Beyond Equi-joins: Ranking, Enumeration and Factorization

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

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

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

ABSTRACT

We study 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 $O(n \operatorname{polylog} n + k \log k)$ time. This is within a polylogarithmic factor of the best known complexity for equi-joins and even of $O(n + k)$, 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, such as the so-called free-connex queries. Remarkably, they hold even when the entire output is of size n^ℓ for a join of ℓ relations. The key ingredient is a novel $O(n \operatorname{polylog} n)$-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 [47, 58, 61, 62]. Similarly, work on *constant-delay enumeration* [8, 17, 42, 74] strives to pre-process the database for a given query on-the-fly so that the first answer is returned in linear time (in database size), followed by all other answers with constant (i.e., independent of database size) delay between them. 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 generalizes the heavily studied *top-k* paradigm [35, 45] by continuously returning join answers in ranking order. This enables the output consumer to select k on-the-fly while observing the answers. For top-k, the value of k must be chosen in advance, without knowing any query answers. Unfortunately, non-trivial complexity guarantees of previous top-k techniques, including the celebrated Threshold Algorithm [35], are limited to the "middleware" cost model, which only accounts for the number of distinct data items accessed [78]. While some of those top-k algorithms can be applied to joins with inequality and general theta-join predicates, they do not provide non-trivial guarantees in the standard RAM model of computation, i.e., their time complexity for a join of ℓ relations is $O(n^\ell)$.

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, 25], 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 DBMS 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.) We tried a large variety of inputs, indexes on the input relations, join queries, and values of k, and the optimizer of PostgreSQL and two other widely used commercial DBMS always chose to execute the join before applying the ranking and top-k condition on the entire output. (In general, for non-trivial ranking functions or when the attributes used for joining differ from those used for ranking, the DBMS cannot determine if some subset of the join output it has produced so far already contains all k top-ranked answers.) While this approach ensures low join cost because the ranking is not taken into account when performing the join, it implies that overall time complexity to return even the top-1 result cannot be better than the worst-case join output size, which is $O(n^\ell)$ for an acyclic join of ℓ relations.

Beyond equi-joins. Recent work on ranked enumeration [30, 32, 77, 78, 83, 84] achieved much stronger worst-case guarantees, but only considered *equi-joins*. However, big-data analytics often also requires other join conditions [31, 34, 48, 52] such as *inequalities* (e.g. `S.age < T.age`), *non-equalities* (e.g. `S.id ≠ T.id`), and *band* predicates (e.g. `|S.time - T.time| < ε`). 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 joins that involve inequalities concerns graph-based approaches that detect "lateral movement" between infected computers in a network [53]. By modeling computers as nodes and connections as timestamped edges, these approaches search for anomalous access patterns that take the form of paths in the graph (or in general, any pattern) scored by the probability of occurrence according to historical data. Such patterns need to satisfy a time constraint; the timestamps of two consecutive edges need to be in ascending order. Concretely, consider the relation* `G(From, To, Time, Prob)`*. Valid* $2-$*hop paths can be computed with a self-join (where* G_1, G_2 *are*

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

Figure 1: Preprocessing time $\mathcal{P}(n)$ and space complexity $\mathcal{S}(n)$ of our approach for various join conditions. Our novel factorized representation allows ranked enumeration to return the k top-ranked results in time $\mathrm{TT}(k) = O(\mathcal{P}(n) + k \log k)$, using $\mathrm{MEM}(k) = O(\mathcal{S}(n) + k)$ space.

aliases of G) where the join condition is an equality G_1.To = G_2.From *and an inequality* G_1.Time < G_2.Time*, while the score of a path is* G_1.Prob × G_2.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 we develop in this paper, patterns of much larger size can be ranked efficiently without considering all the polynomially-many instantiations of the pattern.*

Main contributions. We provide the first comprehensive study on ranked enumeration for joins with conditions other than equality, including general theta-joins and conjunctions and disjunctions of inequalities and equalities. While such joins are considered expensive to compute [48, 52], we show that for many of them the top-ranked answers can *always* be found in time complexity that only slightly exceeds the complexity of sorting the input. This is remarkable, given that input may be heavily skewed and output size of a join of ℓ relations is $O(n^\ell)$. We achieve this with a carefully designed factorized representation of the join *output* that can be constructed in relatively small time and space. Then the ranking function determines the traversal order on this representation.

Recall that ranked-enumeration algorithms must continuously output answer tuples in order and the goal is to achieve non-trivial complexity guarantees no matter at which value of k the algorithm is stopped. Hence we express algorithm complexity as a function of k: $\mathrm{TT}(k)$ and $\mathrm{MEM}(k)$ denote the algorithm's time and space complexity, respectively, until the moment it returns the k-th answer in ranking order. Our main contributions (see also Figure 1) are:

(1) We generalize a previously-proposed equi-join-specific ranked-enumeration construction [77] by breaking it into two orthogonal abstractions: one to represent the overall join structure as a tree of joining relations and the other to represent the join condition for each pair of adjacent relations in the tree. For the latter, we propose the Tuple-Level Factorization Graph (TLFG, Section 3)—a novel factorized representation for any theta-join between two relations—and show how its size and depth affect the complexity of ranked enumeration. Interestingly, some TLFGs can be used to transform a given theta-join to an equi-join, a property we leverage for ranked enumeration for *cyclic* join queries.

(2) For join conditions that are a DNF of inequalities (Section 4), we propose concrete TLFGs with space and construction-time complexity $O(n \operatorname{polylog} n)$. Using them, our algorithm guarantees $\mathrm{TT}(k) = O(n \operatorname{polylog} n + k \log k)$ for acyclic joins, which is within a polylogarithmic factor of the equi-join case, where $\mathrm{TT}(k) = O(n + k \log k)$ [77], 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 DBMS, 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 appendix. More information can be found on our project website at https://northeastern-datalab.github.io/anyk/.

2 PRELIMINARIES

2.1 Queries

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

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

where R_i are relational symbols, $\mathbf{X}_i$ are lists of variables (or attributes), $\mathbf{Z}, \mathbf{Y}_i$ are subsets of $\bigcup \mathbf{X}_i$, $i \in [\ell]$, $j \in [q]$, and θ_j are Boolean formulas called *join predicates*. The terms $R_i(\mathbf{X}_i)$ are called the atoms of the query. Equality predicates are encoded by repeat occurrences of the same variable in different atoms; all other join predicates are encoded in the corresponding θ_j. If no predicates θ_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 of finite relations (or tables) R_i, which draw values from a domain that we assume to be $\mathbb{R}$ for simplicity.[1] The maximum number of tuples in an input relation is denoted by n. We write $R_i.A$ for an attribute A of relation R and $r.A$ for the value of A in tuple $r \in R_i$. Without loss of generality, we assume that relational symbols in different atoms are distinct since self-joins can be handled with linear overhead by copying a relation to a new one. The output of the query is the Cartesian product of the ℓ relations, from which only tuples that satisfy the equi-join conditions and θ_j predicates are selected, followed by projection on the $\mathbf{Z}$ attributes. (This semantics can typically be implemented with more efficient query plans.) 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 ($\mathbf{Z} = \bigcup \mathbf{X}_i$). While our approach can handle projections by applying them in the end, the strong asymptotic $\mathrm{TT}(k)$ guarantees may not hold any more. The reason is that 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 suppressed, 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 DBMS use when the SQL query has a SELECT clause instead of SELECT DISTINCT.

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

Even for set semantics and SELECT DISTINCT queries, it is straightforward to extend our strong guarantees to non-full queries that are *free-connex* [8, 11, 15, 43].

Atomic Join Predicates. We define the following types of predicates between attributes $S.A$ and $T.B$: an *inequality* is $S.A < T.B$, $S.A > T.B$, $S.A \leq T.B$, or $S.A \geq T.B$, a *non-equality* is $S.A \neq T.B$ and a *band* is $|S.A - T.B| < \varepsilon$ for some $\varepsilon > 0$. Our approach also supports numerical expressions over input tuples, e.g., $f(S.A_1, S.A_2, \ldots) < g(T.B_1, T.B_2, \ldots)$, with f and g arbitrary $O(1)$-time computable functions that map to $\mathbb{R}$. The join predicates θ_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 returns join answers one-at-a-time, in the order dictated by a given ranking function. We further require that no answer be output more than once. An obvious solution would be to compute the entire join output, insert it into a heap data structure, and then repeatedly remove the top element from the heap. (Similarly, an array of output tuples could be batch-sorted, followed by iterating over the sorted array.) Our goal is to find more efficient solutions for appropriate ranking functions.

For simplicity, in this paper we focus on ranking by *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 represented by a *selective dioid* [77]. Intuitively, a selective dioid [37] is a semiring that also establishes a total order on the domain. It has two operations, min and + for sum-of-weight, where one *distributes* over the other. These structures include even less obvious cases such as lexicographic ordering by relation attributes. In fact, all practically relevant *monotonic* ranking functions [35] we are aware of can be represented by a selective dioid.

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* [79], 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 [13, 20, 38], 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^{th} result is returned ($\text{TT}(k)$) for all values of k. In Appendix B, we further discuss the relationship of $\text{TT}(k)$ to enumeration delay as complexity measures. Since we do not assume any given indexes, a trivial lower bound is $\text{TT}(k) = O(n + k)$: 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 $\text{MEM}(k)$ to denote the required memory until the k^{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.

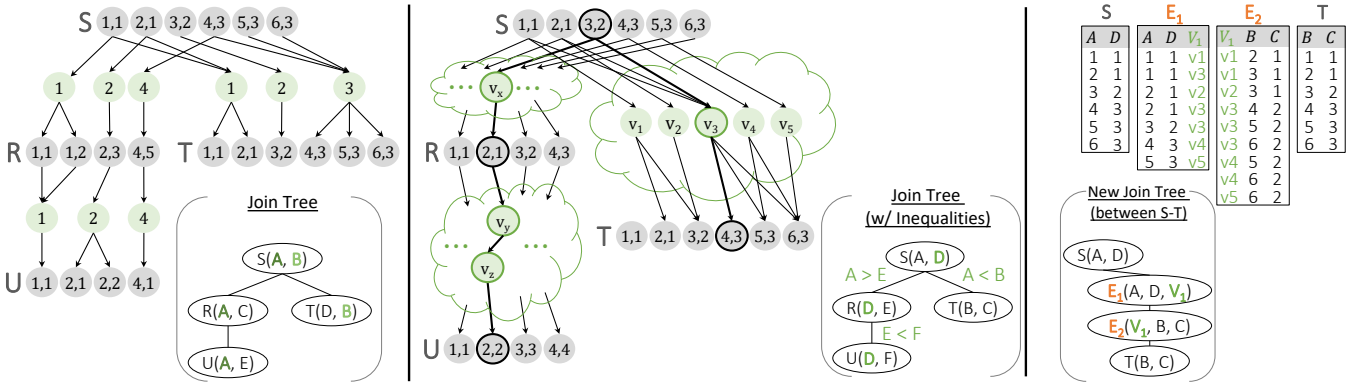
3.1 Previous Work: Any-k Algorithms

Our *any-k* algorithms [77] reduce ranked enumeration for *equi-joins* to the problem of finding the k^{th}-lightest sub-trees (called T-DP), generalizing Dynamic Programming and k^{th}-shortest path algorithms. After determining the join tree for a given equi-join query, e.g., by using the classic GYO reduction [86] for acyclic joins, the any-k algorithm works on an *enumeration graph* as depicted for an example in Fig. 2a. It is a layered DAG in the following sense: (1) Each node is assigned a unique layer ID (not shown in the figure to avoid clutter). (2) Each edge is directed, going from a lower to a higher layer ID. (3) All tuples from an input relation appear as nodes in the same layer (black shaded nodes), called a *relation layer*. Each relation layer has a unique ID, with ID 1 assigned to the root of the join tree (S in the example). Furthermore, a relation in the join tree must have a greater layer ID than its ancestors. (4) If and only if two relations are adjacent in the join tree, then they are connected via a *connection layer* that contains nodes representing their join-attribute values (green shaded nodes). A connection-layer's ID must be between the IDs of its adjacent relation layers.

