Beyond Equi-joins: Ranking, Enumeration and Factorization

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

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

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

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 $O(n + k \log k)$, i.e., the best known complexity for equi-joins, and even of $O(n + k)$, i.e., the time it takes to look at the input and return k answers in any order. Our guarantees extend to join queries with selections and many types of projections (namely those called "free-connex" queries and those that use bag semantics). Remarkably, they hold even when the number of join results is n^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.

PVLDB Reference Format:
Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. Beyond Equi-joins: Ranking, Enumeration and Factorization. PVLDB, 14(11): 2599-2612, 2021.
doi:10.14778/3476249.3476306

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

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]. Work on constant-delay (unranked) enumeration [10, 19, 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 delay (i.e., independent of database size) between them. Together, linear pre-processing and constant delay guarantee that all answers are returned in time linear in input and output size, which is optimal.

Ranked enumeration. Ranked enumeration [78] generalizes the heavily studied *top-k* paradigm [35, 45] by continuously returning join answers in ranking order. This enables the output consumer to select the cut-off k on-the-fly while observing the answers. For top-k, the value of k must be chosen in advance, before seeing any query answer. Unfortunately, non-trivial complexity guarantees of previous top-k techniques, including the celebrated Threshold Algorithm [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 general predicates, they do not provide non-trivial guarantees in the standard RAM model of computation, and their time complexity for a join of ℓ relations can be $O(n^\ell)$.

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, 27], we aim for time bounds that are within such polylogarithmic factors of $O(n + k)$. At the same time, we want space complexity to be reasonable; e.g., for small k to be within a polylogarithmic factor of $O(n)$, which is the required space to hold the input.

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

Beyond equi-joins. Recent work on ranked enumeration [30, 32, 77, 78, 86, 87] achieves much stronger worst-case guarantees, but only considers *equi-joins*. However, big-data analysis 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

Proceedings of the VLDB Endowment, Vol. 14, No. 11 ISSN 2150-8097.
doi:10.14778/3476249.3476306

[1] For non-trivial ranking functions, or when the attributes used for joining differ from those used for ranking, the DBMS cannot determine if a subset of the join output so far produced already contains all k top-ranked answers. This applies to general theta joins as well as equi-joins.

major challenges must be addressed. First, the join itself must be computed efficiently in the presence of complex conditions, possibly consisting of conjunctions and disjunctions of such predicates. Second, to avoid having to produce the entire output, *ranking has to be pushed deep into the join itself.*

Example 1. *A concrete application of ranked enumeration for inequality joins concerns graph-based approaches for detecting "lateral movement" between infected computers in a network [53]. By modeling computers as nodes and connections as timestamped edges, these approaches search for anomalous access patterns that take the form of paths (or more general subgraphs) ranked by the probability of occurrence according to historical data. The inequalities arise from a time constraint: the timestamps of two consecutive edges need to be in ascending order. Concretely, consider the relation* `G(From,To,Time,Prob)`*. Valid 2-hop paths can be computed with a self-join (where G_1, G_2 are aliases of G) where the join condition is an equality* $\mathtt{G_1.To} = \mathtt{G_2.From}$ *and an inequality* $\mathtt{G_1.Time} < \mathtt{G_2.Time}$*, while the score of a path is* $\mathtt{G_1.Prob} \cdot \mathtt{G_2.Prob}$*. 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 developed in this paper, patterns of much larger size can be retrieved efficiently in ranked order without considering all possible instantiations of the pattern.*

Main contributions. We provide the first comprehensive study on ranked enumeration for joins with conditions other than equality, notably general theta-joins and conjunctions and disjunctions of inequalities and equalities. While such joins are expensive to compute [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 the input may be heavily skewed and the 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 an equi-join-specific ranked-enumeration construction [77] by representing the overall join structure as a tree of joining relations and then introducing a join-condition-sensitive abstraction between 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 for acyclic joins, our algorithm guarantees $\mathrm{TT}(k) = O(n \operatorname{polylog} n + k \log k)$, 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 $O(n+k)$.

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 ("Time-To") $\mathrm{TT}(k) = O(\mathcal{P}(n) + k \log k)$**, using** $\mathrm{MEM}(k) = O(\mathcal{S}(n) + k)$ **space.**

(3) Our experiments (Section 6) on synthetic and real datasets show orders-of-magnitude improvements over highly optimized top-k implementations in state-of-the-art DBMSs, as well as over an idealized competitor that is not charged for any join-related cost.

Due to space constraints, formal proofs and several details of improvements to our core techniques (Section 5) are in the full version of this paper [79]. Our project website contains more information including source code: https://northeastern-datalab.github.io/anyk/.

