It is easy to see that any TLFG is a layered graph: Assign w.l.o.g. layer ID 0 to all source nodes \( v_s \); each intermediate node \( v \) is assigned layer ID \( i \), where \( i \) is the length of the longest path (measured in number of edges) from any source node to \( v \). Here \( i \) is well-defined due to the TLFG’s acyclicity. All target-relation nodes are assigned to layer \( d \), which is the maximum layer ID assigned to any intermediate node, plus 1. In the example in Figure 4d, node \( v_3 \) is in layer 3, because the longest path from any S-node to \( v_3 \) has 3 edges (from (1, 1) in the example). All T-nodes are in layer 6.

Since the entire generalized enumeration graph consists of \( t \) relation layers and \( t - 1 \) TLFGs (one for each edge of the theta-join tree), using **Lemma 2** we can show:

**Theorem 6.** Given a theta-join \( Q \) of \( t = O(1) \) relations, a theta-join tree, and the corresponding enumeration graph \( G_Q \), where for each edge of the theta-join tree the corresponding TLFG has \( O(|E|) \) size and \( O(d) \) depth, then ranked enumeration of the \( k \)-lightest tree solutions can be performed with \( TT(k) = O(|E| + k \log k + kd) \) and \( MEM(k) = O(|E| + kd) \).

The theorem states that worst-case size and depth of the TLFG determine the time and space complexity of enumerating the theta-join answers in weight order. Hence the main challenge is to encode join condition with the smallest and most shallow TLFG possible.

**Direct TLFGs.** For any theta-join, a naive way to construct a TLFG is to directly connect each source node with all the target nodes it joins with. This results in \( |E| = O(n^2) \) and \( d = 1 \), thus \( TT(k) = O(n^2 + k \log k) \) and \( MEM(k) = O(n^2 + k) \), respectively. Hence even the top-ranked result requires quadratic time and space. To improve this complexity, we must find a TLFG with a smaller number of edges, while keeping the depth low. Our results are summarized in the and, with details discussed in later sections.

**Output duplicates.** A subtle issue with **Theorem 6** is that two non-isomorphic tree solutions of the enumeration graph may contain the exact same input tuples (the relation-layer nodes), causing duplicate query answers. This happens if and only if a TLFG has multiple paths between the same source and destination node. While one would like to avoid this, it may not be possible to find a TLFG that is both efficient in terms of size and depth, and also free of duplicate paths. Among the inequality conditions studied in this paper, this only happens for disjunctions (Section 4.3).

Since duplicate join answers must be removed, the time to return the \( k \) top-ranked answers may increase. Fortunately, for our disjunction construction it is easy to show that the number of duplicates per output tuple is \( O(1) \), i.e., it does not depend on input size \( n \). This implies that we can filter the duplicates on-the-fly without increasing the complexity of \( TT(k) \) (or \( MEM(k) \), for that matter): We maintain the top-k join answers returned so far in a lookup structure and, before outputting the next join answer, we check in \( O(1) \) time if the same output had been returned before.

To prove that the number of duplicates per join answer is independent of input size, it is sufficient to show that for each TLFG the maximum number of paths from any source node \( v_s \) to any target node \( v_t \), which we will call the **duplication factor**, is independent of input size. We show this to be the case for the only TLFG construction that could introduce duplicate paths: disjunctions (Section 4.3). A duplicate-free TLFG has a duplication factor equal to 1 (which is the case for most TLFGs we discuss).

---

*As an optimization, we can clear this lookup structure whenever the weight of an answer is greater than the previous, since all duplicates share the same weight. While this does not impact worst-case complexity, it can greatly reduce computation cost in practice whenever output tuples have diverse sum-of-weight values.*
3.4 Theta-join to Equi-join Reduction

The factorized representation of the output of a theta-join as an enumeration formula (using TLFGs to connect adjacent relation layers) enables a novel reduction from complex theta-joins to equi-joins.

**Theorem 7.** Let $G = (V, E)$ be a TLFG of depth $d$ for a theta-join $S \bowtie T$ of relations $S$, $T$ and $X$ be the union of their attributes. For $0 < i \leq d$, let $E_i$ be the set of edges from layer $i - 1$ to $i$. If $E = \bigcup_i E_i$, i.e., every edge connects nodes in adjacent layers, then $S \bowtie T = \pi_X(S \bowtie E_1 \bowtie \cdots \bowtie E_d \bowtie T)$ where $\pi_X$ is an $X$-projection.

Intuitively, the theorem states that if no edge in the TLFG skips a layer, then the theta-join $S \bowtie T$ can equivalently be computed as an equi-join between $S$, $T$, and $d$ auxiliary relations. Each of those relations is the set of edges between adjacent layers of the TLFG.

The theorem is easy to prove by construction, which we explain using the example in Figure 2b. Consider the TLFG for $S$ and $T$ and notice that all edges are between adjacent layers and $d = 2$. In Figure 2c, the first tuple $(1, 1, v_1) \in E_1$ represents the edge from $S$-node $(1, 1)$ to intermediate node $v_1$. (The tuple is obtained as the Cartesian product of the edge’s endpoints.) Similarly, the first tuple in $E_2$ represents the edge from $v_1$ to $T$-node $(2, 1)$. It is easy to verify that $S(A, D) \bowtie_{A \leq B} T(B, C) = \pi_{ADBC}(S \bowtie E_1 \bowtie E_2 \bowtie T)$. The corresponding branch of the join tree is shown in Figure 2c. Compared to the theta-join tree in Figure 2b, the inequality condition disappeared from the edge and is replaced by new nodes $E_1(A, D, V_1)$ and $E_2(V_1, B, C)$.

**QuadEqui** for direct TLFGs. Recall that any theta-join $S \bowtie T$ between relations of size $O(n)$ can be represented by a 1-layer TLFG that directly connects the joining $S$- and $T$-nodes. Since this TLFG satisfies the condition of Theorem 7, it can be reduced to equi-join $S \bowtie T$, where $|E| = O(n^2)$. We refer to the algorithm that first applies this construction to each edge of the theta-join tree (and thus reducing the entire theta-join query between $f$ relations to an equi-join) and then uses the equi-join ranked-enumeration algorithm [79] as QuadEqui.

Below we will show that better constructions with smaller auxiliary relations $E_i$ can be found for any join condition that is a DNF of inequalities. In particular, such joins can be expressed as $S \bowtie E_1 \bowtie E_2 \bowtie T$ where $E_1, E_2$ are of size $O(n \log n)$. Figure 2c shows a concrete instance. However, note that not all TLFGs satisfy the condition of Theorem 7. For example, Fig. 4d shows a TLFG which cannot be reduced to an equi-join with our theorem.

4 FACTORIZATION OF INEQUALITIES

We now show how to construct TLFGs of size $O(n \log n)$ and depth $O(1)$ when the join condition $\theta$ in a join $S \bowtie T$ is a DNF of inequalities (and equalities). Starting with a single inequality, we then generalize to conjunctions and finally to DNF. Non-equalities and bands will be discussed in Section 5.

4.1 Single Inequality Condition

Efficient TLFGs for equi-joins exploit that equality conditions group input tuples into disjoint equivalence classes (Fig. 4b). For inequalities, this is generally not possible and therefore we need a different approach to leverage their structural properties (see Fig. 4c).

**Binary partitioning.** Our binary-partitioning based TLFG is inspired by quicksort [42]. Consider condition $S.A < T.B$ and a pivot value $v$. We partition 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 $t \in T_2$. This guarantees that all $A$-values in $S_1$ are strictly less than all $B$-values in $T_2$. Instead of representing this with $|S_1| \cdot |T_2|$ direct edges ($s_1 \in S_1, t_j \in T_2$), we introduce an intermediate “pivot node” $v$ and use only $|S_1| + |T_2|$ edges ($s_1 \in S_1, v$) and ($v, t_j \in T_2$).