Given a query, the enumeration graph is constructed on-the-fly and bottom-up, according to a join-tree representation 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. 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 weights for a node's outgoing edges are obtained at a constant cost per weight 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-k algorithm. The following lemma summarizes the main result from [77]. We restate it here in terms of data complexity (where query size ℓ is a constant) and using λ to denote the number of layers.[2]

[2] Due to the specific construction in [77], there λ was linear in query size ℓ and hence ℓ and λ were used interchangeably. In our generalization this may not be the case, therefore we use the more precise parameter λ here.



(a) Equi-join enumeration graph, join tree [77]. **(b) Abstraction proposed in this paper.** **(c) Transformation to equi-join.**

Figure 2: Overview of our approach.

To extend the any-k framework beyond equi-joins, we must address two major challenges. First, the more general join conditions need to be mapped to a layered graph where each connected subtree that contains exactly 1 node from each relation layer corresponds to a join answer, and vice versa. (Figure 2b shows an example subtree highlighted for the answer obtained by joining input tuples $s = (3, 2)$, $t = (4, 3)$, $r = (2, 1)$, and $u = (2, 2)$.) And second, for the strongest complexity guarantees (Lemma 2), the graph should have a small number of edges $|E|$ and layers λ. We address these challenges by breaking the "monolithic" equi-join approach into two orthogonal abstractions. The first is a theta-join tree that can be efficiently computed for any theta-join query, and the second is a novel factorized representation of the join between two relations.

3.2 Generalized Join Tree and Graph Structure

The join tree is essential for generating the enumeration graph, but in contrast to equi-joins (aka conjunctive queries), for general join conditions there is no established methodology for how to define or find a join tree. We need a generalized join tree that can be constructed efficiently for a given theta-join and then be mapped to a compact layered graph. To this end, we propose to extend the usual definition of (alpha-)acyclicity [39, 75, 86] from equi-joins to theta-joins by defining a join query as *acyclic* if we can construct a *theta-join tree* with the atoms (relations) as the nodes where (1) for every attribute A appearing in an atom, all nodes containing A form a connected subtree and (2) every join predicate θ_j is assigned to an edge (S, T) such that S and T contain all the attributes referenced in θ_j. Notice that condition (1) alone is the standard definition of a join tree for equi-joins [15]. Edge (S, T) represents the join $S \bowtie_\theta T$, where join condition θ is the conjunction of all predicates θ_j 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 3. *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 was assigned to an edge whose adjacent nodes together contain all referenced attributes. For example,* $A < B$ *was assigned to* (S, T) *(*S *contains* A *and* T *contains* B*). The join-tree edges represent join predicates* $\theta_1 = S.A < T.B$ *(edge* (S, T)*),* $\theta_2 = S.A > R.E \wedge S.D = T.D$ *(edge* (S, R)*), and* $\theta_3 = R.E < U.F \wedge R.D = U.D$ *(edge* (R, U)*).*

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

We can construct the theta-join tree efficiently by first removing all θ_j 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 3, this join tree looks like the one in Figure 2b, but without the edge labels and without the edge from S to T. Finally, we attempt to add each θ_j predicate to a join-tree edge: θ_j can be assigned to any edge where the two adjacent nodes contain all the attributes referenced in it. If the join "tree" for Q' is actually a forest of separate trees, two separate trees are connected by making one the subtree of the other until we are left with a single tree. In the example, we make T a child of S so that predicate $A < B$ can be assigned to the newly created edge from S to T. The resulting theta-join tree for the example query is shown in Figure 2b. If either the GYO algorithm fails to find a join tree for Q' or any of the θ_j conditions cannot be assigned to a tree edge, then the query is considered *cyclic* and will be processed as discussed in Section 5.3. (Note that there may exist different join trees for Q' and different choices for merging the separate trees into a single one. Our algorithm can try all possible options, therefore the query is cyclic only if *all* of those cases fail.)

The theta-join-tree construction only considers the atoms and θ_j predicates of the query. And since query size is a constant, the algorithm has constant space and time complexity. 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

Lemma 2 ([77]). *Given an enumeration graph with* $|E|$ *edges and* λ *layers, ranked-enumeration algorithms for the returning the* k *lightest subtrees (which contain exactly* 1 *node from each relation layer) achieve* $\mathrm{TT}(k) = O(|E| + k \log k + k\lambda)$ *and* $\mathrm{MEM}(k) = O(|E| + k\lambda)$.

and each join-tree edge is mapped to a subgraph implementing the join condition between the corresponding relation layers. This is visualized by the green 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 for the S-to-T connection in the example) or even multiple layers (like for the connection between R-node $(2,1)$ and U-node $(2,2)$).

To be able to apply our any-k algorithms [77] to this generalized enumeration graph we must ensure that (1) each "green cloud" can be mapped to a layered graph and (2) there is a 1-to-1 correspondence between join answers and connected subtrees that contain exactly 1 node from each relation layer (like the one highlighted in Figure 2b). 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 iff the corresponding input tuples join. In the example, there is a path from S-node $(3,2)$ via v_3 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 "green 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 4. *A* Tuple-Level Factorization Graph (TLFG) *of a theta-join $S \bowtie_\theta T$ 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 θ.

The *size* of a TLFG $G(V,E)$ is $|V| + |E|$ and its *depth* 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: Source nodes v_s are assigned layer ID 0 and an 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 λ, 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 enumeration graph consists of ℓ relation layers and $\ell - 1$ TLFGs (one for each edge of the theta-join tree), using Lemma 2 we can show:

Theorem 5. *Given a theta-join Q of $\ell = O(1)$ 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 $O(|E|)$ edges and $O(\lambda)$ layers, then the time and space complexity of enumerating the k lightest subtrees in G_Q (which contain exactly 1 node from each relation layer) are $\mathrm{TT}(k) = O(|E| + k \log k + k\lambda)$ and $\mathrm{MEM}(k) = O(|E| + k\lambda)$, respectively.*

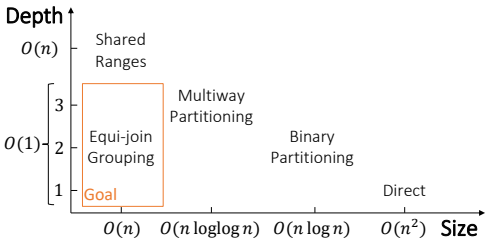


Figure 3: Tradeoff between size and depth of the TLFG for a single inequality. Ideally, we want to achieve $O(n)$ size and $O(1)$ depth, which is possible for equi-joins.

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 $\lambda = 1$, thus $\mathrm{TT}(k) = O(n^2 + k \log k)$ and $\mathrm{MEM}(k) = O(n^2 + k)$, respectively. Hence even the top-ranked result requires quadratic time and space. Clearly, 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 Figure 3, with details discussed in later sections.

Output duplicates. A subtle issue with Theorem 5 is that two non-isomorphic subtrees of the enumeration graph may contain the exact same input tuples (actually, their corresponding relation-layer nodes), causing output duplicates. 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 $\mathrm{TT}(k)$ (or $\mathrm{MEM}(k)$, 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.[3]

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).

[3] 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 graph, using TLFGs to connect adjacent relation layers, enables a novel reduction from complex theta-joins to equi-joins.

Theorem 6. *Let $G = (V, E)$ be a TLFG with λ layers for a theta-join $S \bowtie_\theta T$ between relations S and T and X be their attributes. For $0 < i \leq \lambda$ let E_i denote the set of edges from layer $i-1$ to i. If $E = \bigcup_i E_i$, i.e., every edge in E connects nodes in adjacent layers, then $S \bowtie_\theta T = \pi_X(S \bowtie E_1 \bowtie \cdots \bowtie E_\lambda \bowtie T)$.*

Intuitively, the theorem states that if no edge in the TLFG skips a layer, then the theta-join $S \bowtie_\theta T$ can equivalently be computed as an equi-join between S, T, and λ *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 $\lambda = 2$. 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 $S(A, D) \bowtie_{A<B} T(B, C) = S \bowtie E_1 \bowtie E_2 \bowtie T$. 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 $S \bowtie_\theta T$ 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 6, it can be converted to equi-join $S \bowtie E \bowtie T$, 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 converting the entire theta-join query between ℓ relations to an equi-join) and then just runs the equi-join ranked-enumeration algorithm [77] on the converted join problem 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 both E_1 and E_2 are of size $O(n \operatorname{polylog} n)$. (The example in Figure 2c shows a concrete instance.)

4 FACTORIZATION OF INEQUALITIES

We now show how to construct TLFGs that have $O(n \operatorname{polylog} n)$ size and $O(1)$ depth when the join condition θ in a join $S \bowtie_\theta T$ is a DNF[4] of inequalities (and equalities). First, we consider the simple case of a single inequality, then we generalize it to conjunctions and finally, show how to combine the conjunctions of the DNF. Non-equalities and bands will be discussed later in Section 5.

4.1 Single Inequality Condition

Efficient TLFGs for equi-joins leverage the fact that equality conditions group tuples into *disjoint* equivalence classes (Fig. 4b). Inequality conditions are more challenging because they lack that property; even though shared structure exists (see Fig. 4c), there is no way to separate the joining tuples into disjoint groups. Thus, we develop an approach that allows edges to cross over the partitions.

Binary partitioning. Our representation is inspired by how quicksort partitions an array based on a pivot element [40]. We call this approach *binary partitioning*. Suppose that we have a less-than condition $S.A < T.B$. We pick a pivot value v and then partition both relations S and T s.t. $s.A < v$ for $s \in S_1$ and $s.A \geq v$ for $s \in S_2$, and similarly $t.B < v$ for $t \in T_1$ and $t.B \geq v$ for $t \in T_2$. Thereby, we know that all values in S_1 are strictly less than those in T_2. Thus, we connect them in the graph via a single intermediate node. Then, we continue on the two horizontal partitions (S_1, T_1) and (S_2, T_2) *recursively*. Since S_2 tuples cannot join with T_1 tuples by construction, we do not miss any joining pairs. Importantly, the intermediate node we create will never be used again in subsequent recursive calls, therefore the depth of the TLFG will be 2.

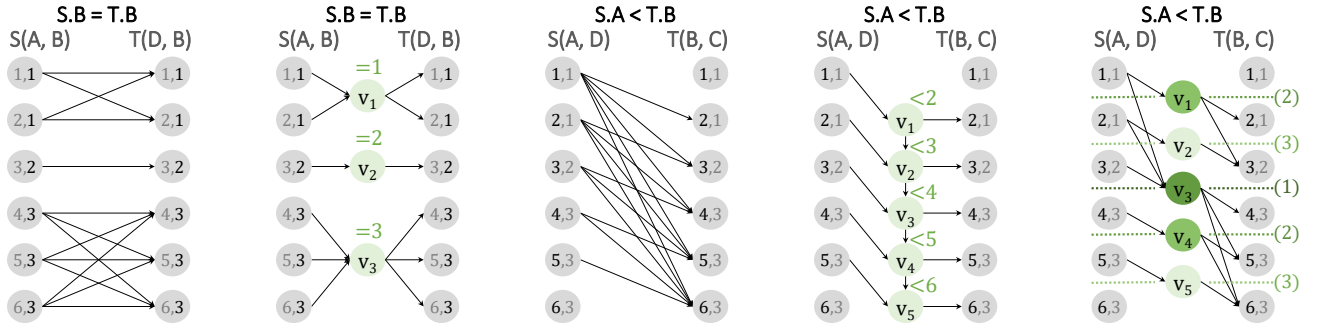
In all recursive steps we pick the *median of the distinct values* as our pivot. For multiset $\{1, 1, 1, 1, 2, 3, 3\}$ the set of distinct values is $\{1, 2, 3\}$ and hence the median is 2. This pivot is easy to find in $O(n)$ if the relations have been sorted on the appropriate attributes beforehand.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

Example 7. *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. Our approach for less-than ($<$) is straightforward to generalize to greater-than ($>$), since it is exactly symmetrical: We simply connect the partitions in the opposite direction, i.e., S_2 connects to T_1 (instead of S_1 to T_2). For inequality predicates with equality ($\leq, \geq$), only a minor change in the base case of the algorithm is needed: Instead of simply returning from the recursive call when only 1 distinct value remains, we connect all the source-target nodes that contain that (equal) value. This modification does not affect any of our guarantees.