2 PRELIMINARIES

2.1 Queries

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

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

where R_i are relational symbols, $\mathbf{X}_i$ are lists of variables (or attributes), $\mathbf{Z}, \mathbf{Y}_i$ are subsets of $\mathbf{X} = \bigcup \mathbf{X}_i$, $i \in [\ell]$, $j \in [q]$, 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 that associates with each R_i a finite relation (or table) that draws values from a domain that we assume to be $\mathbb{R}$ for simplicity.[2] 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 maximum number of tuples in an input relation is denoted by n. We write $R.A$ for an attribute A of relation R and $r.A$ for the value of A in tuple $r \in R_i$. The semantics of a theta-join query is to (*i*) create the Cartesian product of the ℓ relations, (*ii*) select the tuples that satisfy the equi-join conditions and θ_j predicates, and (*iii*) project on the $\mathbf{Z}$ attributes. 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 ($\mathbf{Z} = \mathbf{X}$). While our approach can handle

[2] 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.

projections by applying them in the end, the strong asymptotic TT(k) guarantees may not hold any more. The reason is that a projection could map multiple distinct output tuples to the same projected answer. In the strict relational model where relations are sets, those "duplicates" would have to be eliminated, creating larger gaps between consecutive answers returned to the user. Fortunately, our strong guarantees still hold for *arbitrary projections* in the presence of bag semantics, which is what DBMSs use when the SQL query has a SELECT clause instead of SELECT DISTINCT. Even for set semantics and SELECT DISTINCT queries, it is straightforward to extend our strong guarantees to non-full queries that are *free-connex* [10, 13, 17, 43].

Join trees for equi-joins. An equi-join query is (alpha-)*acyclic* [39, 75, 89] if it admits a join tree. A *join tree* is a tree with the atoms (relations) as the nodes where for every attribute A appearing in an atom, all nodes containing A form a connected subtree. The GYO reduction [89] computes such a join tree for equi-joins.

Atomic join predicates. We define the following types of predicates between attributes $S.A$ and $T.B$: an *inequality* is $S.A < T.B$, $S.A > T.B$, $S.A \leq T.B$, or $S.A \geq T.B$, a *non-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 [78] returns distinct join answers one-at-a-time, in the order dictated by a given ranking function on the output tuples. Since this paradigm generalizes top-k (top-k for "any k" value, or "anytime top-k"), it is also called any-k [77, 86]. An obvious solution is to compute the entire join output, and then either batch-sort it or insert it into a heap data structure. Our goal is to find more efficient solutions for appropriate ranking functions.

For simplicity, in this paper we only discuss ranking by increasing *sum-of-weights*, where each input tuple has a real-valued weight and the weight of an output tuple is the sum of the weights of the input tuples that were joined to derive it. Ranked enumeration returns the join answers in increasing order of output-tuple weight. It is straightforward to generalize our approach to any ranking function that can be interpreted as a *selective dioid* [77]. Intuitively, a selective dioid [37] is a semiring that also establishes a total order on the domain. It has two operators (min and + for sum-of-weights) where one *distributes* over the other (+ distributes over min). These structures include even less obvious cases such as lexicographic ordering by relation attributes.

2.3 Complexity Measures

We consider in-memory computation and analyze all algorithms in the standard Random Access Machine (RAM) model with uniform cost measure. Following common practice, we treat query size $|Q|$—intuitively, the length of the SQL string—as a constant. This corresponds to the classic notion of *data complexity* [80], where one is interested in scalability in the size of the input data, and not of the query (because users do not write arbitrarily large queries).

In line with previous work [15, 22, 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 (TT(k)) for all values of k. In the full version [79], we further discuss the relationship of TT(k) to enumeration delay as complexity measures. Since we do not assume any given indexes, a trivial lower bound is TT(k) = $O(n + k)$: the time to inspect each input tuple at least once and to return k output tuples. *Our algorithms achieve that lower bound up to a polylogarithmic factor.* For space complexity, we use MEM(k) to denote the required memory until the k^{th} result is returned.

3 GRAPH FRAMEWORK FOR JOINS

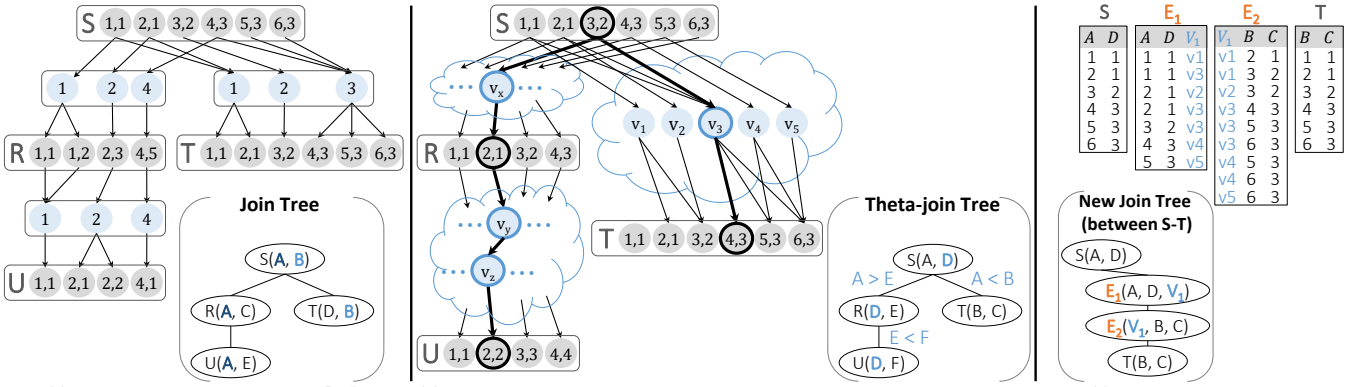
We summarize our recent work on ranked enumeration for equi-joins, then show our novel generalization to theta-joins.