Then we continue recursively with the remaining partition pairs $(S_1, T_1)$ and $(S_2, T_2)$. (Note that $(S_2, T_1)$ cannot contain joining tuples by construction.) Each recursive step will create a new intermediate node connecting a set of source and target nodes, therefore the TLFG has depth 2.

As the pivot, we use the median of the distinct join-attribute values appearing in the tuples in both input partitions. E.g., 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 $O(n)$ time if the relations have been sorted on the join attributes beforehand. Since each partition step cuts the number of distinct values per partition in half, it takes $O(\log n)$ steps until we reach the base case where all input tuples in a partition share the same join-attribute value and the recursion terminates. Overall, the algorithm takes time $O(n \log n)$ to construct a TLFG of size $O(n \log n)$ and depth 2. It is easy to see that there is exactly one path from each source to joining target node, hence the TLFG is _duplicate-free_.

**Example 8.** Figure 4e illustrates the approach, with dotted lines showing how the relations are partitioned. Initially, we create partitions containing the values $\{1, 2, 3\}$ and $\{4, 5, 6\}$ respectively. The source nodes containing $A$-values of the first partition are connected to target nodes containing $B$-values 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.** The construction for greater-than ($>$) is symmetric, connecting $S_2$ to $T_1$ instead of $S_1$ to $T_2$. For $\leq$ and $\geq$, we only need to modify handling of the base case of the recursion: instead of simply returning from the last call (when all tuples in a partition have the same join-attribute value), the algorithm connects the corresponding source and target nodes via an intermediate node (like for equality predicates).

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

4.2 Conjunctions

TLFG construction for conjunctions can be integrated elegantly into the recursive binary partitioning.
Figure 4: 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.

**Figure 5: Example 10:** 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).

**Example 10.** Consider join condition $S.A < T.C \land S.B > T.D$ for relations $S(A, B), T(C, D)$ as shown in Fig. 5a. The algorithm initially considers the first inequality $S.A < T.C$, splitting the relations into $S_1, T_1, S_2, T_2$ as per the binary partitioning method (see Section 4.1). All pairs $(s_i \in S_1, t_j \in T_2)$ satisfy $S.A < T.C$, but not all of them satisfy the other conjunct $S.B > T.D$. To correctly connect the source and target nodes, we therefore run the same binary partitioning algorithm on input partitions $S_1$ and $T_2$, but now with predicate $S.B > T.D$ as illustrated by the diagonal blue edge in Fig. 5a; the resulting graph structure is shown in Fig. 5b. For the remaining partition pairs ($S_1, T_1$) and ($S_2, T_2$), the recursive call still needs to enforce both conjuncts as illustrated by the orange edges in Fig. 5a.

**Strict inequalities.** The example generalizes in a straightforward way to the conjunction of any number of strict inequalities as shown in Algorithm 1. We note that the order in which the predicates are handled does not impact the asymptotic analysis, but in practice, handling the most selective predicates first is bound to give better performance. Whenever two partitions are guaranteed to satisfy a conjunct, that conjunct is removed from consideration in the next recursive call (Line 19). An intermediate node for the pivot and the corresponding edges connecting it to source and target nodes are only added to the TLFG when no predicates remain (Lines 14 to 16). 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 21 and 22). In the other direction, when the current predicate is satisfied for a partition pair, nextPredicate() is called with one less predicate (Line 19). 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 time a new predicate is processed by a recursive call, the join-attribute values in the corresponding partitions are sorted according to the new attributes (Line 6) to find the pivot.

**Non-strict inequalities.** Like for a single predicate, we only need to modify handling of the base case when all join-attribute values in a partition are the same. While a strict inequality is not satisfied and thus no edges are added to the TLFG, the non-strict one is satisfied for all pairs of source and target nodes in the partition. Hence instead of exiting the recursive call (Line 10), the partition pair is treated like the $(S_1, T_2)$ case (Lines 14 to 19).

**Equalities.** If the conjunction contains both equality and inequality predicates, then we reduce the problem to an inequality-only conjunction by first partitioning the inputs into equivalence classes according to all equality predicates (see Fig. 4b). Then the inequality-only algorithm introduced above is executed on each of these partitions. Since the equality-based partitioning takes linear time and space, complexity is determined by the inequality predicates.

**Lemma 11.** Let $\theta$ 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

Given a join condition that can be expressed as a disjunction $P = \lor_i P_i$ where $G_i$ is the TLFG for $P_i$, we construct the TLFG $G$ for $P$
We propose improvements that lead to our main result: strong worst-case guarantees for binary partitioning and depth 3 (vs. 2). Unfortunately, it is unclear how to generalize this idea to a conjunction of inequalities.

Shared ranges. A simple inequality can be encoded even more compactly with $O(n)$ edges by exploiting the transitivity of “<” as illustrated in Figure 4d. Intuitively, our shared ranges 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. The resulting TLFG has size and depth $O(n)$. The latter causes a high delay between consecutive join answers. From Theorem 6 and the fact that we need to sort to construct the TLFG, we obtain $TT(k) = O(n log n + n + k log k + kn) = O(n log n + kn)$ and $MEM(k) = O(n + kn) = O(kn)$. Compared to binary partitioning’s $O(n log n + k log k)$ and $O(n log n + k)$ (Theorem 6, Lemma 9), respectively, space complexity is reduced by a factor $log n$, and without affecting time complexity, only for small $k$, i.e., $k = o(log n)$. For larger $k = O(n)$ both space and time complexity are worse by (almost) a factor $n$. (Recall that $k = O(n^)$ for a join of $t$ relations.) Moreover, like for multiway partitioning, it is not clear how to generalize this construction to conjunctions of inequalities.

Non-Equality and Band Predicates. A non-equality predicate can be expressed as a conjunction of $2$ inequalities; a band predicate as a conjunction of $2$ inequalities. Hence both can be handled by the techniques discussed in Section 4, at the cost of increasing query size by up to a constant factor. This can be avoided by a specialized construction that leverages the structure of these predicates. It is similar to the binary partitioning for an inequality (and hence omitted due to space constraints) and achieves the same size and depth guarantees for the TLFG.

5.2 Putting Everything Together

Using multiway partitioning and the specialized techniques for non-equality and band predicates yields:

**Lemma 13.** Let $\theta$ be a simple inequality, non-equality, or band predicate for relations $S, T$, of size $O(n)$. A duplicate-free TLFG for $S \bowtie T$ of size $O(n log log n)$ and depth 3 can be constructed in $O(n log n)$ time.

Applying the approach for a DNF of inequalities (Section 4), but using the specialized TLFGs for non-equality and band predicates and multiway partitioning for the base case of the conjunction construction (when only one predicate remains), we obtain:

**Theorem 14 (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, and band predicates. Let $p$ be the maximum number of predicates, excluding equalities, in a conjunction of a DNF on any edge of the theta-join tree. Ranked enumeration of the answers to $Q$ over $D$ can be performed with $TT(k) = O(n log^p n + k log k)$. The space requirement is $MEM(k) = O(n log^{p-1} n \cdot log log n + k)$.