Lemma 8. *Let θ be an inequality predicate for relations S, T of total size n. A duplicate-free TLFG of $S \bowtie_\theta T$ of size $O(n \log n)$ and depth 2 can be constructed in $O(n \log n)$ time.*

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



(a) Equality: naive construction with edges between all joining pairs. $O(n^2)$ size, $O(1)$ depth.

(b) Equality: grouping tuples with common join values together. $O(n)$ size, $O(1)$ depth.

(c) Inequality: naive construction with edges between all joining pairs. $O(n^2)$ size, $O(1)$ depth.

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

(e) Inequality: binary partitioning. Dotted lines indicate partitioning steps. $O(n \log n)$ size, $O(1)$ depth.

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.

4.2 Conjunctions

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

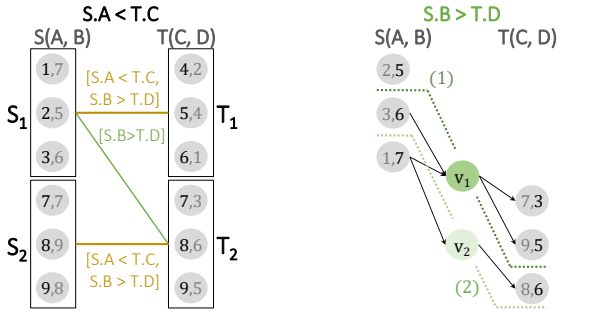
Equalities. First, consider the case where the conjunction contains equality predicates together the inequalities. As we saw before, equalities (irrespective of their number) create disjoint partitions of tuples (see Fig. 4b). Instead of connecting all the nodes within partition via an intermediate node as would be the case for an equi-join, we now have to take into account the inequality predicates and connect only those nodes that satisfy them. Thus, we first separate the disjoint partitions and then for each partition, we create a TLFG for our conjunction with the equality predicates removed.

Example 9. *Suppose $\theta \equiv (S.B = T.B) \wedge (S.A < T.D)$. We first process the equality predicate and then the inequality. For the example of Fig. 4b, the equality creates three disjoint partitions: (S_1, T_1) with value 1, (S_2, T_2) with value 2, and (S_3, T_3) with value 3. Rather than connecting each partition via an intermediate node as in the figure, we now have to construct three independent TLFGs for the inequality predicate $S.A < T.D$, one TLFG for each (S_i, T_i). Source-target nodes that are connected by that process will satisfy both predicates since they belong to the same equality partition, and were also connected by the inequality TLFG.*

Inequalities. We generalize the same idea to the case of multiple inequality predicates. To handle each inequality, we use the binary partitioning approach we developed in Section 4.1. Algorithm 1 shows the pseudocode of our approach. The two partitions are connected via an intermediate node only when no other predicates remain (Lines 15 to 17), otherwise they are passed on to the next predicate (Line 20). Overall, we perform two recursions simultaneously. In one direction, we make recursive calls on smaller partitions of the data and the same set of predicates (Lines 22 and 23). In the other direction, when the current predicate is satisfied for some source-target nodes, nextPredicate() is called with one less predicate (Line 20). The recursion stops either when we are left with 1 join value (base case for binary partitioning) or we exhaust the predicate list (base case for conjunction). Finally, notice that each time a new predicate is encountered, the nodes have to be sorted according to the new attributes (Line 6).

Example 10. *Consider two inequalities $S.A < T.C \wedge S.B > T.D$ for relations $S(A, B), T(C, D)$ as shown in Fig. 5a. The algorithm initially processes the first inequality and splits the relations into $(S_1, T_1), (S_2, T_2)$ as per the binary partitioning method (see Section 4.1). The recursive calls on these two partitions (depicted with horizontal edges) are made with the same list of predicates. While for one inequality S_1 and T_2 would be connected via intermediate node, we now make a third recursive call (depicted with a diagonal edge) that will process the next inequality $S.B > T.D$. The result of this recursive call is shown in Fig. 5b. Only some pairs of these nodes satisfy this second predicate and are eventually connected. Also notice that inside this recursive call, we had to sort on attributes B, D before using binary partitioning.*

Lemma 11. *Let θ be a conjunction of p inequality and any number of equality predicates for relations S, T of total size n. A duplicate-free TLFG of $S \bowtie_\theta T$ of size $O(n \log^p n)$ and depth 2 can be constructed in $O(n \log^p n)$ time.*



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

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).

Algorithm 1: Factorizing a conjunction of inequalities

```
Input: Relations S, T, nodes v_s, v_t for s ∈ S, t ∈ T,
       Conjunction θ as list of conditions [θ_1 | θ_L]
Output: A TLFG of the join S ⋈_θ T
nextPredicate (S, T, θ)
Procedure nextPredicate(S, T, θ = [θ_1 | θ_L])
    S', T' = S, T sorted by the attributes of θ_1
    if (θ_1 == S.A < T.B) then
        partIneqBinary (S', T', [θ_1 | θ_L])
Procedure partIneqBinary(S, T, [θ_1 = S.A < T.B | θ_L])
    δ = vals(S ∪ T) // Number of distinct A, B values
    if δ == 1 then return // Base case for binary partitioning
    Partition (S ∪ T) into (S_1 ∪ T_1), (S_2 ∪ T_2) with median distinct value
      as pivot
    if θ_L == [] then
        // Base case for #predicates: connect S_1 to T_2
        Materialize intermediate node x
        foreach s in S_1 do Create edge v_s ⟶ x
        foreach t in T_2 do Create edge x ⟶ v_t
    else
        // Check S_1 ⟶ T_2 against the rest of the predicates
        nextPredicate (S_1, T_2, θ_L)
    // Recursive calls on horizontal partitions, same predicates
    partIneqBinary (S_1, T_1, [θ_1 | θ_L])
    partIneqBinary (S_2, T_2, [θ_1 | θ_L])
```

4.3 Disjunctions

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

Lemma 12. *Let θ be a disjunction of predicates $\theta_1, \ldots, \theta_p$ for relations S, T. If for each $\theta_i, i \in [p]$ we can construct a duplicate-free TLFG of $S \bowtie_{\theta_i} T$ of size $O(\mathcal{S}_i)$ and depth d_i in $O(\mathcal{T}_i)$ time, then we can construct a TLFG of $S \bowtie_{\theta} T$ of size $O(\sum_i \mathcal{S}_i)$ and depth $\max_i d_i$ in $O(\sum_i O(\mathcal{T}_i))$ time. The duplication factor of the latter is at most p.*

5 IMPROVEMENTS AND EXTENSIONS

We propose improvements that lead to our main result: strong worst-case guarantees for $\mathrm{TT}(k)$ and $\mathrm{MEM}(k)$ for acyclic join queries with inequalities, which we then extend to cyclic joins.

5.1 Improved Factorization Methods

We explore how to reduce the size of the TLFG for inequalities. More details can be found in Appendices C to E.

Shared ranges. A simple inequality like $S.A < T.B$ can be encoded 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. Even though the resulting TLFG is of size $O(n)$, its depth is $O(n)$ as well. The latter causes a high delay between consecutive join answers. From Theorem 5 and the fact that we need to sort to construct the TLFG, we obtain $\mathrm{TT}(k) = O(n+k\log k+kn) = O(kn)$ and $\mathrm{MEM}(k) = O(n+kn) = O(kn)$. This improves over terms like $O(n\,\mathrm{polylog}\,n + k\log k)$ for small k, but since $k = O(n^\ell)$ for a join of ℓ relations, it can get much worse for large k. Our experiments confirm this analysis. Moreover, it is unclear how to generalize this idea to a conjunction of inequalities.

Multiway partitioning. When the θ_j predicate on an edge of the theta-join tree is a simple inequality, we propose to construct a smaller TLFG. The approach works like binary partitioning (Section 4.1), but it splits the tuples into $O(\sqrt{n})$ partitions (instead of 2) per step—hence the name *multiway partitioning*. Analogous to binary partitioning, we prove that this results in a TLFG of size $O(n\log\log n)$ (vs. $O(n\log n)$ for binary partitioning) and depth 3 (vs. 2). Unfortunately, like for shared ranges, 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 disjunction 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 θ be a simple inequality, non-equality, or band predicate for relations S, T of size $O(n)$. A duplicate-free TLFG for $S \bowtie_\theta T$ 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 $\mathrm{TT}(k) = O(n \log^p n + k \log k)$. *The space requirement is* $\mathrm{MEM}(k) = O(n \log^{p-1} n \cdot \log\log n + k)$.

5.3 Cyclic Queries

So far, we have focused only on acyclic queries, but our techniques are also applicable to cyclic queries with some modifications. Recall that acyclic 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 process the query as a cyclic one. We delineate two possible ways that cyclic queries can be handled and include an example of triangle query with inequalities in Appendix G.

Post-processing filter. An approach that is common in handling cyclic queries in practice is to ignore some cyclic predicates for join processing, and apply them as a selection filter on the output. Specifically, we can keep all predicates that can be assigned to some edge of the join tree, then apply the techniques of our paper on the resulting acyclic query, and finally use the remaining 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 $\mathrm{TT}(k) = O(n^\ell \log n)$ for an ℓ-relation acyclic join.

Transformation to equi-join. An alternative approach that gives non-trivial guarantees is to apply our equi-join transformation on the cyclic query, and then use existing algorithms for ranked enumeration of cyclic equi-joins [77]. We deal with the case where each θ_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 *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 we 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 [77].

6 EXPERIMENTS

We demonstrate the superiority of our approach for ranked enumeration against existing DBMS, 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 $S \bowtie_\theta T$ to equi-join $S \bowtie E \bowtie T$ via the edge set E of the TLFG. Then previous ranked-enumeration techniques for equi-joins [77] 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 DBMS. 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 $\ell = 2$ 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. (S) 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 $[0 \ldots 10^4 - 1]$ uniformly at random, discarding duplicate tuples. The weights W_i are real numbers drawn from $[0, 10^4)$. (T) 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)`.

(R) For real data, we use a temporal graph RedditTitles [51] 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)`. (B) OceaniaBirds [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 ℓ as a parameter; for those we only write the join conditions between the i^{th} and $(i+1)^{\text{st}}$ relations, with the rest similarly organized in a chain. In Appendix H 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 dates. Query Q_{R1} computes temporal paths [81] on RedditTitles, and ranks them by a measure of sentiment such that sequences of negative posts are retrieved first. Query Q_{R2} uses instead the sentiment in the join condition, keeping only paths along which the negative sentiment increases. For ranking, we use readability to focus on posts of higher quality. Last, Q_B is a spatial band join on OceaniaBirds that finds pairs of high-count bird sightings that are close based on proximity.