3.1 Previous Work: Any-k for Equi-joins

Any-k algorithms [77] for *acyclic equi-joins* reduce ranked enumeration to the problem of finding the k^{th}-lightest trees in a layered DAG, which we call the *enumeration graph*. Its structure depends on the join tree of the given query; an example is depicted in Fig. 2a. The enumeration graph is a layered DAG in the sense that we associate it with a particular topological sort: (1) Conceptually, each node is labeled with a layer ID (not shown in the figure to avoid clutter). A layer is a set of nodes that share the same layer ID (depicted with rounded rectangles). (2) Each edge is directed, going from lower to higher layer ID. (3) All tuples from an input relation appear as (black-shaded) nodes in the same layer, called a *relation layer*. Each relation layer has a unique ID and for each join-tree edge (S, T), S has a lower layer ID than T. (4) If and only if two relations are adjacent in the join tree, then their layers are connected via a *connection layer* that contains (blue-shaded) nodes representing their join-attribute values. (5) The edges from a relation layer to a connection layer connect the tuples with their corresponding join-attribute values and vice-versa.

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

The any-k algorithm then goes through two phases on the enumeration graph. The first is a Dynamic Programming computation, where every graph node records for each of its outgoing edges the lowest weight among all subtrees that contain 1 node from each relation layer below. The minimum-subtree and input-tuple weights are not shown in Figure 2a to avoid clutter. For instance, the outgoing edge for R-node $(2, 3)$ would store the smaller of the weights of U-tuples $(2, 1)$ and $(2, 2)$. Similarly, the left edge from S-node $(2, 1)$ would store the sum of the weight of R-tuple $(2, 3)$ and the



(a) Equi-join enumeration graph [77]. (b) Theta-join enumeration graph and abstraction proposed in this paper. (c) Reduction to equi-join.

Figure 2: Overview of our approach. We generalize the equi-join-specific construction to theta-joins by introducing an abstraction (blue clouds) that factorizes binary joins. Some factorizations can also be used to reduce theta-joins to equi-joins.

minimum subtree weight from R-node $(2, 3)$. The minimum-subtree weight for a node's outgoing edge is obtained at a constant cost by pushing the minimum weight over all outgoing edges up to the node's parent. Afterwards, enumeration is done in a second phase, where the enumeration graph is traversed top-down (from S in the example), with the traversal order determined by the layer IDs and minimum-subtree weights on a node's outgoing edges.

The size of the enumeration graph and its number of layers determine space and time complexity of the any-k algorithm. The following lemma summarizes the main result from our previous work [77]. We restate it here in terms of data complexity (where query size ℓ is a constant) and using λ for the number of layers.[3]

Lemma 2 ([77]). *Given an enumeration graph with $|E|$ edges and λ layers, ranked enumeration of the k-lightest tree solutions can be performed with* $\mathrm{TT}(k) = O(|E| + k \log k + k\lambda)$ *and* $\mathrm{MEM}(k) = O(|E| + k\lambda)$.

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

3.2 Theta-Join Tree

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

Definition 3 (Theta-join Tree). *A theta-join tree for a theta-join query Q is a join tree for the equi-join Q' that has all the θ_j predicates of Q removed, and every θ_j is assigned to an edge (S, T) of the tree such that S and T contain all the attributes referenced in θ_j.*

We call a theta-join query *acyclic* if it admits a theta-join tree. In the theta-join tree, edge (S, T) represents the join $S \bowtie_\theta T$, 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.

[3]Due to the specific construction for equi-joins [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.

Example 4. *Consider $Q(A, B, C, D, E, F) :- R(D, E), S(A, D), T(B, C), U(D, F), (A < B), (A > E), (E < F)$.[a] This query is acyclic since we can construct the theta-join tree shown in Fig. 2b. Notice that all nodes containing attribute D are connected and each inequality is assigned to an edge whose adjacent nodes together contain all referenced attributes. For example, $A < B$ is assigned to (S, T) (S contains A and T contains B). The join-tree edges represent join predicates $\theta_1 = S.A < T.B$ (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 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 4, this join tree looks like the one in Figure 2b, but without the edge labels. 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. Note that there may exist different join trees for Q', and we may have to try all possible options to obtain a theta-join tree. Fortunately, this computation depends only on the query, thus takes $O(1)$ space and time in data complexity. If either the GYO algorithm fails to find a join tree for Q' or no join tree allows us to assign the θ_j predicates to tree edges, then the query is *cyclic* and can be handled as discussed in Section 5.3. We discuss next how to create the enumeration graph for a given theta-join tree.