8
5.3 Cyclic Queries

So far, we have focused only on cyclic queries, but our techniques are also applicable to cyclic queries with some modifications. Recall that cyclic queries admit a theta-join tree, which is found by assigning predicates to the edges of a join tree. If this procedure fails, we can handle the query as follows:

Post-processing filter. An common practical solution for cyclic queries is to ignore some predicates during join processing, then apply them as a filter on the output. Specifically, we can remove \( \theta_j \) predicates and equality conditions encoded by the same variable names until the query admits a theta-join tree, then apply our technique to the resulting acyclic query, and finally use the removed predicates as a filter. While this approach is simple to implement, it can suffer from large intermediate results. In the worst case, all answers to the acyclic join except the last one may be discarded, giving us \( \mathrm{TT}(k) = O(n^e \log n) \) for an \( e \)-relation cyclic join.

Transformation to equi-join. An alternative approach with non-trivial guarantees is to apply our equi-join transformation to the cyclic query, and then use existing algorithms for ranked enumeration of cyclic equi-joins [79]. We deal with the case where each \( \theta_j \) predicate is covered by at most 2 input relations; the general case is left for future work. To handle that case, we add edges to the join tree as needed (creating a cyclic \( \text{theta-join graph} \)) and assign predicates to covering edges. To achieve the equi-join transformation, we consider all pairs of connected relations in the join graph, build a TLFG according to the join condition, and then materialize relations “in the middle” as illustrated in Section 3.4. The resulting query contains only equality predicates, hence is a cyclic equi-join. Ranked enumeration for cyclic equi-joins is possible with guarantees that depend on a width measure of the query [79].

Example 15 (Inequality Cycle). The following triangle query variant joins three relations with inequalities in a cyclic way:

\[
Q(A, B, C, D, E, F) := \langle R(A, B), S(C, D), T(E, F), (B < C), (D < E), (F < A) \rangle.
\]

Notice that there is no way to organize the relations in a tree with the inequalities over parent-child pairs. However, if we remove the last inequality \( (F < A) \), the query becomes acyclic and a generalized join tree can be constructed. Thus, we can apply our techniques on that query and filter the answers with the selection condition \( (F < A) \).

Alternatively, we can factorize the pairs of relations using our TLFGs, to obtain a cyclic equi-join. If we use binary partitioning, this introduces three new attributes \( V_1, V_2, V_3 \) and six new \( O(n \log n) \)-size relations:

\[
E_1(A, B, V_1), E_2(V_1, C, D), E_3(C, D, V_2), E_4(V_2, E, F), E_5(E, F, V_3), E_6(V_3, A, B).
\]

The transformed query can be shown to have a submodular width [5, 58] of \( 5/3 \), making ranked enumeration possible with \( \mathrm{TT}(k) = O((n \log n)^{5/3} + k \log k) \).

6 EXPERIMENTS

We demonstrate the superiority of our approach for ranked enumeration against existing DBMSs, and even idealized competitors that receive the join output “for free” as an (unordered) array.

Algorithms. We compare 5 algorithms: (1) FACTORIZED is our proposed approach. (2) QUADEQUI is an idealized version of the fairly straightforward reduction to equi-joins described in Section 3.4, which for each edge \((S, T)\) of the theta-join tree uses the direct TLFG (no intermediate nodes) to convert \( S \bowtie E \bowtie T \) to equi-join \( S \bowtie E \bowtie T \) via the edge set \( E \) of the TLFG. Then previous ranked-enumeration techniques for equi-joins [79] can be applied directly. To avoid any concerns regarding the choice of technique for generating \( E \), we provide it “for free.” Hence the algorithm is not charged for essentially executing theta-joins between all pairs of adjacent relations in the theta-join tree, meaning the QUADEQUI numbers reported here represent a lower bound of real-world running time. (3) BATCH is an idealized version of the approach taken by state-of-the-art DBMSs. It computes the entire join output and puts it into a heap for ranked enumeration. To avoid concerns about the most efficient join implementation, we give BATCH the entire join output “for free” as an in-memory array. It simply needs to read those output tuples (instead of having to execute the actual join) to rank them, therefore the numbers reported constitute a lower bound of real-world running time. We note that for a join of only \( f = 2 \) relations, there is no difference between QUADEQUI and BATCH since they both receive all the query results; we thus omit QUADEQUI for binary joins. (4) PSQL is the open-source PostgreSQL system. (5) SYSTEM X is a commercial database system that is highly optimized for in-memory computation.

We also compare our factorization methods (1a) Binary Partitioning, (1b) Multiway Partitioning, and (1c) Shared Ranges against each other. Recall that the latter two can only be applied to single-inequality type join conditions. Unless specified otherwise, FACTORIZED is set to (1b) Multiway Partitioning for the single-predicate cases and (1a) Binary Partitioning for all others.

Data. Our synthetic data generator creates relations \( S_i(A_i, A_{i+1}, W_i), i \geq 1 \) by drawing \( A_i, A_{i+1} \) from integers in \([0, 10^8]\). We also use the LINEITEM relation of the TPC-H benchmark [2], keeping the schema Item(OrderKey, PartKey, Suppkey, LineNumber, Quantity, Price, ShipDate, CommitDate, ReceiptDate).

For real data, we use a temporal graph \( \text{REDDIT T ITLES} \) [53] whose 286k edges represent 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). (1b) OCEANIA BIRDS [1] reports bird observations from Oceania with schema Birds(ID, Latitude, Longitude, Count). We keep only the 452k observations with a non-empty Count attribute.

Queries. We test queries with various join conditions and sizes. Figure 6 gives the Datalog notation and the ranking function. Some of the queries have the number of relations \( f \) as a parameter; for those we only write the join conditions between the \( j \)th and \((j + 1)\)th relations, with the rest similarly organized in a chain. In the full version [81] we give the equivalent SQL queries.

On our synthetic data, \( Q_{S1} \) is a single inequality join, while \( Q_{S2} \) has a more complicated join condition that is a conjunction of a band and a non-equality. On TPC-H, \( Q_T \) finds a sequence of items sold by the same supplier with the quantity increasing over time, ranked by the price. To test disjunctions, we run query \( Q_{TD} \), which puts the increasing time constraint on either of the three possible
6.1 Comparison Against Alternatives

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** 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 **Batch** require significantly more memory and are infeasible for many queries. (3) **PSQL** and **System X**, similarly to **Batch**, must produce the entire output, which is very costly. While **System X** is clearly faster than **PSQL**, it can be several orders of magnitude slower than our **Factorized**, and is outperformed across all tested queries.

6.1.1 Effect of Data Size. We run queries \( Q_{R1}, Q_{R2} \) for different input sizes \( n \) and two distinct query lengths. Figure 7 depicts the time to return the top \( k = 10^3 \) results. We also plot how the size of the output grows with increasing \( n \) on a secondary y-axis. Even though **QuadEqui** and **Batch** are given precomputed join results for free 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 7b). **PSQL** does not face a memory problem because it can resort to secondary storage, yet becomes unacceptably slow. The in-memory optimized **System X** is 10 times faster than **PSQL**, but still follows the same trend because it is materializing the entire output. In contrast, our **Factorized** approach scales smoothly across all tests and requires much less memory. For instance, in Figure 7b **QuadEqui** fails after 8k input size, while we can easily handle 2M. For very small input, the idealized methods **QuadEqui** and **Batch** are sometimes faster, but their real running time would be much higher if join computation was accounted for. **Q2s** has more join predicates and thus smaller output size (Figures 7c and 7d). Our advantage is smaller in this case, yet still significant for large \( n \).

We similarly run queries \( Q_T \) (Figure 8a) and \( Q_{TD} \) (Figure 8b) for \( \ell = 3 \) with an increasing scale factor (which determines data size). Here, the equi-join condition on the supplier severely limits the blowup of the output compared to the input. Still, **Factorized** is again superior. Disjunctions in \( Q_{TD} \) increase the running time of our technique only slightly by a small constant factor.