Query	Ranking
$Q_{S1}(\ldots) \coloneq S_1(A_1, A_2), S_2(A_3, A_4), \ldots, S_\ell(A_{2\ell-1}, A_{2\ell}), (A_{2i} < A_{2i+1})$	$\min(W_1 + W_2 + \ldots)$
$Q_{S2}(\ldots) \coloneq S_1(A_1, A_2), S_2(A_3, A_4), \ldots, S_\ell(A_{2\ell-1}, A_{2\ell}), (\lvert A_{2i} - A_{2i+1}\rvert < 50), (A_{2i-1} \neq A_{2i+2})$	$\min(W_1 + W_2 + \ldots)$
$Q_T(\ldots) \coloneq \text{Item}(O_1, PK_1, SK, L_1, Q_1, P_1, S_1, C_1, R_1), \text{Item}(O_2, PK_2, SK, L_2, Q_2, P_2, S_2, C_2, R_2), \ldots, (Q_i < Q_{i+1}), (S_i < S_{i+1})$	$\min(P_1 + P_2 + \ldots)$
$Q_{TD}(\ldots) \coloneq \text{Item}(O_1, PK_1, SK, L_1, Q_1, P_1, S_1, C_1, R_1), \text{Item}(O_2, PK_2, SK, L_2, Q_2, P_2, S_2, C_2, R_2), \ldots, (Q_i < Q_{i+1}), (S_i < S_{i+1} \vee C_i < C_{i+1} \vee R_i < R_{i+1})$	$\min(P_1 + P_2 + \ldots)$
$Q_{R1}(\ldots) \coloneq \text{Reddit}(N_1, N_2, T_1, S_1, R_1), \text{Reddit}(N_2, N_3, T_2, S_2, R_2), \ldots, (T_i < T_{i+1})$	$\min(S_1 + S_2 + \ldots)$
$Q_{R2}(\ldots) \coloneq \text{Reddit}(N_1, N_2, T_1, S_1, R_1), \text{Reddit}(N_2, N_3, T_2, S_2, R_2), \ldots, (T_i < T_{i+1}), (S_i > S_{i+1})$	$\max(R_1 + R_2 + \ldots)$
$Q_B(\ldots) \coloneq \text{Birds}(I_1, LA_1, LO_1, C_1), \text{Birds}(I_2, LA_2, LO_2, C_2) (\lvert LA_1 - LA_2\rvert < \varepsilon), (\lvert LO_1 - LO_2\rvert < \varepsilon)$	$\max(C_1 + C_2)$

Figure 6: Queries used in our experiments expressed in Datalog. The head always contains all body variables (no projections). Length ℓ of queries range from 2 to 10. Indices i range from 1 to $\ell - 1$.

Details. Our algorithms are implemented in Java 8 and executed on an Intel Xeon E5-2643 CPU running Ubuntu Linux. Queries execute in memory on a Java VM with 100GB of RAM. If that is exceeded, we report an Out-Of-Memory (OOM) error. The any-k algorithm used by Factorized and QuadEqui is Lazy [21, 77] which was found to outperform others in previous work. The version of PostgreSQL is 9.5.25. We set its parameters such that it is optimized for main-memory execution and system overhead related to logging or concurrency is minimized, as it is standard in the literature [10, 77]. To enable input caching for PSQL and System X, each execution is performed twice and we only time the second one. Additionally, we create B-tree or hash indexes for each attribute of the input relations, while our methods do not receive these indexes. Even though the task is ranked enumeration, we still give the database systems a LIMIT clause whenever we measure a specific TT(k), and thus allow them to leverage the k value. All data points we show are the median of 5 measurements. We timeout any execution that does not finish within 2 hours.

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.

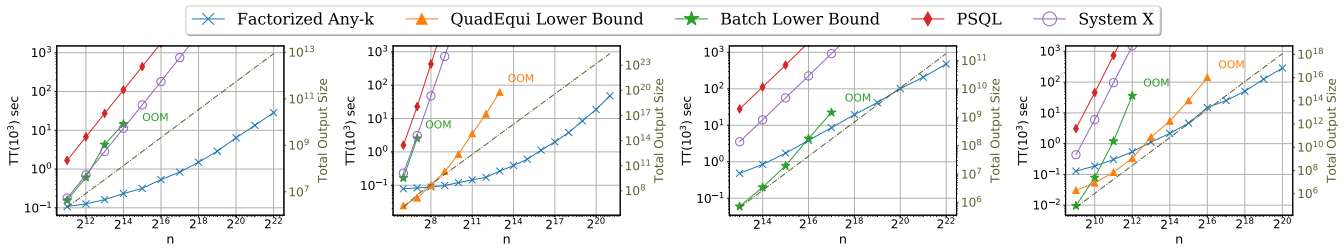
Summary. *① Factorized is superior when the total output size is large, even when compared against a lower bound of the running time of the other methods. ② QuadEqui and ③ Batch require significantly more memory and are infeasible for many queries. ④ PSQL and ⑤ System X, similarly to Batch, must produce 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_{S1}, Q_{S2} 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 $\sim 8k$ input size, while we can easily handle $\sim 2M$. For very small input sizes, the lower bounds of QuadEqui and Batch are sometimes lower, but their real running times are actually much higher than that. Q_{S2} has more join predicates and thus, a more restricted output size (Figures 7c and 7d). Our advantage is smaller in this case, yet still significant for large values of 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.

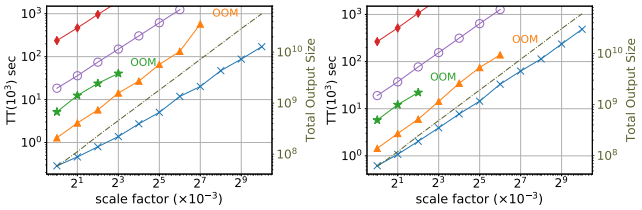
6.1.2 Effect of Query Length. Next, we test the effect of query length on RedditTitles. We plot TT(k) for three values ($k = 1, 10^3, 10^6$) when the length is small ($\ell = 2, 3$) and one value ($k = 10^3$) for longer queries. Note that for $k = 1$, the time of Factorized is essentially the time required for building our TLFGs, and doing a bottom-up Dynamic Programming pass [77]. 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 $k = 10^6$ it 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 $\ell = 3$, PSQL times out, 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 more restricted. Thus, the Batch lower bound is slightly better than our approach for $\ell = 2$ (Figure 9e), but we expect that this would not happen when the actual cost of materializing the output was taken into account. Still, for $\ell \geq 3$ (Figure 9g), our approach dominates even when compared against the lower bounds. PSQL again times out for $\ell = 3$ (Figure 9f), 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 OceaniaBirds dataset with various band widths ε. Figure 9d shows that Factorized is superior for all tested k values for $\varepsilon = 0.01$. Increasing the band width yields more joining pairs and causes the size of the output to grow (Figure 9h). Hence, Batch consumes more memory and cannot handle $\varepsilon \geq 0.16$. On the other hand, the performance of Factorized is mildly affected by increasing ε. PSQL and System X were not able to terminate within the time limit even for the smallest ε because they use only one of the indexes (for `Longitude`), searching over a huge number of possible results.



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

Figure 7: Section 6.1.1: Synthetic data with a growing database size n. While all four alternative methods either run out of memory ("OOM") or exceed a reasonable running time, our method scales quasilinearly ($O(n \operatorname{polylog} n)$) with n.



(a) Query Q_T, $\ell = 3$. (b) Query Q_{TD}, $\ell = 3$.

Figure 8: Section 6.1.1: TPC-H data with increasing scale factor. Disjunctions do not affect the scaling of our algorithm.

6.2 Comparison of our Variants

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

6.2.1 Delay and TT(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 TT(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 yields 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 Multiway Partitioning (recall that Multiway 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 given in an implementation-agnostic measure. As n increases there is an asymptotic difference between the three methods ($O(n \log n)$ vs $O(n \log \log n)$ vs $O(n)$) 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 often. This is expected since the intermediate nodes of our TLFG essentially represent ranges in the domain and are more compact for smaller domains. A phase transition around $n = 2^{16}$ occurs when the tuple values become entirely different.

7 RELATED WORK

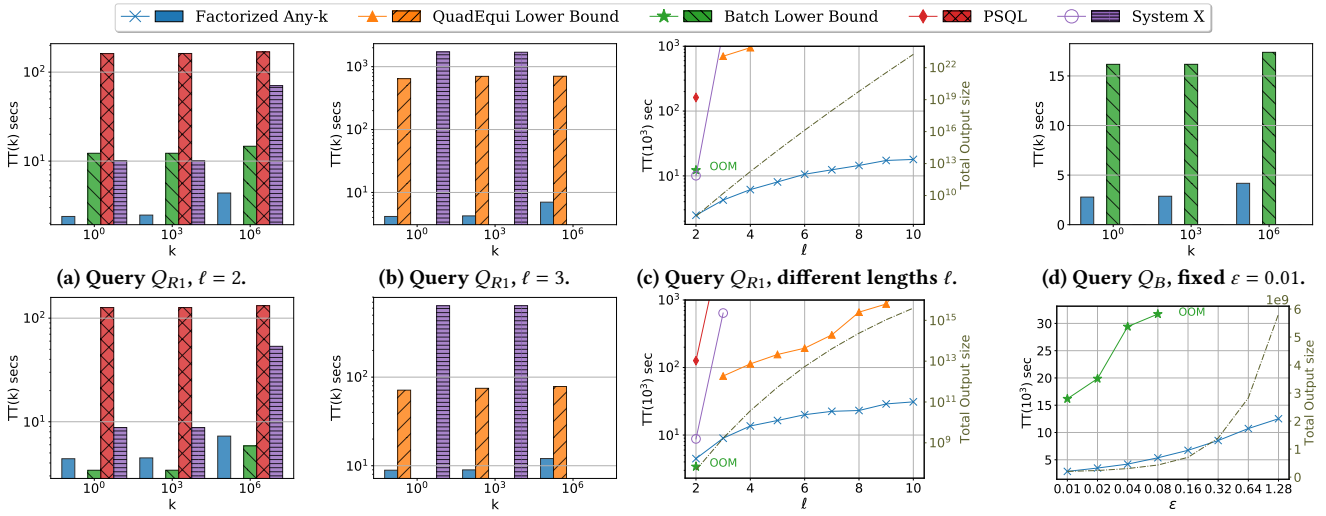
Enumeration for equi-joins. *Unranked* enumeration for equi-joins has been explored in various contexts [11, 12, 17, 18, 33, 74], with a landmark result showing for self-join-free equi-joins that linear preprocessing and constant delay are possible if and only if the query is free-connex acyclic [8, 14]. For the more demanding task of ranked enumeration, a logarithmic delay is unavoidable [16, 30]. Our recently proposed any-k algorithms represent the state of the art for ranked enumeration for equi-joins [77]. Other work in this space focuses on practical implementations [32] and direct access [19, 20] to output tuples.

Non-Equality ($\neq$) and inequality ($<$) joins. Techniques for batch-computation of the entire output for joins with *non-equality* (also called *inequality* [49] or *disequality* [8]) predicates mainly rely on variations of color coding [6, 49, 71]. The same core idea is leveraged by the unranked enumeration algorithm of Bagan et al. [8]. Queries with negation can be answered by rewriting them with *not-all-equal-predicates* [46], a generalization of non-equality.

Khayatt et al. [48] provide optimized and distributed *batch* algorithms for up to two inequalities per join. Handling inequalities efficiently typically requires indexes [43] that have a lookup time of $O(\log n)$ or higher, e.g. $O(\log^{p-1} n)$ for range trees [29] indexing $p > 1$ inequalities. In the same spirit, Khamis et al. [3] rely on Chazelle's data structures [23] for aggregate computation with inequality predicates.