3.3 Factorized Join Representation

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

To be able to apply our any-k algorithms [77] to this generalized enumeration graph we must ensure that (1) each "blue cloud" can be mapped to a layered graph and (2) each tree solution corresponds to a join answer, and vice versa (like the one highlighted in Figure 2b which corresponds to joining input tuples $s = (3, 2)$, $t = (4, 3)$, $r = (2, 1)$, and $u = (2, 2)$). For (2) it is sufficient to ensure for each adjacent parent-child pair of relations in the theta-join tree that there exists a path from a node in the parent-relation layer to a node in the child-relation layer 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 "blue cloud" is a DAG with parent-relation nodes only having edges going into the cloud, while all child-relation edges must point out of the cloud. We formalize these properties with the notion of a *Tuple-Level Factorization Graph* (TLFG).

Definition 5 (TLFG). *A* Tuple-Level Factorization Graph *of a theta-join $S \bowtie_\theta T$ 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* d is the maximum length of any path in G. The graphs depicted in Fig. 4a and Fig. 4b are valid TLFGs for equi-joins.

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

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

Theorem 6. *Given a theta-join Q of $\ell = O(1)$ 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 $O(|E|)$ size and $O(d)$ depth, then ranked enumeration of the k-lightest tree solutions can be performed with $\mathrm{TT}(k) = O(|E| + k \log k + kd)$ and $\mathrm{MEM}(k) = O(|E| + kd)$.*

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

Direct TLFGs. For any theta-join, a naive way to construct a TLFG is to directly connect each source node with all the target nodes it joins with. This results in $|E| = O(n^2)$ and $d = 1$, thus

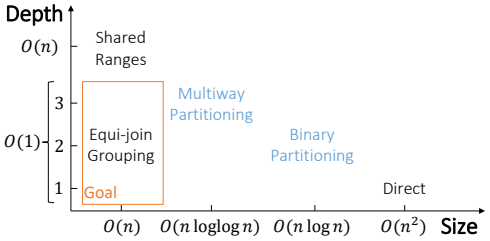


Figure 3: We propose 4 different TLFGs for a single inequality. These trade off size with depth and 2 of them (in blue) achieve the equi-join guarantee up to a logarithmic factor.

$\mathrm{TT}(k) = O(n^2 + k \log k)$ and $\mathrm{MEM}(k) = O(n^2 + k)$, respectively. Hence even the top-ranked result requires quadratic time and space. 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 6 is that two non-isomorphic tree solutions of the enumeration graph may contain the exact same input tuples (the relation-layer nodes), causing duplicate query answers. This happens if and only if a TLFG has multiple paths between the same source and destination node. While one would like to avoid this, it may not be possible to find a TLFG that is both efficient in terms of size and depth, and also free of duplicate paths. Among the inequality conditions studied in this paper, this only happens for disjunctions (Section 4.3).

Since duplicate join answers must be removed, the time to return the k top-ranked answers may increase. Fortunately, for our disjunction construction it is easy to show that the number of duplicates per output tuple is $O(1)$, i.e., it does not depend on input size n. This implies that we can filter the duplicates on-the-fly without increasing the complexity of $\mathrm{TT}(k)$ (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.[4]

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).

3.4 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 7. *Let $G = (V, E)$ be a TLFG of depth d for a theta-join $S \bowtie_\theta T$ of relations S, T and X be the union of their attributes. For $0 < i \leq d$, let E_i be the set of edges from layer $i - 1$ to i. If*

[4] 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.

$E = \bigcup_i E_i$, *i.e., every edge connects nodes in adjacent layers, then* $S \bowtie_\theta T = \pi_X(S \bowtie E_1 \bowtie \cdots \bowtie E_d \bowtie T)$ *where* π_X *is an X-projection.*

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

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

QuadEqui for direct TLFGs. Recall that any theta-join $S \bowtie_\theta T$ 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 7, it can be reduced 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 reducing the entire theta-join query between ℓ relations to an equi-join) and then uses the equi-join ranked-enumeration algorithm [77] as QuadEqui.

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

4 FACTORIZATION OF INEQUALITIES

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

4.1 Single Inequality Condition

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

Binary partitioning. Our *binary-partitioning* based TLFG is inspired by quicksort [40]. Consider condition $S.A < T.B$ and a *pivot* value v. We partition relations S and T s.t. $s.A < v$ for $s \in S_1$ and $s.A \geq v$ for $s \in S_2$, and similarly $t.B < v$ for $t \in T_1$ and $t.B \geq v$ for $t \in T_2$. This guarantees that all A-values in S_1 are strictly less than all B-values in T_2. Instead of representing this with $|S_1| \cdot |T_2|$ direct edges ($s_i \in S_1, t_j \in T_2$), we introduce an intermediate "pivot node" v and use only $|S_1| + |T_2|$ edges ($s_i \in S_1, v$) and ($v, t_j \in T_2$).

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

As the pivot, we use the median of the *distinct* join-attribute values appearing in the tuples in both input partitions. E.g., for multiset $\{1, 1, 1, 1, 2, 3, 3\}$ the set of distinct values is $\{1, 2, 3\}$ and hence the median is 2. This pivot is easy to find in $O(n)$ time if the relations have been sorted on the join attributes beforehand. Since each partition step cuts the number of distinct values per partition in half, it takes $O(\log n)$ steps until we reach the base case where all input tuples in a partition share the same join-attribute value and the recursion terminates. Overall, the algorithm takes time $O(n \log n)$ to construct a TLFG of size $O(n \log n)$ and depth 2. It is easy to see that there is exactly one path from each source to joining target node, hence the TLFG is *duplicate-free*.