6.1.2 Effect of Query Length. Next, we test the effect of query length on **RedditTitles**. We plot \( TT(k) \) for three values \( k = 1, 10^3, 10^5 \) when the length is small \( (\ell = 2, 3) \) and one value \( k = 10^5 \) for longer queries. Note that for \( k = 1 \), the time of **Factorized** is essentially the time required for building our LGTFs, and doing a bottom-up Dynamic Programming pass [79]. Figure 9 depicts our results for queries \( Q_{R1}, Q_{R2} \). Increasing the value of \( k \) does not have a serious impact for most of the approaches except for **System X**, which for \( k = 10^6 \) is not able to provide the same optimized execution. For binary-join \( Q_{R1} \), our **Factorized** is faster than the **Batch** lower bound (Figure 9a), and its advantage increases for longer queries, since the output also grows (Figure 9c). **Batch** runs out of memory for \( \ell = 3 \), while **QuadEqui** and **System X** are more than 100 times slower (Figure 9b). Query \( Q_{R2} \) has an additional join predicate, hence its output size is smaller. Thus, the **Batch** lower bound is slightly better than our approach for \( \ell = 2 \) (Figure 9e), but we expect it to be significantly slower if the cost of computing and materializing the output was taken into account. Either way, for \( \ell \geq 3 \) (Figure 9g), our approach dominates even when compared against the lower bounds. **PSQL** again times out for \( \ell = 3 \) (Figure 9i), and the highly optimized **System X** is outclassed by our approach.

6.1.3 Effect of Band Parameter. We now test the band-join \( Q_B \) on the **OceanaiBirds** dataset with various band widths \( \ell \). Figure 9d
shows that \textsc{Factorized} is superior for all tested \( k \) values for \( \epsilon = 0.01 \). Increasing the band width yields more joining pairs and causes the size of the output to grow (Figure 9b). Hence, Batch consumes more memory and cannot handle \( \epsilon \geq 0.16 \). On the other hand, the performance of \textsc{Factorized} is mildly affected by increasing \( \epsilon \). \textsc{PSQL} and System X were not able to terminate within the time limit even for the smallest \( \epsilon \) because they use only one of the indexes (for Longitude), searching over a huge number of possible results.

### 6.2 Comparison of our Variants

We now compare our 3 factorization methods 1(a), 1(b), 1(c).

#### 6.2.1 Delay and \( \text{T}(k) \)

Since only Binary Partitioning is applicable to all types of join conditions considered, we compare the different methods on \( Q_{S1} \), which has only one inequality-type predicate. Figure 10a depicts \( \text{T}(k) \) for \( k = 1, 10^4, 2 \cdot 10^4, 3 \cdot 10^4 \). Even though Shared Ranges starts returning results faster because its TLFG is constructed in a single pass (after sorting), it suffers from a high enumeration delay (linear in data size), and quickly deteriorates as \( k \) increases. The delay is also depicted in Figure 10b, where we observe that Binary Partitioning returns results with lower delay than Multway Partitioning (recall that Multway Partitioning has a depth of 3 vs Binary Partitioning’s 2). These results are a consequence of the size-depth tradeoff of the TLFGs (Fig. 3). Note that the higher delay observed in the beginning is due to lazy initialization of data structures needed by the any-\( k \) algorithm.

#### 6.2.2 Join Representation

We show the sizes of the constructed representation in Figure 10c, using an implementation-agnostic measure. As \( n \) increases there is an asymptotic difference between the three methods \( \mathcal{O}(n \log n) \) vs \( \mathcal{O}(n \log \log n) \) vs \( \mathcal{O}(n) \) that manifests in our experiment. To see how the presence of the same domain values could affect the construction of the TLFG, we also measure the time to the first result for different domain sizes (Figure 10d). All three of our methods become faster when the domain is small and multiple occurrences of the same value are more likely. This is expected since the intermediate nodes of our TLFG essentially represent ranges in the domain and they are more compact for smaller domains. Domain size does not significantly impact running time once it exceeds sample size (around \( n = 2^{10} \)) and the probability of sampling duplicate domain values approaches zero.

## 7 RELATED WORK

### Enumeration for equi-joins

\textit{Unranked} enumeration for equi-joins has been explored in various contexts \cite{13, 14, 19, 20, 35, 76}, with a landmark result showing for self-join-free equi-joins that linear preprocessing and constant delay are possible if and only if the query is free-connex acyclic \cite{10, 16}. For the more demanding task of ranked enumeration, a logarithmic delay is unavoidable \cite{18, 32}. Our recently proposed any-\( k \)-algorithms represent the state of the art for ranked enumeration for equi-joins \cite{79}. Other work in this space focuses on practical implementations \cite{34} and direct access \cite{21, 22} to output tuples.

\textbf{Non-Equality (\#) and inequality (\langle \) joins.} Techniques for batch-computation of the entire output for joins with \textit{non-equality} (also called \textit{inequality} \cite{51} or \textit{disequality} \cite{10}) predicates mainly rely on variations of color coding \cite{8, 51, 73}. The same core idea is leveraged by the unranked enumeration algorithm of Bagian et al. \cite{10}. Queries with negation can be answered by rewriting them with \textit{not-all-equal-predicates} \cite{48}, a generalization of non-equality.

Khayatt et al. \cite{30} provide optimized and distributed \textit{batch} algorithms for up to two inequalities per join. Aggregate computation \cite{3} and Unranked enumeration under updates \cite{45} have been studied for inequality predicates by using appropriate index structures.

We are the first to consider \textit{ranked} enumeration for non-equality and inequality predicates, including DNF conditions containing both types, and to prove strong worst-case guarantees for a large class of these queries.

### Orthogonal range search.

Our binary partitioning method shares a similar intuition with index structures that have been devised for orthogonal range search \cite{6, 25}. For unranked enumeration, it has been shown \cite{7, 84, 85} how, for two relations, a range tree \cite{31} can be used to identify pairs of matching tuple sets. This gives an alternative method to construct our depth-2 TLFGs because a pair of matching tuple sets can be connected via one intermediate node. Our approach supports ranking and it is simpler since it does
(a) Query $Q_{R1}$, $\ell = 2$. (b) Query $Q_{R1}$, $\ell = 3$. (c) Query $Q_{R1}$, different lengths $\ell$. (d) Query $Q_{R2}$, fixed $\varepsilon = 0.01$. (e) Query $Q_{R2}$, $\ell = 2$. (f) Query $Q_{R2}$, $\ell = 3$. (g) Query $Q_{R2}$, different lengths $\ell$. (h) Query $Q_{R3}$, different bands $\ell$. Figure 9: Section 6.1.2: a,b,c,e,f,g: Section 6.1.3: d, h: Temporal paths of different lengths on REDDIT/TITLES (left), and spatial band-join on OCEANIABIRDS (right). Our method is robust to increasing query sizes and band-join ranges.

Figure 10: Section 6.2: Comparing different aspects of our factorization methods on query $Q_{SI}$, $\ell = 2$.

not require building a range tree. Our TLFG abstraction is also more general: our other representations (such as multiway partitioning) do not have any obvious representation as range trees.

**Factorized databases.** Factorized representations of query results [11, 68] have been proposed for equi-joins in the context of enumeration [70, 71], aggregate computation [11], provenance management [56, 69, 70] and machine learning [4, 52, 67, 72, 75]. Our novel TLFG approach to factorization complements this line of research and extends the fundamental idea of factorization to ranked enumeration for theta-joins. For probabilistic databases, factorization of non-equalities [65] and inequalities [66] is possible with OBDDs. Although these are for a different purpose, we note that the latter exploits the transitivity of inequality, as our **Shared Ranges** (Figure 4d) and other approaches for aggregates do [26].