We are the first to consider *ranked* enumeration for non-equality and inequality predicates, including DNF containing both types, and to prove strong worst-case guarantees for a large class of these queries. For previous work on batch computation and unranked enumeration, is not clear how to extend prior data structures to efficiently support ranking.

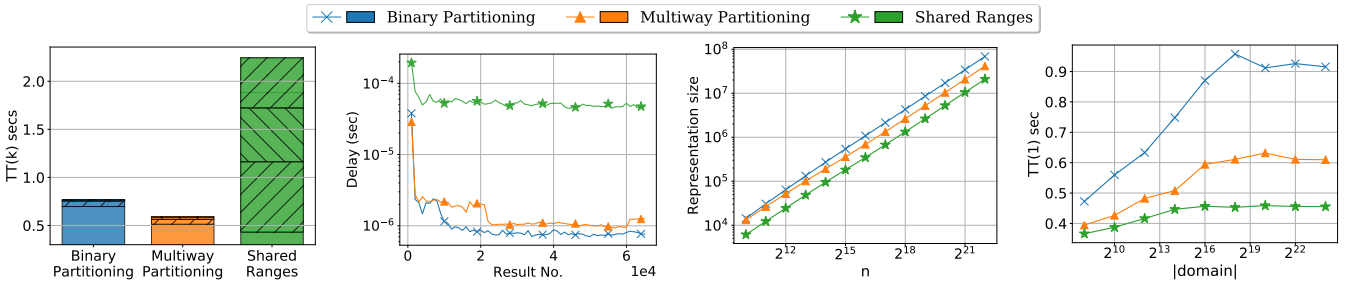
Factorized databases. Factorized representations of query results [9, 66] have been proposed for *equi-joins* in the context of enumeration [68, 69], aggregate computation [9], provenance management [54, 67, 68] and machine learning [4, 50, 65, 70, 73]. Our novel TLFG approach to factorization complements this line of research and extends the fundamental idea of factorization to ranked enumeration for theta-joins. For probabilistic databases, factorization of non-equalities [63] and inequalities [64] 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 [24].



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

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

Figure 9: Section 6.1.2: a,b,c,e,f,g: Section 6.1.3: d, h: Temporal paths of different lengths on RedditTitles (left), and spatial band-join on OceaniaBirds (right). Our method is robust to increasing query sizes and band-join ranges.



(a) $\mathrm{TT}(k)$ for $n = 2^{16}$. After pre-processing (no pattern), each bar represents 10^4 results (alternating pattern).

(b) Delay between the first n consecutive results for $n = 2^{16}$. The delay is averaged in a 10^3 window.

(c) Size of the constructed TLFG for increasing n. Measured as the total number of nodes and edges in the graph.

(d) Time to the first result for $n = 2^{16}$ and different domain sizes. The tuple values are sampled randomly.

Figure 10: Section 6.2: Comparing different aspects of our factorization methods on query Q_{S1}, $\ell = 2$.

Top-k queries. Top-k queries [72] 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. [35] 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 [36, 44, 55, 82], including theta-joins [57]. 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 [78]. Ilyas et al. [45] survey some of these approaches, along with some related ones such as building top-k indexes [22, 76] or views [27, 41].

Optimal batch algorithms for joins. Acyclic equi-joins are evaluated optimally in $O(n + |\text{out}|)$ by the Yannakakis algorithm [85], where $|\text{out}|$ is the output size. This bound is unattainable for cyclic queries [61], thus worst-case optimal join algorithms [58, 61, 62, 80] settle for the AGM bound [7], i.e., the worst-case output size. (Hyper)tree decomposition methods [5, 38, 56] can improve over these guarantees, while a geometric perspective has led to even stronger notions of optimality [47, 60]. Ngo [59] 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 DBMS 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 ($O(n \operatorname{polylog} n)$) 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 $O(n^\ell)$ 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.

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

REFERENCES

[1] 2020. Bird Occurrences in Oceania. https://doi.org/10.15468/dl.d6u6tj From https://www.gbif.org/.

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

[3] Mahmoud Abo Khamis, Ryan R. Curtin, Benjamin Moseley, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2019. On Functional Aggregate Queries with Additive Inequalities. In *PODS*. 414–431. https://doi.org/10.1145/3294052.3319694

[4] Mahmoud Abo Khamis, Hung Q Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2018. In-database learning with sparse tensors. In *PODS*. 325–340. https://doi.org/10.1145/3196959.3196960

[5] Mahmoud Abo Khamis, Hung Q Ngo, and Dan Suciu. 2017. What do Shannon-type Inequalities, Submodular Width, and Disjunctive Datalog have to do with one another?. In *PODS*. 429–444. https://doi.org/10.1145/3034786.3056105

[6] Noga Alon, Raphael Yuster, and Uri Zwick. 1995. Color-coding. *J. ACM* 42, 4 (1995), 844–856. https://doi.org/10.1145/210332.210337

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

[8] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. 2007. On acyclic conjunctive queries and constant delay enumeration. In *International Workshop on Computer Science Logic (CSL)*. 208–222. https://doi.org/10.1007/978-3-540-74915-8_18

[9] Nurzhan Bakibayev, Tomáš Kočiský, Dan Olteanu, and Jakub Závodný. 2013. Aggregation and Ordering in Factorised Databases. *PVLDB* 6, 14 (2013), 1990–2001. https://doi.org/10.14778/2556549.2556579

[10] Nurzhan Bakibayev, Dan Olteanu, and Jakub Závodný. 2012. FDB: A Query Engine for Factorised Relational Databases. *PVLDB* 5, 11 (2012), 1232–1243. https://doi.org/10.14778/2350229.2350242

[11] Christoph Berkholz, Fabian Gerhardt, and Nicole Schweikardt. 2020. Constant Delay Enumeration for Conjunctive Queries: A Tutorial. *ACM SIGLOG News* 7, 1 (2020), 4–33. https://doi.org/10.1145/3385634.3385636

[12] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. 2017. Answering Conjunctive Queries Under Updates. In *PODS*. 303–318. https://doi.org/10.1145/3034786.3034789

[13] Christoph Berkholz and Nicole Schweikardt. 2019. Constant Delay Enumeration with FPT-Preprocessing for Conjunctive Queries of Bounded Submodular Width. In *44th International Symposium on Mathematical Foundations of Computer Science (MFCS) (LIPIcs)*, Vol. 138. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 58:1–58:15. https://doi.org/10.4230/LIPIcs.MFCS.2019.58

[14] Johann Brault-Baron. 2013. *De la pertinence de l'énumération: complexité en logiques propositionnelle et du premier ordre*. Ph.D. Dissertation. Université de Caen. https://hal.archives-ouvertes.fr/tel-01081392

[15] Johann Brault-Baron. 2016. Hypergraph Acyclicity Revisited. *ACM Comput. Surv.* 49, 3, Article 54 (Dec. 2016), 26 pages. https://doi.org/10.1145/2983573

[16] David Bremner, Timothy M Chan, Erik D Demaine, Jeff Erickson, Ferran Hurtado, John Iacono, Stefan Langerman, and Perouz Taslakian. 2006. Necklaces, convolutions, and X+ Y. In *European Symposium on Algorithms*. Springer, 160–171. https://doi.org/10.1007/s00453-012-9734-3

[17] Nofar Carmeli and Markus Kröll. 2019. On the Enumeration Complexity of Unions of Conjunctive Queries. In *PODS*. 134–148. https://doi.org/10.1145/3294052.3319700

[18] Nofar Carmeli and Markus Kröll. 2020. Enumeration Complexity of Conjunctive Queries with Functional Dependencies. *Theory Comput. Syst.* 64, 5 (2020), 828–860. https://doi.org/10.1007/s00224-019-09937-9

[19] Nofar Carmeli, Nikolaos Tziavelis, Wolfgang Gatterbauer, Benny Kimelfeld, and Mirek Riedewald. 2020. Tractable Orders for Direct Access to Ranked Answers of Conjunctive Queries. *CoRR* abs/2012.11965 (2020). arXiv:2012.11965

[20] Nofar Carmeli, Shai Zeevi, Christoph Berkholz, Benny Kimelfeld, and Nicole Schweikardt. 2020. Answering (Unions of) Conjunctive Queries Using Random Access and Random-Order Enumeration. In *PODS*. 393–409. https://doi.org/10.1145/3375395.3387662

[21] Lijun Chang, Xuemin Lin, Wenjie Zhang, Jeffrey Xu Yu, Ying Zhang, and Lu Qin. 2015. Optimal enumeration: Efficient top-k tree matching. *PVLDB* 8, 5 (2015), 533–544. https://doi.org/10.14778/2735479.2735486

[22] Yuan-Chi Chang, Lawrence Bergman, Vittorio Castelli, Chung-Sheng Li, Ming-Ling Lo, and John R Smith. 2000. The onion technique: indexing for linear optimization queries. In *SIGMOD*. 391–402. https://doi.org/10.1145/342009.335433

[23] Bernard Chazelle. 1988. Functional approach to data structures and its use in multidimensional searching. *SIAM J. Comput.* 17, 3 (1988), 427–462. https://doi.org/10.1137/0217026

[24] Sophie Cluet and Guido Moerkotte. 1995. Efficient evaluation of aggregates on bulk types. In *Proceedings of the Fifth International Workshop on Database Programming Languages 5*. 1–10. https://doi.org/10.14236/ewic/DBPL1995.6

[25] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3rd ed.). The MIT Press. https://dl.acm.org/doi/book/10.5555/1614191

[26] Yves Crama and Peter L Hammer. 2011. *Boolean functions: Theory, algorithms, and applications*. Cambridge University Press. https://doi.org/10.1017/CBO9780511852008

[27] Gautam Das, Dimitrios Gunopulos, Nick Koudas, and Dimitris Tsirogiannis. 2006. Answering top-k queries using views. In *VLDB*. 451–462. https://dl.acm.org/doi/10.5555/1182635.1164167

[28] Sanjoy Dasgupta, Christos H Papadimitriou, and Umesh Virkumar Vazirani. 2008. *Algorithms*. McGraw-Hill Higher Education. https://dl.acm.org/doi/book/10.5555/1177299

[29] Mark De Berg, Marc Van Kreveld, Mark Overmars, and Otfried Schwarzkopf. 1997. Computational geometry. In *Computational geometry*. Springer, 1–17. https://doi.org/10.1007/978-3-540-77974-2

[30] Shaleen Deep and Paraschos Koutris. 2021. Ranked Enumeration of Conjunctive Query Results. In *ICDT*, Vol. 186. 5:1–5:19. https://doi.org/10.4230/LIPIcs.ICDT.2021.5

[31] David J. DeWitt, Jeffrey F. Naughton, and Donovan A. Schneider. 1991. An Evaluation of Non-Equijoin Algorithms. In *VLDB*. 443–452. https://dl.acm.org/doi/10.5555/645917.672320

[32] Mengsu Ding, Shimin Chen, Nantia Makrynioti, and Stefan Manegold. 2021. Progressive Join Algorithms Considering User Preference. In *CIDR*. https://ir.cwi.nl/pub/30501/30501.pdf

[33] Arnaud Durand. 2020. Fine-Grained Complexity Analysis of Queries: From Decision to Counting and Enumeration. In *PODS*. 331–346. https://doi.org/10.1145/3375395.3389130

[34] Jost Enderle, Matthias Hampel, and Thomas Seidl. 2004. Joining Interval Data in Relational Databases. In *SIGMOD*. 683–694. https://doi.org/10.1145/1007568.1007645

[35] Ronald Fagin, Amnon Lotem, and Moni Naor. 2003. Optimal aggregation algorithms for middleware. *J. Comput. System Sci.* 66, 4 (2003), 614–656. https://doi.org/10.1016/S0022-0000(03)00026-6

[36] Jonathan Finger and Neoklis Polyzotis. 2009. Robust and efficient algorithms for rank join evaluation. In *SIGMOD*. 415–428. https://doi.org/10.1145/1559845.1559890

[37] Michel Gondran and Michel Minoux. 2008. *Graphs, Dioids and Semirings: New Models and Algorithms (Operations Research/Computer Science Interfaces Series)*. Springer. https://doi.org/10.1007/978-0-387-75450-5

[38] Georg Gottlob, Gianluigi Greco, Nicola Leone, and Francesco Scarcello. 2016. Hypertree Decompositions: Questions and Answers. In *PODS*. 57–74. https://doi.org/10.1145/2902251.2902309

[39] M.H. Graham. 1979. *On the universal relation*. Technical Report. Univ. of Toronto.