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

Other inequality types. The construction for greater-than ($>$) is symmetric, connecting S_2 to T_1 instead of S_1 to T_2. For $\leq$ and $\geq$, we only need to modify handling of the base case of the recursion: instead of simply returning from the last call (when all tuples in a partition have the same join-attribute value), the algorithm connects the corresponding source and target nodes via an intermediate node (like for equality predicates).

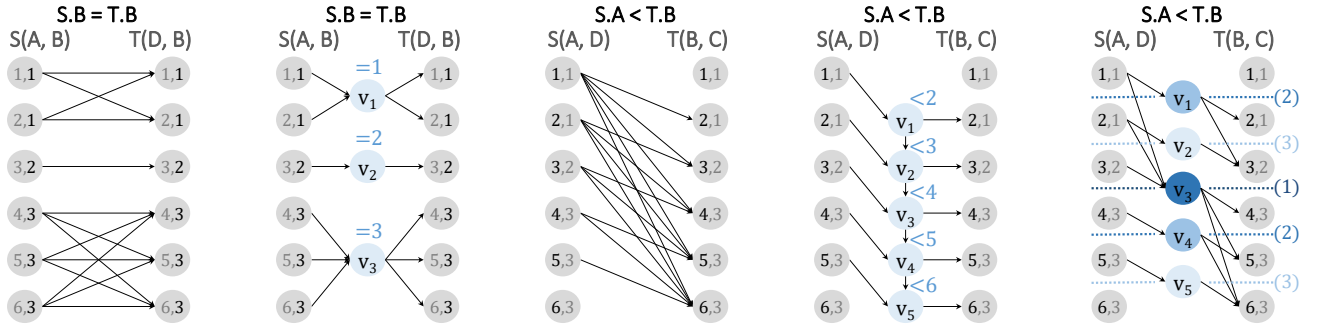
Lemma 9. *Let* θ *be an inequality predicate for relations* S, T *of total size n. A duplicate-free TLFG of* $S \bowtie_\theta T$ *of size* $O(n \log n)$ *and depth* 2 *can be constructed in* $O(n \log n)$ *time.*

4.2 Conjunctions

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

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

[5] 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.

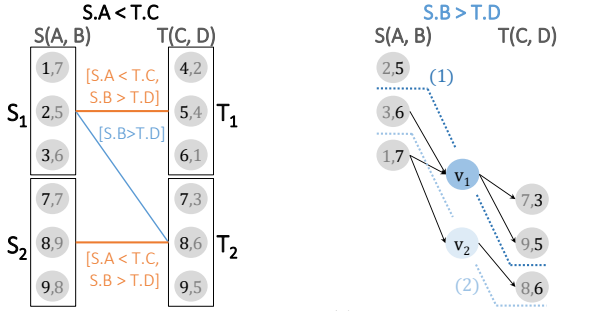
(b) Equality: grouping tuples with common join values together. $O(n)$ size, $O(1)$ depth.

(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.



(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).

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

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

Algorithm 1: Factorizing a conjunction of p strict inequalities

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

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

Lemma 11. *Let θ be a conjunction of p inequality and any number of equality predicates for relations S, T of total size n. A duplicate-free*

TLFG of $S \bowtie_\theta T$ of size $O(n \log^p n)$ and depth 2 can be constructed in $O(n \log^p n)$ time.

4.3 Disjunctions

Given a join condition that can be expressed as a disjunction $P = \bigvee_i P_i$ where G_i is the TLFG for P_i, we construct the TLFG G for P by simply "unioning" the G_i, i.e., G's set of nodes and edges are the unions of the node and edge sets of the G_i, respectively. Note that the auxiliary "pivot" nodes added by the binary partitioning algorithm to the G_i are all distinct. Hence if there is a path from source s to target t in j of the individual G_i, then there are exactly j different paths from s to t in G. This creates duplicate join results when traversing G during the enumeration phase. Fortunately, since the number of "duplicate" paths depends only on the number of terms in P and hence query size (not input size), the number of duplicates per join output tuple is constant.

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.*

We can now factorize any DNF of equality and inequality predicates by applying the conjunction construction to each conjunct, and then the union construction for their disjunction.

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.

Multiway partitioning. When the join predicate on an edge of the theta-join tree is a simple inequality like $S.A < T.B$, we can split the set of input tuples into $O(\sqrt{n})$ partitions per step—instead of 2 partitions for binary partitioning (Section 4.1)—hence the name *multiway partitioning*. This results in a smaller TLFG of size $O(n \log \log n)$ (vs. $O(n \log n)$ for binary partitioning) and depth 3 (vs. 2). Unfortunately, it is unclear how to generalize this idea to a conjunction of inequalities.