**Top-k queries.** Top-k queries [74] are a special case of ranked enumeration where the value of $k$ is given in advance and its knowledge can be exploited. Fagin et al. [37] present the Threshold Algorithm, which is instance-optimal under a "middleware" cost model for a restricted class of 1-to-1 joins. Follow-up work generalizes the idea to more general joins [38, 46, 57, 87], including theta-joins [59]. Since all these approaches focus on the middleware cost model, they do not provide non-trivial worst-case guarantees when the join cost is taken into account [80]. Ilyas et al. [47] survey some of these approaches, along with some related ones such as building top-$k$ indexes [24, 78] or views [29, 43].

**Optimal batch algorithms for joins.** Acyclic equi-joins are evaluated optimally in $O(n + |out|)$ by the Yannakakis algorithm [90], where |out| is the output size. This bound is unattainable for cyclic queries [63], thus worst-case optimal join algorithms [60, 63, 64, 83] settle for the AGM bound [9], i.e., the worst-case output size. (Hyper)tree decomposition methods [5, 40, 58] can improve over these guarantees, while a geometric perspective has led to even stronger notions of optimality [49, 62]. Ngo [61] recounts the development of these ideas. That line of work focuses on batch-computation, i.e., on **producing all the query results**, or on Boolean queries, while we explore ranked enumeration.
8 CONCLUSIONS AND FUTURE WORK

Theta- and inequality-joins of multiple relations are generally considered “hard” and even state-of-the-art commercial DBMSs struggle with their efficient computation. We developed the first ranked-enumeration techniques that achieve non-trivial worst-case guarantees for a large class of these joins: For small k, returning the k top-ranked join answers for full acyclic queries takes only slightly more-than-linear time and space (O(n polylog n)) for any DNF of inequality predicates. For general theta-joins, time and space complexity are quadratic in input size. These are strong worst-case guarantees, close to the lower time bound of O(n) and much lower than the O(n^2) size of intermediate or final results traditional join algorithms may have to deal with. Our results apply to many cyclic joins (modulo higher pre-processing cost depending on query width) and all acyclic joins, even those with selections and many types of projections. In the future, we will study parallel computation and more general cyclic joins and projections.

ACKNOWLEDGMENTS

This work was supported in part by the National Institutes of Health (NIH) under award number R01 NS091421 and 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.de6tu. From www.gbif.org/

[2] 2021. TPC Benchmark H (Decision Support) Revision 3.0.0. http://tpc.org/tpch/

[3] 2020. Bird Occurrences in Oceania. https://doi.org/10.15468/dl.d6u6tj From

[4] 2015. Optimal enumeration: Efficient top-k tree matching. PVLDB 8, 5 (2015), 533–544. https://doi.org/10.14778/2935468.2935478

[5] 2020. Enumeration Complexity of Conjunctive Queries with Functional Dependencies. Theory Comput. Syst. 64, 5 (2020), 828–860. https://link.springer.com/article/10.1007/s00224-020-09937-9

[6] 2012. FDB: A Query Algorithms. Springer, 160–171. https://doi.org/10.1007/978-3-642-30341-2_9

[7] 2019. On Func-

[8] 2017. What do Shannon-Entropy and Ordering in Factorised Databases. VLDB 35, 7 (2019), 844–856. https://doi.org/10.14778/3385634.3385636

[9] 2004. Joining Interval Data

[10] 2015. Robust and efficient algorithms for rank join evaluation. In Proceedings of the Fifth International Workshop on Database Algorithms for middleware. J. Comput. System Sci. 66, 4 (2003), 614–656. https://doi.org/10.1016/S0022-0000(03)00026-6

[11] 2021. Optimal enumeration: Efficient top-

[12] 2013. On the parameterized complexity of an enumeration problem: The black-white vertex ordering problem. In Proceedings of the 11th International Conference on Computing and berthol, Fabian Gerhardt, and Nicole Schweikardt. 2020. Constant Delay Enumeration for Conjunctive Queries: A Tutorial. ACM SIGLOG News 7, 3 (2020), 4–33. https://doi.org/10.1145/3383563.3383566

[13] 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) (LiPtCS), Vol. 138. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 58:1–58:15. https://doi.org/10.4230/LIPIcs.MFCS.2019.58

[14] 2016. Hypergraph Acyclicity Revisited. ACM Comput. Surv. 49, 3, Article 54 (Dec. 2016), 26 pages. https://doi.org/10.1145/2939573

[15] 2014. Optimal enumeration: Efficient top-

[16] 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-01813392

[17] 2006. Necklaces, convolutions, and X+ Y. In European Symposium on Algorithms. Springer, 160–171. https://doi.org/10.1007/11844832_9

[18] 2019. On the Enumeration Complexity of Unions of Conjunctive Queries. In PODS 134–148. https://doi.org/10.1145/3294052.3319700

[19] 2020. Enumeration Complexity of Conjunctive Queries with Functional Dependencies. Theory Comput. Syst. 64, 5 (2020), 828–860. https://link.springer.com/article/10.1007/s00224-020-09937-9

[20] 2013. Optimal enumeration: Efficient top-

[21] 2000. The onion technique: indexing for linear optimization queries. in SIGMOD 391–402. https://doi.org/10.1145/142009.1420331

[22] 2001. Dynamic Enumeration of Similarity Joins. CoRR abs/2105.01818 (2021). arXiv:2105.01818

[23] 2021. Optimal enumeration: Efficient top-

[24] 2010. Boolean functions: Theory, algorithms, and applications. Cambridge University Press. https://doi.org/10.1017/CBO9780511852008

[25] 2009. Robust and efficient algorithms for rank join evaluation. In Proceedings of the Fifth International Workshop on Database Algorithms for middleware. J. Comput. System Sci. 66, 4 (2003), 614–656. https://doi.org/10.1016/S0022-0000(03)00026-6

[26] 2019. On the parameterized complexity of an enumeration problem: The black-white vertex ordering problem. In Proceedings of the 11th International Conference on Computing and
Algorithm 2: Multiway partitioning

1. **Input**: Relations $S, T$, nodes $v_s, v_t$ for $s \in S, t \in T$, predicate $\theta \equiv S.A < T.B$
2. **Output**: A TLFG from the join $S \bowtie T$
3. Sort $S, T$ according to attributes $A, B$
4. **Procedure** `partIneqMulti(S, T, $\theta$)`
   1. $d = \text{vals}(S \cup T) \cap \text{Number of distinct A, B values}$
   2. **if** $d = 1$ **then** return //Base case
   3. $\rho = \lfloor \sqrt{d} \rfloor$ //Number of partitions
   4. Partition $(S \cup T), (S_1 \cup T_1), \ldots, (S_\rho \cup T_\rho)$ with $\rho$-quantiles of distinct values as pivots
   5. **for** $i \leftarrow 1$ to $\rho$ **do**
      6. Materialize intermediate nodes $x_i, y_i$
      7. **foreach** $s$ in $S_i$ do Create edge $v_s \rightarrow x_i$
      8. **foreach** $t$ in $T_i$ do Create edge $y_t \rightarrow v_t$
      9. **for** $j \leftarrow 1$ to $i - 1$ do Create edge $x_j \rightarrow y_i$
   10. `partIneqMulti(S_i, T_i, $\theta$) //Recursive call