[40] C. A. R. Hoare. 1962. Quicksort. *Comput. J.* 5, 1 (01 1962), 10–16. https://doi.org/10.1093/comjnl/5.1.10

[41] Vagelis Hristidis, Nick Koudas, and Yannis Papakonstantinou. 2001. PREFER: A system for the efficient execution of multi-parametric ranked queries. *SIGMOD Record* 30, 2 (2001), 259–270. https://doi.org/10.1145/375663.375690

[42] Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. 2019. Efficient Query Processing for Dynamically Changing Datasets. *SIGMOD Record* 48, 1 (2019), 33–40. https://doi.org/10.1145/3371316.3371325

[43] Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. 2020. General dynamic Yannakakis: conjunctive queries with theta joins under updates. *VLDB J.* 29 (2020), 619–653. https://doi.org/10.1007/s00778-019-00590-9

[44] Ihab F Ilyas, Walid G Aref, and Ahmed K Elmagarmid. 2004. Supporting top-k join queries in relational databases. *VLDB J.* 13, 3 (2004), 207–221. https://doi.org/10.1007/s00778-004-0128-2

[45] Ihab F Ilyas, George Beskales, and Mohamed A Soliman. 2008. A survey of top-k query processing techniques in relational database systems. *Comput. Surveys* 40, 4 (2008), 11. https://doi.org/10.1145/1391729.1391730

[46] Mahmoud Abo Khamis, Hung Q. Ngo, Dan Olteanu, and Dan Suciu. 2019. Boolean Tensor Decomposition for Conjunctive Queries with Negation. In *ICDT*. 21:1–21:19. https://doi.org/10.4230/LIPIcs.ICDT.2019.21

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

[48] Zuhair Khayyat, William Lucia, Meghna Singh, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Panos Kalnis. 2017. Fast and scalable inequality joins. *VLDB J.* 26, 1 (2017), 125–150. https://doi.org/10.1007/s00778-

016-0441-6
[49] Paraschos Koutris, Tova Milo, Sudeepa Roy, and Dan Suciu. 2017. Answering Conjunctive Queries with Inequalities. *Theory of Computing Systems* 61, 1 (2017), 2–30. https://doi.org/10.1007/s00224-016-9684-2
[50] Arun Kumar, Jeffrey Naughton, and Jignesh M Patel. 2015. Learning generalized linear models over normalized data. In *SIGMOD*. 1969–1984. https://doi.org/10.1145/2723372.2723713
[51] Srijan Kumar, William L Hamilton, Jure Leskovec, and Dan Jurafsky. 2018. Community interaction and conflict on the web. https://snap.stanford.edu/data/soc-RedditHyperlinks.html. In *WWW*. 933–943.
[52] Rundong Li, Wolfgang Gatterbauer, and Mirek Riedewald. 2020. Near-Optimal Distributed Band-Joins through Recursive Partitioning. In *SIGMOD*. 2375–2390. https://doi.org/10.1145/3318464.3389750
[53] Qingyun Liu, Jack W. Stokes, Rob Mead, Tim Burrell, Ian Hellen, John Lambert, Andrey Marochko, and Weidong Cui. 2018. Latte: Large-Scale Lateral Movement Detection. In *MILCOM*. 1–6. https://doi.org/10.1109/MILCOM.2018.8599748
[54] Neha Makhija and Wolfgang Gatterbauer. 2021. Towards a Dichotomy for Minimally Factorizing the Provenance of Self-Join Free Conjunctive Queries. *CoRR* abs/2105.14307 (2021). arXiv:2105.14307 https://arxiv.org/abs/2105.14307
[55] Nikos Mamoulis, Man Lung Yiu, Kit Hung Cheng, and David W Cheung. 2007. Efficient top-k aggregation of ranked inputs. *TODS* 32, 3 (2007), 19. https://doi.org/10.1145/1272743.1272749
[56] Dániel Marx. 2013. Tractable Hypergraph Properties for Constraint Satisfaction and Conjunctive Queries. *J. ACM* 60, 6, Article 42 (2013), 51 pages. https://doi.org/10.1145/2535926
[57] Apostol Natsev, Yuan-Chi Chang, John R Smith, Chung-Sheng Li, and Jeffrey Scott Vitter. 2001. Supporting incremental join queries on ranked inputs. In *VLDB*. 281–290. http://www.vldb.org/conf/2001/P281.pdf
[58] Gonzalo Navarro, Juan L. Reutter, and Javiel Rojas-Ledesma. 2020. Optimal Joins Using Compact Data Structures. In *ICDT*, Vol. 155. 21:1–21:21. https://doi.org/10.4230/LIPIcs.ICDT.2020.21
[59] Hung Q Ngo. 2018. Worst-case optimal join algorithms: Techniques, results, and open problems. In *PODS*. 111–124. https://doi.org/10.1145/3196959.3196990
[60] Hung Q Ngo, Dung T Nguyen, Christopher Re, and Atri Rudra. 2014. Beyond worst-case analysis for joins with minesweeper. In *PODS*. 234–245. https://doi.org/10.1145/2594538.2594547
[61] Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-case optimal join algorithms. *J. ACM* 65, 3 (2018), 16. https://doi.org/10.1145/3180143
[62] Hung Q Ngo, Christopher Ré, and Atri Rudra. 2014. Skew Strikes Back: New Developments in the Theory of Join Algorithms. *SIGMOD Record* 42, 4 (Feb. 2014), 5–16. https://doi.org/10.1145/2590989.2590991
[63] Dan Olteanu and Jiewen Huang. 2008. Using OBDDs for efficient query evaluation on probabilistic databases. (2008), 326–340. https://doi.org/10.1007/978-3-540-87993-0_26
[64] Dan Olteanu and Jiewen Huang. 2009. Secondary-storage confidence computation for conjunctive queries with inequalities. In *SIGMOD*. 389–402. https://doi.org/10.1145/1559845.1559887
[65] Dan Olteanu and Maximilian Schleich. 2016. F: Regression Models over Factorized Views. *PVLDB* 9, 13 (2016), 1573–1576. https://doi.org/10.14778/3007263.3007312
[66] Dan Olteanu and Maximilian Schleich. 2016. Factorized databases. *SIGMOD Record* 45, 2 (2016). https://doi.org/10.1145/3003665.3003667
[67] Dan Olteanu and Jakub Závodnỳ. 2011. On factorisation of provenance polynomials. In *TaPP*. https://www.usenix.org/conference/tapp11/factorisation-provenance-polynomials
[68] Dan Olteanu and Jakub Závodnỳ. 2012. Factorised representations of query results: size bounds and readability. In *ICDT*. 285–298. https://doi.org/10.1145/2274576.2274607
[69] Dan Olteanu and Jakub Závodnỳ. 2015. Size bounds for factorised representations of query results. *TODS* 40, 1 (2015), 2. https://doi.org/10.1145/2656335
[70] Krishna Kumar P., Paul Langton, and Wolfgang Gatterbauer. 2020. Factorized Graph Representations for Semi-Supervised Learning from Sparse Data. In *SIGMOD*. 1383–1398. https://doi.org/10.1145/3318464.3380577
[71] Christos H. Papadimitriou and Mihalis Yannakakis. 1999. On the complexity of database queries. *J. Comput. System Sci.* 58, 3 (1999), 407–427. https://doi.org/10.1006/jcss.1999.1626
[72] Saladi Rahul and Yufei Tao. 2019. A Guide to Designing Top-k Indexes. *SIGMOD Record* 48, 2 (2019). https://doi.org/10.1145/3377330.3377332
[73] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. 2016. Learning linear regression models over factorized joins. In *SIGMOD*. 3–18. https://doi.org/10.1145/2882903.2882939
[74] Luc Segoufin. 2015. Constant Delay Enumeration for Conjunctive Queries. *SIGMOD Record* 44, 1 (2015), 10–17. https://doi.org/10.1145/2783888.2783894
[75] Robert E Tarjan and Mihalis Yannakakis. 1984. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.* 13, 3 (1984), 566–579. https://doi.org/10.1137/0213035
[76] Panayiotis Tsaparas, Themistoklis Palpanas, Yannis Kotidis, Nick Koudas, and Divesh Srivastava. 2003. Ranked join indices. In *ICDE*. IEEE, 277–288. https://doi.org/10.1109/ICDE.2003.1260799
[77] Nikolaos Tziavelis, Deepak Ajwani, Wolfgang Gatterbauer, Mirek Riedewald, and Xiaofeng Yang. 2020. Optimal Algorithms for Ranked Enumeration of Answers to Full Conjunctive Queries. *PVLDB* 13, 9 (2020), 1582–1597. https://doi.org/10.14778/3397230.3397250
[78] Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. 2020. Optimal Join Algorithms Meet Top-k. In *SIGMOD*. 2659–2665. https://doi.org/10.1145/3318464.3383132
[79] Moshe Y. Vardi. 1982. The Complexity of Relational Query Languages (Extended Abstract). In *STOC*. 137–146. https://doi.org/10.1145/800070.802186
[80] Todd L. Veldhuizen. 2014. Triejoin: A Simple, Worst-Case Optimal Join Algorithm. In *ICDT*. 96–106. https://doi.org/10.5441/002/icdt.2014.13
[81] Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. 2014. Path Problems in Temporal Graphs. *PVLDB* 7, 9 (2014), 721–732. https://doi.org/10.14778/2732939.2732945
[82] Minji Wu, Laure Berti-Equille, Amélie Marian, Cecilia M Procopiuc, and Divesh Srivastava. 2010. Processing top-k join queries. *PVLDB* 3, 1 (2010), 860–870. https://doi.org/10.14778/1920841.1920951
[83] Xiaofeng Yang, Deepak Ajwani, Wolfgang Gatterbauer, Patrick K Nicholson, Mirek Riedewald, and Alessandra Sala. 2018. Any-k: Anytime Top-k Tree Pattern Retrieval in Labeled Graphs. In *WWW*. 489–498. https://doi.org/10.1145/3178876.3186115
[84] Xiaofeng Yang, Mirek Riedewald, Rundong Li, and Wolfgang Gatterbauer. 2018. Any-k Algorithms for Exploratory Analysis with Conjunctive Queries. In *International Workshop on Exploratory Search in Databases and the Web (ExploreDB)*. 1–3. https://doi.org/10.1145/3214708.3214711
[85] Mihalis Yannakakis. 1981. Algorithms for Acyclic Database Schemes. In *VLDB*. 82–94. https://dl.acm.org/doi/10.5555/1286831.1286840
[86] Clement Tak Yu and Meral Z Ozsoyoglu. 1979. An algorithm for tree-query membership of a distributed query. In *COMPSAC*. IEEE, 306–312. https://doi.org/10.1109/CMPSAC.1979.762509