Shared ranges. A simple inequality can be encoded even more compactly with $O(n)$ edges by exploiting the transitivity of "$<$" as illustrated in Figure 4d. Intuitively, our *shared ranges* method creates a hierarchy of intermediate nodes, each one representing a range of values. Each range is entirely contained in all those that are higher in the hierarchy, thus we connect the intermediate nodes in a chain. The resulting TLFG has size and depth $O(n)$. The latter causes a high delay between consecutive join answers. From Theorem 6 and the fact that we need to sort to construct the TLFG, we obtain $\mathrm{TT}(k) = O(n \log n + n + k \log k + kn) = O(n \log n + kn)$ and $\mathrm{MEM}(k) = O(n + kn) = O(kn)$. Compared to binary partitioning's $O(n \log n + k \log k)$ and $O(n \log n + k)$ (Theorem 6, Lemma 9), respectively, space complexity is reduced by about a factor $\log n$, and without affecting time complexity, only for small k, i.e., $k = o(\log n)$. For larger $k = \Omega(n)$ both space and time complexity are worse by (almost) a factor n. (Recall that $k = O(n^\ell)$ for a join of ℓ relations.) Moreover, like for multiway partitioning, it is not clear how to generalize this construction to conjunctions of inequalities.

Non-Equality and Band Predicates. A non-equality predicate can be expressed as a 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 handle the query as follows:

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

Transformation to equi-join. An alternative approach with non-trivial guarantees is to apply our equi-join transformation to the cyclic query, and then use existing algorithms for ranked enumeration of cyclic equi-joins [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 *theta-join graph*) and assign predicates to covering edges. To achieve the equi-join transformation, we consider all pairs of connected relations in the join graph, build a TLFG according to the join condition, and then materialize relations "in the middle" as illustrated in Section 3.4. The resulting query contains only equality predicates, hence is a cyclic equi-join. Ranked enumeration for cyclic equi-joins is possible with guarantees that depend on a width measure of the query [77].

Example 15 (Inequality Cycle). *The following triangle query variant joins three relations with inequalities in a cyclic way:* $Q(A,B,C,D,E,F) :- R(A,B), S(C,D), T(E,F), (B < C), (D < E), (F < A)$. *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* $\mathrm{TT}(k) = O((n \log n)^{5/3} + k \log k)$.

6 EXPERIMENTS

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

Algorithms. We compare 5 algorithms: ① Factorized is our proposed approach. ② QuadEqui is an idealized version of the fairly straightforward reduction to equi-joins described in Section 3.4, which for each edge (S,T) of the theta-join tree uses the direct TLFG (no intermediate nodes) to convert $S \bowtie_\theta T$ 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 DBMSs. It computes the entire join output and puts it into a heap for ranked enumeration. To avoid concerns about the most efficient join implementation, we give Batch the entire join output "for free" as an in-memory array. It simply needs to read those output tuples (instead of having to execute the actual join) to rank them, therefore the numbers reported constitute a *lower bound* of real-world running time. We note that for a join of only $\ell = 2$ 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 with replacement, 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 the full version [79] 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 [84] 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.

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 [23, 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 [12, 77]. To enable input caching for PSQL and System X, each execution is performed twice and we only measure 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

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

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 $\ell - 1$.

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 $8k$ input size, while we can easily handle $2M$. For very small input, the idealized methods QuadEqui and Batch are sometimes faster, but their real running time would be much higher if join computation was accounted for. Q_{S2} has more join predicates and thus smaller output size (Figures 7c and 7d). Our advantage is smaller in this case, yet still significant for large n.

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

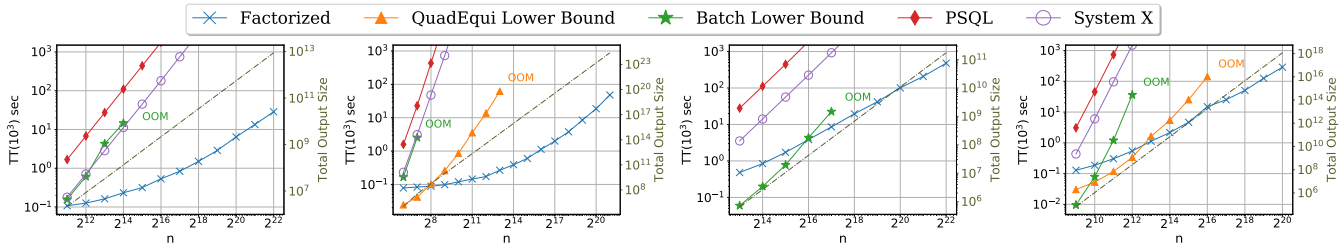
6.1.2 Effect of Query Length. Next, we test the effect of query length on RedditTitles. We plot TT(k) for three values ($k = 1, 10^3, 10^6$) when the length is small ($\ell = 2, 3$) and one value ($k = 10^3$) for longer queries. Note that for $k = 1$, the time of Factorized is essentially the time required 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 $k = 10^6$ 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 smaller. Thus, the Batch lower bound is slightly better than our approach for $\ell = 2$ (Figure 9e), but we expect it to be significantly slower if the cost of computing and materializing the output was taken into account. Either way, for $\ell \geq 3$ (Figure 9g), our approach dominates even when compared against the lower bounds. PSQL again times out for $\ell = 3$ (Figure 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.