A NOMENCLATURE

| Symbol | Definition |
|--------|------------|
| $Q$    | Join query |
| $R, S, T$ | Relations |
| $A, B, C$ | Attributes |
| $X, Y, Z$ | Lists of attributes |
| $r, s, t$ | Tuples |
| $\theta$ | Predicate $S \rightarrow T$ on predicate $\theta$ |
| $n$ | Total number of tuples |
| $S$ | Number of distinct values |
| $t$ | 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 $s \in S, t \in T$ |
| $S$ | Size of TLFG |
| $d$ | Depth of TLFG |
| $u$ | Duplication factor of TLFG |
| $\rho$ | Number of conjuncts or disjuncts |
| $P$ | Number of partitions in equality/inequality factorization |
| $M_l$ | Partition in inequality factorization |
| $m$ | Number of groups in band factorization |
| $H_l$ | Group in band factorization |
| $TT(k)$ | Time-to-$k^{th}$ result |
| $MEM(k)$ | Memory until the $k^{th}$ result |
| $T$ | Time for constructing a TLFG |
| $P(n)$ | Time for preprocessing |
| $h$ | Height of tree |
| $f, g$ | (Computable) functions |

B DELAY VS TT$(k)$ AS COMPLEXITY MEASURE

In this section, we discuss the relationship between delay and TT$(k)$ as complexity measures for enumeration. For unranked enumeration, our goal is to achieve TT$(k) = O(\mathcal{P}(n) + k)$ with the lowest possible preprocessing time $\mathcal{P}(n)$. The majority of papers on enumeration [10, 19, 45, 76] have traditionally focused instead on constant delay after $\mathcal{P}(n)$ preprocessing. This is desirable because it implies the same guarantee TT$(k) = O(\mathcal{P}(n) + k)$. However, setting constant delay as the goal can lead to misjudgments about practical performance, as we illustrate next:

**Example 16.** 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 $A$ and $B$ spend preprocessing $\mathcal{P}(n)$, then generate a sequence of results with constant delay. For $A$, let this sequence be $1, 2, \ldots, n/2, 1, 2, \ldots, n/2, n/2 + 1, \ldots$ and for $B$ it is $1, 2, \ldots, n/2, 2, n/2 + 1, \ldots$ (see Fig. 11). Even though both achieve TT$(k) = O(\mathcal{P}(n) + k)$, due to duplicate filtering the worst-case delay of $A$ is $O(n)$ (between $n/2$ and $n/2 + 1$), while $B$ has $O(1)$ delay. However, $B$ is clearly slower than $A$ by a factor of 2 for all $k \in [n/2]$. Since $A$ outputs all these values earlier than $B$, we could make $A$ simulate the delay of $B$ 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 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 TT$(k) = O(\mathcal{P}(n) + k \log k)$.

C MULTIWAY PARTITIONING

We provide more details on the multiway partitioning method discussed in Section 5.1. 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. 3).

The main idea is to create more data partitions per recursive step. In particular, we pick $\rho - 1$ pivots that create $\rho$ partitions of nodes with a roughly equal number of distinct values. Fig. 12b 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 $O(\sqrt{d})$, 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 17.** Let $\theta$ be an inequality predicate between relations $S, T$ of total size $n$. A duplicate-free TLFG of the join $S \bowtie T$ 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
This adds up to \( O(\log \log n) \) time. Since we spend linear time per problem, the total work in every level of the recursion tree is \( O(\sqrt{\frac{n}{\sqrt{T}}} + 1) \). Since we spend linear time per problem, the total work in every level of the recursion tree is \( O(\sqrt{\frac{n}{\sqrt{T}}} + 1) \). Since we spend linear time per problem, the total work in every level of the recursion tree is \( O(\sqrt{\frac{n}{\sqrt{T}}} + 1) \). Since we spend linear time per problem, the total work in every level of the recursion tree is \( O(\sqrt{\frac{n}{\sqrt{T}}} + 1) \).

\[ \text{Lemma 18.} \quad \text{Let } \theta \text{ be an non-equality predicate between relations } S, T \text{ of total size } n. \quad \text{A duplicate-free TLFG of the join } S \bowtie_0 T \text{ of size } O(n \log \log n) \text{ and depth } 3 \text{ can be constructed in } O(n \log n) \text{ time.} \]

**Proof.** We sort once in \( O(n \log n) \) and then call the inequality multway 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.

**Theorem 19.** TLFG of the join \( S \bowtie_0 T \) of size \( S.A < T.B \) can be constructed in \( O(n \log n) \) time and \( O(n) \) space.

\[ \text{Theorem 19.} \quad \text{TLFG of the join } S \bowtie_0 T \text{ of size } S.A < T.B \text{ can be constructed in } O(n \log n) \text{ time and } O(n) \text{ space.} \]

**Proof.** We sort once in \( O(n \log n) \) and then call the inequality multway 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.

**Theorem 19.** TLFG of the join \( S \bowtie_0 T \) of size \( S.A < T.B \) can be constructed in \( O(n \log n) \) time and \( O(n) \) space.

**Proof.** We sort once in \( O(n \log n) \) and then call the inequality multway 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.

**Theorem 19.** TLFG of the join \( S \bowtie_0 T \) of size \( S.A < T.B \) can be constructed in \( O(n \log n) \) time and \( O(n) \) space.

**Proof.** We sort once in \( O(n \log n) \) and then call the inequality multway 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.
Algorithm 3: Handling a band predicate

1. **Input:** Relations \( S, T \), nodes \( v_s, v_t \) for \( s \in S, t \in T \), predicate \( \theta \equiv |S.A - T.B| < \varepsilon \)
2. **Output:** A TLFG of the join \( S \bowtie \theta \) \( T \)
3. Sort \( S, T \) according to attributes \( A, B \)
4. **foreach** \((S_b, T_b, s_b, t_b)\) in bandToIneq \((S, T, \theta)\) do
   5. partineqMulti \((S_b, T_b, s_b, t_b)\)
7. **Function** bandToIneq \((S, T, \theta)\)
   8. ineqs = []
9. //Find the limits of the groups on the right
10. \( H_1.\text{start} = t_1.B, m = 1 \)
11. for \( i \leftarrow 1 \) to \(|T|\) do
12. \( H_m.\text{start} = T[i].B, m = 1 \)
13. \( H_m.\text{end} = T[1].B \)
14. ineqs.add \((S, T, \theta)\)
15. **foreach** \( H_j \) in \([H_1, \ldots, H_m]\) do
16. //Assign tuples to the group
17. \( S_j = \{ s \in S | H_j.\text{start} - \varepsilon \leq s.A \leq H_j.\text{end} + \varepsilon \} \)
18. \( T_j = \{ t \in T | H_j.\text{start} \leq t.B \leq H_j.\text{end} \} \)
19. //Greater-than inequality
20. \( S_\alpha = \{ s \in S_j | s.A > T_j.\text{start} + \varepsilon \} \)
21. ineqs.add \((S_j, T_j, S_\alpha > T_j.B - \varepsilon)\)
22. //Less-than inequality
23. ineqs.add \((S_j - S_\alpha, T_j, S_\alpha < T_j.B + \varepsilon)\)
24. ineqs = []
25. return ineqs

\( \varepsilon \). A source tuple is assigned to a group if it joins with at least one target tuple in the group. Since the groups represent -intervals of target tuples, each source tuple can be assigned to at most three groups.

Example 19. Figure 13 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 \( \varepsilon \) grows together with the domain size. Initially, we group the target tuples by \( \varepsilon \) intervals (Fig. 13a). 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. 13b. 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.

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 \( \varepsilon \). 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 20.** Let \( \theta \) 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 \( O(n \log n) \) and depth 3 can be constructed in \( O(n \log n) \) time.