Algorithm 2: Multiway partitioning

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

A NOMENCLATURE

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

B DELAY VS $\mathrm{TT}(k)$ AS COMPLEXITY MEASURE

In this section, we discuss the relationship between delay and $\mathrm{TT}(k)$ as complexity measures for enumeration. For unranked enumeration, our goal is to achieve $\mathrm{TT}(k) = O(\mathcal{P}(n) + k)$ with the lowest possible preprocessing time $\mathcal{P}(n)$. The majority of papers on enumeration [8, 17, 43, 74] have traditionally focused instead on *constant delay* after $\mathcal{P}(n)$ preprocessing. This is desirable because it implies the same guarantee $\mathrm{TT}(k) = O(\mathcal{P}(n))+k\cdot O(1) = O(\mathcal{P}(n)+k)$. However, setting constant delay as the goal can lead to misjudgments about practical performance, as we illustrate next:

Example 15. *Consider an enumeration problem where the output consists of the integers $1, 2, \ldots, n$, but algorithms produce duplicates that have to be filtered out on-the-fly. Assume that two algorithms $\mathcal{A}$ and $\mathcal{B}$ spend preprocessing $\mathcal{P}(n)$, then generate a sequence of results with constant delay. For $\mathcal{A}$, let this sequence be $1, 2, \ldots, n/2, 1, 2, \ldots, n/2, n/2+1, \ldots$ and for $\mathcal{B}$ it is $1, 1, 2, 2, \ldots, n/2, n/2, n/2+1, \ldots$ (see Fig. 11). Even though both achieve $\mathrm{TT}(k) = O(\mathcal{P}(n) + k)$, due to duplicate filtering the worst-case delay of $\mathcal{A}$ is $O(n)$ (between $n/2$ and $n/2+1$), while $\mathcal{B}$ has $O(1)$ delay. However, B is clearly* slower *than A by a factor of 2 for all $k \in [n/2]$. Since $\mathcal{A}$ outputs all these values earlier than $\mathcal{B}$, we could make $\mathcal{A}$* simulate the delay *of $\mathcal{B}$ for $k \in [n/2]$ by storing the computed values on even iterations and returning them later.*

As the example illustrates, for a preprocessing cost of $O(\mathcal{P}(n))$, the ultimate goal is to guarantee $\mathrm{TT}(k) = O(\mathcal{P}(n) + k)$. Constant-delay enumeration is a sufficient condition for achieving this goal, but not necessary. Similarly, for ranked enumeration, we aim for $\mathrm{TT}(k) = O(\mathcal{P}(n) + k \log k)$.

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 ρ 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 $\Theta(\sqrt{d})$, so that the connections between them can be built in $O(\sqrt{d}^2) = O(n)$, i.e., the same that binary partitioning needs per recursive step. The advantage of the multiple partitions is that we can reach the base case $d = 1$ faster since each partition is smaller. Algorithm 2 shows the pseudocode of this approach.

Lemma 16. *Let θ be an inequality predicate between relations S, T of total size n. A duplicate-free TLFG of the join $S \bowtie_\theta T$ of size $O(n \log\log n)$ and depth 3 can be constructed in $O(n \log n)$ time.*

Proof. The arguments for correctness and the duplicate-free property are similar to the case of binary partitioning (Lemma 8). 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

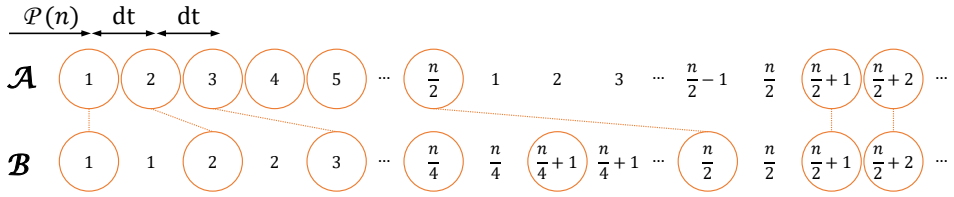
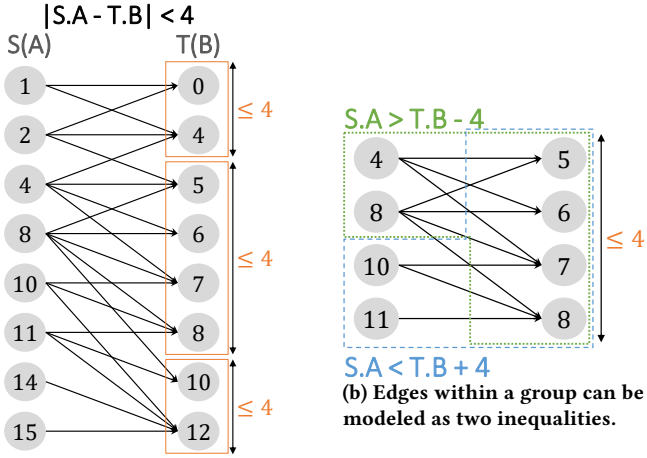


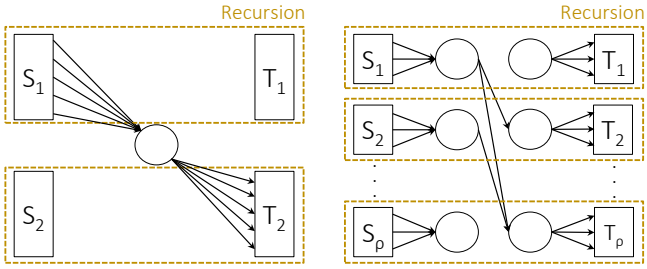
Figure 11: Two enumeration algorithms with $\mathrm{TT}(k) = O(\mathcal{P}(n) + k)$.



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

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

Figure 13: Example 18: TLFG construction for band conditions.



(a) 2 partitions: depth 2 with one intermediate node.

(b) ρ partitions: depth 3 with 2ρ intermediate nodes.

Figure 12: Binary vs Multi-way partitioning for inequalities.

relations. Then, we materialize $O(n)$ edges for source and target nodes, $O(\sqrt{\delta})$ intermediate nodes and $O(\sqrt{\delta}^2)$ edges between them. This adds up to $O(n)$ because $\delta \leq n$. We then invoke $\rho = \lceil\sqrt{\delta}\rceil = O(\sqrt{n})$ recursive calls with sizes $n_1 + n_2 + \ldots + n_\rho = n$. Therefore, in every level of the recursion tree, the sizes of all the subproblems add up to n. Since we spend linear time per problem, the total work per level of the recursion tree is $O(n)$. The height h of the tree is the number of times we have to take the square root of δ (and then the ceil function) in order to reach $d = 1$, which is $O(\log\log\delta) = O(\log\log n)$. To see this, observe that $d^{(\frac{1}{2})^h} = 2 \Rightarrow (\frac{1}{2})^h \log\delta = 1 \Rightarrow h = \log\log\delta$. Overall, the time spent on the recursion and thus, the size of the TLFG is bounded by $O(n\log\log n)$. □

D NON-EQUALITY PREDICATES

A non-equality condition $S.A \neq T.B$ is satisfied if either $S.A < T.B$ or $S.A > T.B$. Even though it can be modeled as a disjunction of two inequalities, we now establish that (in contrast to arbitrary disjunctions), they do not increase the TLFG duplication factor. The main observation is that the pairs which satisfy one of the inequalities cannot satisfy the other one. Therefore, if we union the two inequality TLFGs no path will be duplicated. The guarantees we obtain are the same as the inequality case by using multiway partitioning (once for each inequality).

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

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

E BAND PREDICATES

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

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

The first step is to sort the input relations and group the tuples of the target relation into maximal ε-intervals. More specifically, we start from the first T tuple and group together all those whose B values are at most ε apart from it. We then repeat the same process starting from the T tuple that is immediately after the group, creating $m \leq n$ groups, whose range of B values is at most

Algorithm 3: Handling a band predicate

```
1  Input: Relations S, T, nodes v_s, v_t for s ∈ S, t ∈ T,
2         predicate θ ≡ |S.A − T.B| < ε
3  Output: A TLFG of the join S ⋈_θ T
4  Sort S, T according to attributes A, B
5  foreach (S_b, T_b, θ_b) in bandToIneq (S, T, θ) do
6  |  partIneqMulti(S_b, T_b, θ_b)
7  Function bandToIneq(S, T, θ)
8  |  ineqs = []
9  |  //Find the limits of the groups on the right
10 |  H_1.start = t_1.B,  m = 1
11 |  for i ← 1 to |T| do
12 |  |  if t[i].B > H_m.start + ε then
13 |  |  |  H_m.end = T[i − 1].B
14 |  |  |  m++
15 |  |  |  H_m.start = T[i].B
16 |  H_m.end = T[i].B
17 |  foreach H_j in [H_1, ..., H_m] do
18 |  |  //Assign tuples to the group
19 |  |  S_j = [s ∈ S | H_j.start − ε ≤ s.A ≤ H_j.end + ε]
20 |  |  T_j = [t ∈ T | H_j.start ≤ t.B ≤ H_j.end]
21 |  |  //Greater-than inequality
22 |  |  S_> = [s ∈ S_j | s.A < H_j.start + ε]
23 |  |  ineqs.add((S_<, T_j, S.A > T.B − ε))
24 |  |  //Less-than inequality
25 |  |  ineqs.add((S_j − S_>, T_j, S.A < T.B + ε))
26 |  |  return ineqs
```

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

Example 18. *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 ε grows together with the domain size. Initially, we group the target tuples by ε 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 ε. Case (1) can be handled as a greater-than TLFG, case (2) as a less-than, and case (3) as either one of them. As Algorithm 3 shows, `partIneqMulti()` is called twice for each group.

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

Proof. First, we create disjoint T groups based on ε-intervals and assign each S tuple to all groups where it has joining partners (Lines 9 to 16). This can be done with binary search in $O(n \log n)$. Each T tuple is assigned to a single group. An S tuple cannot be assigned to more than three consecutive groups since their values span a range of at least 2ε. Within each group $H_j = (S_j \cup T_j)$, the correctness of our algorithm follows from the fact that the T_j tuples are at most ε apart on the B attribute. Since all the assigned S_j tuples have at least one joining partner in T_j, they have to join either with the first T_j tuple (in sorted B order) or with the last one. Recall that the band condition can be rewritten as $(S.A < T.B + \varepsilon) \wedge (S.A > T.B - \varepsilon)$, i.e., two inequality conditions that both have to be satisfied. In case some $s \in S_j$ joins with the first T_j tuple, then we know that the less-than condition is always satisfied for s within the group H_j. Thus, we just need to connect v_s with all v_t for $t \in T_j$ that satisfy the greater-than condition. We argue similarly for the case when s joins with the last tuple of T_j, where we have to take care only of the less-than condition. Finally, there is also the possibility that s joins with all T_j tuples. In that case, both inequality conditions are satisfied – we assign those tuples to only one of the inequalities which ensures the duplicate-free property. For the running time, the total size of the groups we create is $n_1 + n_2 + \ldots + n_m \leq 3n$. If for a problem of size $|S| + |T| = n$ where the relations have been sorted, $\mathcal{T}_B(n)$ is the time for factorizing a band condition and $\mathcal{T}_I(n)$ for an inequality, we have $\mathcal{T}_B(n) = O(n) + 2\mathcal{T}_I(n_1) + 2\mathcal{T}_I(n_2) + \ldots + 2\mathcal{T}_I(n_m)$, since we call the inequality algorithm twice within each group. For $\mathcal{T}_I(n) = O(n \log\log n)$, we get $\mathcal{T}_B(n) = O(n \log\log n)$, which also bounds the size of the TLFG. Each call to the inequality algorithm involves different S, T pairs, giving us the duplicate-free property and the same depth as the inequality TLFG. □

F ADDITIONAL PROOFS

F.1 Proof of Theorem 5

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 ℓ, which is considered to be constant. The theorem follows by applying Lemma 2 on the resulting enumeration graph.