6.2 Comparison of our Variants

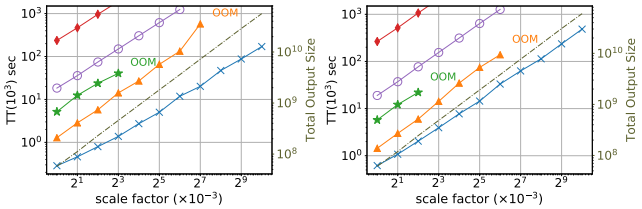
We now compare our 3 factorization methods (1a), (1b), (1c).

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 suffers from a high enumeration delay (linear in data size), and quickly deteriorates as k increases. The delay is also depicted in Figure 10b, where we observe that Binary Partitioning returns results with lower delay than Multiway Partitioning (recall that Multiway Partitioning has a depth of 3 vs Binary Partitioning's 2). These



(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.

results are a consequence of the size-depth tradeoff of the TLFGs (Fig. 3). Note that the higher delay observed in the beginning is due to lazy initialization of data structures needed by the any-k algorithm.

6.2.2 Join Representation. We show the sizes of the constructed representation in Figure 10c, using an implementation-agnostic measure. As n increases there is an asymptotic difference between the three methods ($O(n \log n)$ 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 likely. This is expected since the intermediate nodes of our TLFG essentially represent ranges in the domain and they are more compact for smaller domains. Domain size does not significantly impact running time once it exceeds sample size (around $n = 2^{16}$) and the probability of sampling duplicate domain values approaches zero.

7 RELATED WORK

Enumeration for equi-joins. *Unranked* enumeration for equi-joins has been explored in various contexts [13, 14, 19, 20, 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 [10, 16]. For the more demanding task of ranked enumeration, a logarithmic delay is unavoidable [18, 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 [21, 22] 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* [10]) predicates mainly rely on variations of color coding [8, 49, 71]. The same core idea is leveraged by the unranked enumeration algorithm of Bagan et al. [10]. Queries with negation can be answered by rewriting them with *not-all-equal-predicates* [46], a generalization of non-equality.

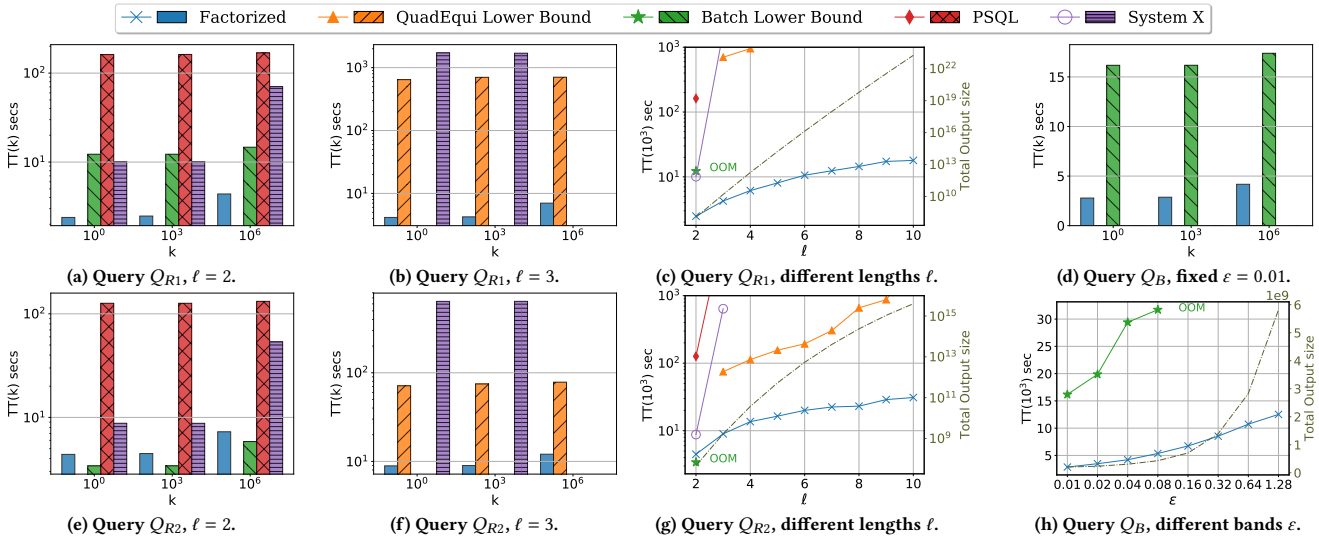
Khayatt et al. [48] provide optimized and distributed *batch* algorithms for up to two inequalities per join. Aggregate computation [3] and Unranked enumeration under updates [43] have been studied for inequality predicates by using appropriate index structures.

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

Orthogonal range search. Our binary partitioning method shares a similar intuition with index structures that have been devised for orthogonal range search [6, 25]. For unranked enumeration, it has been shown [7, 82, 83] how, for two relations, a range tree [29] can be used to identify pairs of matching tuple sets. This gives an alternative method to construct our depth-2 TLFGs because a pair of matching tuple sets can be connected via one intermediate node. Our approach supports ranking and it is simpler since it does not require building a range tree. Our TLFG abstraction is also more general: our other representations (such as multiway partitioning) do not have any obvious representation as range trees.

Factorized databases. Factorized representations of query results [11, 66] have been proposed for *equi-joins* in the context of enumeration [68, 69], aggregate computation [11], 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 [26].