**Proof.** First, we create disjoint \( T \) groups based on \( \varepsilon \)-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 \( 2\varepsilon \). 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 \( \varepsilon \) apart on the \( B \) attribute. Since all the assigned \( S_j \) tuples 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 a 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 \( v_t \) with all \( v_s \) 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, \( T_B(n) \) is the time for factorizing a band condition and \( T_I(n) \) for an inequality, we have \( T_B(n) = O(n) + 2T_I(n_1) + 2T_I(n_2) + \ldots + 2T_I(n_m) \), since we call the inequality algorithm twice within each group. For \( T_I(n) = O(n \log n) \), we get \( T_B(n) = O(n \log n) \), 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.

**F ADDITIONAL PROOFS**

**F.1 Proof of Theorem 6**

Since each TLFG that is in-between two relation layers has \( O(|E|) \) edges and \( O(\lambda) \) layers, the enumeration graph has \( O(|E|) \) edges and \( O(\lambda) \) layers as well. That is because the number of relation layers is \( \ell \), which is considered to be constant. The theorem follows by applying Lemma 2 on the resulting enumeration graph.

**F.2 Proof of Lemma 9**

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 \(h\) of the tree is \(O(\log d) = O(\log n)\). Overall, the time spent on the recursion is \(O(nh) = O(n \log n)\), 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 \(s' \in S', t' \in T'\).

### F.3 Handling equality predicates in a conjunction

**Lemma 21.** Let \(\theta\) be a conjunction of predicates between relations \(S, T\) of total size \(n\), and \(\theta'\) 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 \(d\), 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 \(d\), and duplication factor \(u\) in time \(O(g(n) + n)\).

**Proof.** 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 \(\theta'\) through some algorithm \(A\). If \(A\) elects to connect two nodes, then they satisfy both \(\theta'\), 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 \(A\), thus do not satisfy \(\theta'\).

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 \(A\), or increase the depth of each TLFG.

### F.4 Proof of Lemma 11

As a first step, all the equality predicates are handled by Lemma 21. 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 \(T_I(n, p)\) the running time for \(n\) tuples and \(p\) inequality predicates. We proceed by induction on the number of predicates \(p\) to show that \(T_I(n, p) \leq f(p) n \log^P n\) 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: The 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 \(T_I(n, 1) \leq c n \log n\) for sufficiently large \(n\). For the inductive step, we assume that \(T_I(n, p - 1) \leq f(p - 1) n \log^{P-1} n\). 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 \(T_I(n', p - 1)\). 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 \(T_I(n_1, p - 1) + \ldots + T_I(n_\rho, p - 1) \leq f(p - 1) n_1 \log^{P-1} n_1 + \ldots + f(p - 1) n_\rho \log^{P-1} n_\rho \leq f(p - 1) n \log^{P-1} n\).

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 \(c'' n \log^P n\). Thus, we get that \(T_I(n, p) \leq c' f(p - 1) n \log^P n + c'' n \log^P n\). If we pick a function \(f\) such that \(f(1) \geq c\) and \(f(p) \geq c' f(p - 1) + \frac{c'}{c'' n \log^P n}\), then \(T_I(n, p) \leq f(p) n \log^P n\). This completes the induction, establishing that \(T_I(n, p) = O(n \log^P n)\) 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).

### F.5 Proof of Lemma 12

Correctness follows from the fact that the paths in the constructed TLFG is the union of the paths in the TLFGs for \(S \bowtie \theta\ T\). For the depth, note that each \(T_\theta\) 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 \(p\).

### F.6 Proof of Lemma 13

For each edge of the theta-join tree, we construct a TLFG by processing the join condition as a DNF formula. The guarantees of the theorem follow from Theorem 6 by applying the properties of the TLFGs we construct, along with a duplicate elimination filter.

To construct each TLFG, disjunctions are handled according to Lemma 12 and for conjunctions, the proof is the same as that of Lemma 11 with some changes: 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 (see Lemma 13). Equalities are removed from the conjunction because of Lemma 21 and the fact that \(n \log^P n\) and \(n \log^{P-1} n \cdot \log n\) are superadditive functions. In the conjunction algorithm, we use multiway partitioning for \(p = 1\) and binary partitioning for \(p > 1\). Therefore \(T_I(n, 1) \leq c n \log n\), resulting in \(T_I(n, p) = O(n \log^{P-1} n \cdot \log n)\) overall. Non-equalities and bands are translated into inequalities by using the techniques we developed in Appendices D and E: a non-inequality results into two inequalities on the same set of nodes, while a band creates multiple inequality subproblems. We use the same arguments as in the proofs of Lemmas 18 and 20. We denote by \(T_I(n, p), T_N(n, p), T_B(n, p)\) 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. $T_N(n, p) = O(T_I(n, p) + T_I(n, p))$ and $T_D(n, p) = O(n) + 2T_I(n_1, p) + 2T_I(n_2, p) + \ldots + T_I(n_m, p)$ for $n_1 + n_2 + \ldots + n_m \leq 3n$. By these formulas, and since $T_I(n, p) = O(n \log^{p-1} n \cdot \log \log n)$, 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.

As we are enumerating subtrees of the enumeration graph in order, we detect those that correspond to duplicate query results and filter them out using a lookup table. The duplication factor of our TLFGs is $n$, except if we have disjunctions (Lemma 12). Let $u_{\text{max}}$ be the maximum duplication factor among the constructed TLFGs. The number of "duplicate" query answers (that correspond to the same answer $q$ of $Q$) are bounded by $u_{\text{max}}^\ell$, where $\ell$ is the number of $Q$ atoms. That depends only on the query size which we consider as constant, thus it is $O(1)$. If the time for each answer without the filtering is $TT'(k)$, then we have that $TT(k) = O(TT'(k \cdot u_{\text{max}}^\ell)) = O(TT'(k))$, since $u_{\text{max}}$ and $\ell$ are $O(1)$.

G SQL CODE FOR QUERIES USED IN EXPERIMENTS

Query $Q_{S1}$:

SELECT * AS Weight
FROM S1, S2
WHERE S1.A2 < S2.A3
ORDER BY Weight ASC

Query $Q_{S2}$:

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

Query $Q_{R1}$:

SELECT *, R1.Sentiment + R2.Sentiment AS Weight
FROM Reddit R1, Reddit R2
WHERE R1.To = R2.From AND R2.Timestamp > R1.Timestamp
ORDER BY Weight ASC

Query $Q_{R2}$:

SELECT *, R1.Readability + R2.Readability AS Weight
FROM Reddit R1, Reddit R2
WHERE R1.To = R2.From AND R2.Timestamp > R1.Timestamp AND R2.Sentiment < R1.Sentiment
ORDER BY Weight DESC

Query $Q_{B}$:

SELECT *, B1.IndivCount + R2.IndivCount AS Weight
FROM Birds B1, Birds B2
WHERE ABS(B2.Latitude - B1.Latitude) < $\epsilon$ AND ABS(B2.Longitude - B1.Longitude) < $\epsilon$
ORDER BY Weight DESC

H TLFG FACTORIZATION FORMULAS

Typically, factorization refers to the process of compacting an algebraic formula by factoring out common sub-expressions using the distributivity property [28]. Under that perspective, factorized databases [68] represent the results of an equi-join efficiently, treating them as a formula built with product and union. Besides distributivity, $d$-representations [71] replace shared sub-expressions with variables, further improving succinctness through memoization [30]. 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 22 below.

Example 22. Consider the inequality join $S \bowtie_{A < B} T$. A naive TLFG for some example relations $S, T$ is shown in Fig. 4c. 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. 4e. We can write the corresponding algebraic formula by defining new variables $v_i, i \in [5]$ for the intermediate nodes:

$$
\Phi_3 = (1 \times v_1) \cup (2 \times v_2) \cup (2 \times v_3) \cup (3 \times v_4), \ldots, (5 \times v_5)
$$

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

Notice that the total size of these formulas is asymptotically the same as the TLFG size.