F.2 Proof of Lemma 8

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 20. *Let θ be a conjunction of predicates between relations S, T of total size n, and θ' be that conjunction with all the equality predicates removed. If for S', T' with $|S'|+|T'| = n'$ we can construct a TLFG of the join $S' \bowtie_{\theta'} T'$ of size $O(f(n'))$, depth 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 θ' through some algorithm $\mathcal{A}$. If $\mathcal{A}$ elects to connect two nodes, then they satisfy both θ', and also the equalities since they belong to the same partition. Conversely, two nodes that remain disconnected at the end of the process either do not belong to the same equality partition or were not connected by $\mathcal{A}$, thus do not satisfy θ'.

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

F.4 Proof of Lemma 11

As a first step, all the equality predicates are handled by Lemma 20. Since the time and size guarantees we show are $O(n \log^p n)$ and $n \log^p n$ is a superadditive function, they are unaffected by this step. The remaining inequality predicates are handled by Algorithm 1. We denote by $\mathcal{T}_I(n, p)$ the running time for n tuples and p inequality predicates. We proceed by induction on the number of predicates p to show that $\mathcal{T}_I(n, p) \leq f(p) n \log^p n$ for some function f and sufficiently large n. First, assume that all the predicates are inequalities. For the base case $p = 1$, the analysis is the same as in the proof of Lemma 8: Tthe height of the recursion tree is $O(\log n)$ and the total time is $O(n \log n)$ together with sorting once. In other words, we have $\mathcal{T}_I(n, 1) \leq cn \log n$ for sufficiently large n. For the inductive step, we assume that $\mathcal{T}_I(n, p-1) \leq f(p-1) n \log^{p-1} n$. The inequality at the head of the list creates a recursion tree where every node has a subset of the tuples n' and calls the next inequality, thus is computed in $\mathcal{T}_I(n', p-1)$. The problem sizes in some level of the tree add up to $n_1 + \ldots + n_\rho = n$. Thus, the work per level is bounded by $\mathcal{T}_I(n_1, p-1) + \ldots + \mathcal{T}_I(n_t, p-1) \leq f(p-1) n_1 \log^{p-1} n_1 + \ldots + f(p-1) n_t \log^{p-1} n_t \leq f(p-1) n \log^{p-1} n$. The height of the tree is $O(\log n)$, thus the total work in the tree is bounded by $c' \log n f(p-1) n \log^{p-1} n = c' f(p-1) n \log^p n$. We also take into account the time for sorting according to the attributes of the current inequality, which is bounded by $c'' n \log n$. Thus, we get that $\mathcal{T}_I(n, p) \leq c' f(p-1) n \log^p n + c'' n \log n$. If we pick a function f such that $f(1) \geq c$ and $f(p) \geq c' f(p-1) + \frac{c''}{log^{p-1} n}$, then $\mathcal{T}_I(n, p) \leq f(p) n \log^p n$. This completes the induction, establishing that $\mathcal{T}_I(n, p) = O(n \log^p n)$ in data complexity.

The size of the TLFG cannot exceed the running time, thus it is also $O(n \log^p n)$. The depth is 2 because in all cases we use the binary partitioning method and the duplication factor is 1 because we only connect tuples in the base case of one predicate $p = 1$, which we already proved does not create duplicates (Lemma 8).

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

F.6 Proof of Lemma 13

Lemmas 16, 17 and 19 together prove Lemma 13.

F.7 Proof of Theorem 14

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 5 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 20 and the fact that $n \log^p n$ and $n \log^{p-1} n \cdot \log \log n$ are superadditive functions. In the conjunction algorithm, we use multiway partitioning for $p = 1$ and binary partitioning for $p > 1$. Therefore $T_I(n, 1) \leq cn \log \log n$, resulting in $T_I(n, p) = O(n \log^{p-1} n \cdot \log \log n)$ overall. Non-equalities and bands are translated into inequalities by using the techniques we developed in Appendices D and E: a non-equality results into two inequalities on the same sets of nodes, while a band creates multiple inequality subproblems. We use the same arguments as in the proofs of Lemmas 17 and 19. We denote by $\mathcal{T}_I(n, p), \mathcal{T}_N(n, p), \mathcal{T}_B(n, p)$ the running time for n tuples and p predicates when the head of the list of predicates is an inequality,

non-equality or band respectively. $\mathcal{T}_N(n,p) = O(\mathcal{T}_I(n,p)+\mathcal{T}_I(n,p))$ and $\mathcal{T}_B(n,p) = O(n) + 2\mathcal{T}_I(n_1,p) + 2\mathcal{T}_I(n_2,p) + \ldots + \mathcal{T}_I(n_m,p)$ for $n_1 + n_2 + \ldots + n_m \leq 3n$. By these formulas, and since $\mathcal{T}_I(n,p) = O(n \log^{p-1} n \cdot \log\log n)$, it is easy to show the same bound for the other two. This proves the space consumption of the TLFGs, thus the space bound of the theorem.

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 1, except if we have disjunctions (Lemma 12). Let u_{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_{max}^{ℓ}, where ℓ is the number of Q atoms. That depends only on the query size which we consider as constant, thus it is $O(1)$. If the time for each answer without the filtering is $\mathrm{TT}'(k)$, then we have that $\mathrm{TT}(k) = O(\mathrm{TT}'(k \cdot u_{max}^{\ell})) = O(\mathrm{TT}'(k))$, since u_{max} and ℓ are $O(1)$.

G EXAMPLES OF CYCLIC QUERIES

Example 21 (Inequality Cycle). *The following triangle query variant joins three relations with inequalities in a cyclic way:* $Q(A,B,C,D,E,F) :- R(A,B), S(C,D), T(E,F), (B < C), (D < E), (F < A)$. *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, 56] of* $5/3$*, making ranked enumeration possible with* $\mathrm{TT}(k) = O((n \log n)^{5/3} + k \log k)$.

H SQL CODE FOR QUERIES USED IN EXPERIMENTS

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

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

```
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                                   Q_R1
```

```
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                                  Q_R2
```

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

I TLFG FACTORIZATION FORMULAS

Typically, factorization refers to the process of compacting an algebraic formula by factoring out common sub-expressions using the distributivity property [26]. Under that perspective, factorized databases [66] represent the results of an equi-join efficiently, treating them as a formula built with product and union. Besides distributivity, d-representations [69] replace shared sub-expressions with variables, further improving succinctness through memoization [28]. 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:*

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

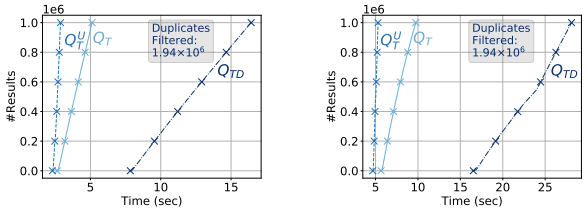
where for convenience we refer to tuples by their A or B value, and $\times$ *and* $\cup$ *denote Cartesian product and union respectively. The flat representation has one term for each query result, separated by the union operator. In terms of the TLFG,* $\times$ *corresponds to path concatenation, 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:*

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

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

J 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 $O(n \,\mathrm{polylog}\, n + k)$. This guarantee improves over the state of the art for large values of k. In particular, any approach that relies on indexes for range search, such as range trees [29], gives $\mathrm{TT}(k) = O(n \,\mathrm{polylog}\, n + k \,\mathrm{polylog}\, n)$.



(a) Scale factor $= 2^5 \times 10^{-3}, \ell = 2$. **(b) Scale factor** $= 2^6 \times 10^{-3}, \ell = 2$.

Figure 14: Appendix J.1: Variants of query Q_T with disjunctions (Q_{TD}) and unranked enumeration (Q_T^U) on TPC-H data.

For sufficiently large k, e.g., $k = n^2$, the second term dominates and our TT(k) guarantee yields an asymptotic improvement.

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, or band predicates. Let p be the maximum number of predicates excluding equalities in a conjunction in every join condition θ of the join tree. Enumeration of the answers to Q over D in an arbitrary order can be performed with* $\text{TT}(k) = O(n \log^p n + k)$ *and* $\text{MEM}(k) = O(n \log^{p-1} n \cdot \log\log n + k)$.

J.1 Experimental Comparison

We use Q_T^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 TT(k) for query Q_T, together with its disjunction Q_{TD} and unranked Q_T^U 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_T^U starts slightly faster than Q_T and has significantly lower delay between results.

K 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.

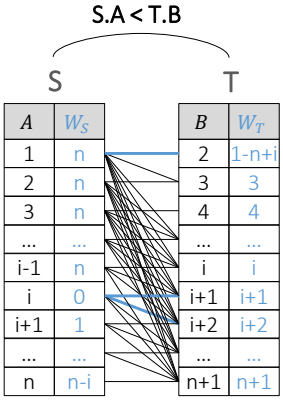
First, we would like to emphasize that **we do not compare against a naive $O(n^2)$ join algorithm**. The quadratic worst-case complexity is not caused by an inferior join algorithm but by the output size itself. In short, even if we want to retrieve only k join output tuples, the algorithm has to insert $O(n^2)$ 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 $O(n \log n)$.

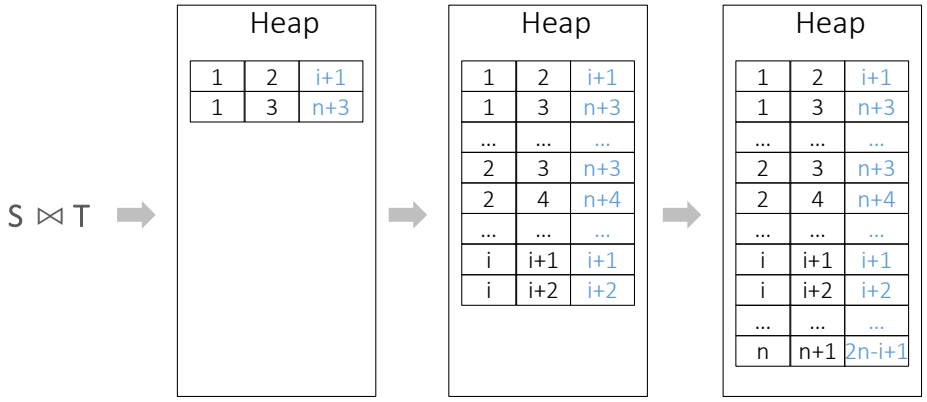
While this works well if we want to get an *arbitrary* set of k result tuples, **ranking makes the situation more challenging**. To illustrate this, suppose in the example we want to find the top-3 join results **according to the minimum sum of weights** $W_S + W_T$. Notice that in general, tuple weights may or may not be correlated with join-attribute values. In our example, we highlight the top-3 joining pairs $((1, n), (2, 1-n+i))$, $((i, 0), (i+1, i+1))$, 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-returned 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 $(i, 0), (i+1, 1), \ldots$, 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.



(a) An example inequality-join.



(b) Steps of the computation. For readability, only heap content is shown, but the actual heap structure is omitted. Clearly, the heap data structure would organize the tuples in a better way.

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 $O(n \operatorname{polylog} n)$ 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.

L MORE MOTIVATING EXAMPLES

Example 24. *Consider an ornithologist studying interactions between bird species using a bird observation dataset* `B(Species, Family, ObsCount, Latitude, Longitude)`. *For her analysis, she decides to extract pairs of observations for birds of different species from the same larger family that have been spotted in the same region. Pairs with higher* `ObsCount` *should also appear first:*

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

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