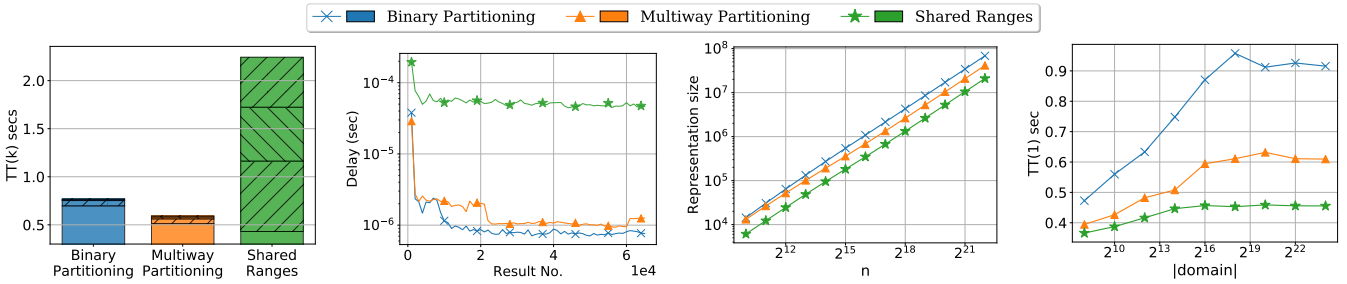
Top-k queries. Top-k queries [72] are a special case of ranked enumeration where the value of k is given in advance and its knowledge can be exploited. Fagin et al. [35] present the Threshold Algorithm, which is instance-optimal under a "middleware" cost model for a restricted class of 1-to-1 joins. Follow-up work generalizes the idea to more general joins [36, 44, 55, 85], 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



(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 preprocessing (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 window of size 10^3. (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 $Q_{S1}, \ell = 2$.

these approaches, along with some related ones such as building top-k indexes [24, 76] or views [28, 41].

Optimal batch algorithms for joins. Acyclic equi-joins are evaluated optimally in $O(n + |\text{out}|)$ by the Yannakakis algorithm [88], where $|\text{out}|$ is the output size. This bound is unattainable for cyclic queries [61], thus worst-case optimal join algorithms [58, 61, 62, 81] settle for the AGM bound [9], 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 DBMSs struggle with their efficient computation. We developed the first ranked-enumeration techniques that achieve non-trivial worst-case guarantees for a large class of these joins: For small k, returning the k top-ranked join answers for full acyclic queries takes only slightly-more-than-linear time and space ($O(n \operatorname{polylog} n)$) 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.

ACKNOWLEDGMENTS

This work was supported in part by the National Institutes of Health (NIH) under award number R01 NS091421 and by the National Science Foundation (NSF) under award numbers CAREER IIS-1762268 and IIS-1956096.

REFERENCES

[1] 2020. Bird Occurrences in Oceania. https://doi.org/10.15468/dl.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] Pankaj K. Agarwal. 2017. Range Searching. In *Handbook of Discrete and Computational Geometry, Third Edition*, Jacob E. Goodman, Joseph O'Rourke, and Csaba D. Tóth (Eds.). Chapman and Hall/CRC, 1057–1092. https://doi.org/10.1201/9781315119601

[7] Pankaj K Agarwal, Xiao Hu, Stavros Sintos, and Jun Yang. 2021. Dynamic Enumeration of Similarity Joins. *CoRR* (2021). arXiv:2105.01818

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

[9] 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

[10] 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

[11] 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

[12] 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

[13] 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

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

[15] 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

[16] 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

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

[18] 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

[19] 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

[20] 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

[21] Nofar Carmeli, Nikolaos Tziavelis, Wolfgang Gatterbauer, Benny Kimelfeld, and Mirek Riedewald. 2021. Tractable Orders for Direct Access to Ranked Answers of Conjunctive Queries. In *PODS*. 325–341. https://doi.org/10.1145/3452021.3458331

[22] 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

[23] 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

[24] 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

[25] 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

[26] 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

[27] 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

[28] 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

[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] Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. 2021. Beyond Equi-joins: Ranking, Enumeration and Factorization. *CoRR* abs/2101.12158 (2021). arXiv:2101.12158
[80] Moshe Y. Vardi. 1982. The Complexity of Relational Query Languages (Extended Abstract). In *STOC*. 137–146. https://doi.org/10.1145/800070.802186
[81] 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
[82] Dan E. Willard. 1996. Applications of Range Query Theory to Relational Data Base Join and Selection Operations. *J. Comput. System Sci.* 52, 1 (1996), 157–169. https://doi.org/10.1006/jcss.1996.0012
[83] Dan E Willard. 2002. An algorithm for handling many relational calculus queries efficiently. *J. Comput. System Sci.* 65, 2 (2002), 295–331. https://doi.org/10.1006/jcss.2002.1848
[84] 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
[85] 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
[86] 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
[87] 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
[88] Mihalis Yannakakis. 1981. Algorithms for Acyclic Database Schemes. In *VLDB*. 82–94. https://dl.acm.org/doi/10.5555/1286831.1286840
[89] 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