I APPLICATION OF THE TECHNIQUE TO UNRANKED ENUMERATION

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 $O(n \log \log n + k \epsilon)$.

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, and band predicates. Let $p$ be the maximum number of predicates, excluding equalities, in a conjunction of a DNF on any edge of the theta-join tree. Ranked enumeration of the answers to $Q$ over $D$ can be performed with $TT(k) = O(n \log^p n + k \log k)$. The space requirement is $MEM(k) = O(n \log^{p-1} n \cdot \log \log n + k)$.

Theorem 23. 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, and band predicates. Let $p$ be the maximum number of predicates, excluding equalities, in a conjunction of a DNF on any edge of the theta-join tree. Enumeration of the answers to $Q$ over $D$ in an arbitrary order can be performed with $TT(k) = O(n \log^p n + k)$ and $MEM(k) = O(n \log^{p-1} n \cdot \log \log n + k)$.
I.1 Experimental Comparison

We use \( Q^L_U \) for the query that is the same as \( Q_T \), but without the ranking. To illustrate how the duplicates from disjunctions or the presence of ranking change the delay of the enumeration, we plot \( T(k) \) for query \( Q_T \), together with its disjunction \( Q_{TD} \) and unranked \( Q^U_T \) variants (Figure 14a). For \( Q_{TD} \) the constructed TLFG is 3 times larger (because of the three date inequalities), which is reflected in the time it starts to return results. The delay is higher by a similar factor, since the three predicates in the disjunction have a very high overlap. In fact, that is the worst case for our technique because of the high number of duplicates that have to be filtered. As illustrated in Figure 14b, this number is not affected by the size of the database and only depends on the query. Without the ranking, the enumeration for \( Q^L_U \) starts slightly faster than \( Q_T \) and has significantly lower delay between results.

J WHY THE DBMS TOP-K PLAN MUST PRODUCE THE ENTIRE OUTPUT

In this section, we discuss why any approach that first applies the join and then the ranking (e.g. with a heap over the join results) will unavoidably spend \( O(n^2) \) even for a simple binary join with one inequality predicate.

First, we would like to emphasize that **we do not compare against a naive \( O(n^2) \) join algorithm.** The quadratic worst-case complexity is not caused by an inferior join algorithm but by the output size itself. In short, even if we want to retrieve only \( k \) join output tuples, the algorithm has to insert \( O(n^2) \) output tuples into the heap: At any moment in time (until the full output is known) the algorithm does not know if all of the top-\( k \) answers are already in the heap or if some of them will be emitted by the join later.

We illustrate this with an example. Consider the inequality join in Figure 15a with join condition \( S.A < T.B. \) To efficiently find joining pairs, we can sort input relation \( S \) on \( A \) and \( T \) on \( B. \) (Alternatively, one could use clustered B-tree indexes—one on \( A \) for \( S \) and the other on \( B \) for \( T \)—to the same effect.) This step indeed takes \( O(n \log n) \) time and it allows us to retrieve the joining pairs with a sort-merge type algorithm. Using the sorted inputs, this algorithm can produce \( k \) output tuples in time \( O(k) \). With \( k \) upper bounded by some constant, say \( k = 3 \), \( k \) join answers can then indeed be retrieved in total time \( O(n \log n) \).

While this works well if we want to get an arbitrary set of \( k \) result tuples, **ranking makes the situation more challenging.** To illustrate this, suppose in the example we want to find the top-3 join results **according to the minimum sum of weights** \( W_S + W_T \). Notice that in general, tuple weights may or may not be correlated with join-attribute values. In our example, we highlight the top-3 joining pairs (\( (1, n), (2, 1 - n + 1) \), (\( (i, 0), (i + 1, i + 1) \), and (\( (i, 0), (i + 2, i + 2) \)) with blue edges, where \( i \) is some value \( 1 < i < n - 1 \). Notice that even after sorting each relation by the join attributes, the algorithm still does not know in which positions in each sorted relation the winning \( W_S + W_T \) combinations occur.

This means that as the join algorithm returns output tuples, the weight sum \( W_S + W_T \) may go up or down between consecutive output tuples as illustrated in Figure 15b, where we show how the heap gradually fills up with output tuples from the join. We cannot determine the winners until all the \( O(n^2) \) join results have been inserted into the heap. Even in the middle step where the top-3 results happen to be in the heap already, we **cannot stop the join computation early because the algorithm does not know if a not-yet-returmed join output tuple could have a lower sum of weights.** Only after all the join result tuples have been inserted into the heap can the algorithm know for sure what the top-\( k \) results based on weight \( W_S + W_T \) are. This implies that in order to find the top-\( k \) results, even for a small value of \( k \), the algorithm must run the join until the end, i.e., consider all matching combinations produced by the join. No matter how efficient the join implementation or the heap data structure, just looking once at each of the \( O(n^2) \) join output tuples already takes time \( O(n^2) \)—and this is the quadratic complexity we refer to.

One may look at the example and think "couldn’t we avoid having to look at the entire join output by making join processing more aware of the weight attributes?" And that is exactly what our algorithm does. The challenge is that when sorting the input by \( W_S \) and \( W_T \), respectively, the **first pairs of \( S \) and \( T \) tuples considered based on weight may not join at all.** In our example, the lightest \( S \)-tuples are (\( i, 0 \), (\( i + 1, 1 \), but unfortunately for larger values of \( i \) they do not join with the lightest \( T \)-tuples (\( 2, 1 - n + 1 \), (3, 3) etc. Therefore, there is no guarantee that the winning pairs will be found in less than \( O(n^2) \) time when following the weight order on the input. (This may seem "not too bad" for the specific example, but is a major concern for more complex DNFs of inequalities and for joins of more than 2 relations.)

To summarize, there are 2 non-trivial aspects of the problem: (1) determine which pairs of input tuples join with each other, and (2) rank the joining pairs by sum of weights or another given ranking function. **No approach that we know of, including the sort-join-and-heap algorithm can do both (1) and (2)—even for a join of only 2 relations—while guaranteeing worst-case time complexity better than \( O(n^2) \).** This holds even if one asks only for the \( k \) top-ranked (by weight) results for some constant \( k \).

The techniques proposed in our paper avoid that cost by **joining and ranking simultaneously, achieving end-to-end complexity of** \( O(n \log n) \) for a 2-relation join with one inequality or one band-join condition (and \( O(n \log n) \) for a general DNF of inequality conditions) to retrieve the top-\( k \) results. Stated differently, it takes a non-trivial combination of both sorting by join attributes and sorting by ranking function—and that is the core of our factorization approach.
K MORE MOTIVATING EXAMPLES

Example 24. Consider an ornithologist studying interactions between bird species using a bird observation dataset \( B(\text{Species, Family, ObsCount, Latitude, Longitude}) \). 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 \( \text{ObsCount} \) should also appear first:

\[
\text{SELECT *}, B1.\text{ObsCount} + B2.\text{ObsCount} \text{ as Weight} \\
\text{FROM } B B1, B B2 \\
\text{WHERE } B1.\text{Family} = B2.\text{Family} \\
\quad \text{AND } \text{ABS}(B1.\text{Latitude} - B2.\text{Latitude}) < 1 \\
\quad \text{AND } \text{ABS}(B1.\text{Longitude} - B2.\text{Longitude}) < 1 \\
\quad \text{AND } B1.\text{Species} <> B2.\text{Species} \\
\text{ORDER BY Weight DESC LIMIT 1000}
\]

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.