Conjunctive Queries with Comparisons

Qichen Wang
Hong Kong University of Science and Technology
Hong Kong, China
qwangbp@cse.ust.hk

Ke Yi
Hong Kong University of Science and Technology
Hong Kong, China
yike@cse.ust.hk

ABSTRACT

Conjunctive queries with predicates in the form of comparisons that span multiple relations have regained interest recently, due to their relevance in OLAP queries, spatiotemporal databases, and machine learning over relational data. The standard technique, predicate pushdown, has limited efficacy on such comparisons. A technique by Willard can be used to process short comparisons that are adjacent in the join tree in time linear in the input size plus output size. In this paper, we describe a new algorithm for evaluating conjunctive queries with both short and long comparisons, and identify an acyclic condition under which linear time can be achieved. We have also implemented the new algorithm on top of Spark, and our experimental results demonstrate order-of-magnitude speedups over SparkSQL on a variety of graph pattern and analytical queries.

CCS CONCEPTS

• **Theory of computation** → **Database query processing and optimization (theory)**; • **Information systems** → *Join algorithms.*

KEYWORDS

Conjunctive query; acyclic joins; inequality joins

ACM Reference Format:
Qichen Wang and Ke Yi. 2022. Conjunctive Queries with Comparisons. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3514221.3517830

1 INTRODUCTION

The asymptotically optimal running time for evaluating a query is $\tilde{O}(N + \text{OUT})$[1], where N is the input size and OUT the output size. This bound, which is often referred to as *linear time*, can be considered *instance-optimal*, because one has to read the input (assuming no indexes are pre-built) and write the output. Thus, a fundamental problem in query processing is to identify the class of queries that can be evaluated in linear time. A 40-year-old result by Yannakakis [24] tells us that linear time can be achieved for α-acyclic *conjunctive queries (CQs)*, and recent negative results [3, 20] suggest that this is also probably the best one can hope for.

A CQ corresponds to a (natural) join-projection query in SQL. Another important relational operator is selection. There are two types of predicates used in a selection: those that involve attributes from one relation, and those that span two or more relations. The former can be trivially handled by scanning the relation in linear time; alternatively, indexes can be pre-built over frequently queried attributes to further reduce query processing time, on which there is extensive literature. On the other hand, the second type of predicate has received much less attention. The naive method for handling this type of predicate is to first compute the join, and then filter the join results with the predicate. A common query optimization technique is *predicate pushdown*, where the predicate is pushed right after the involved relations have been joined.

Note that if a type-2 predicate is an equality, it can be rewritten as a (natural) join condition, so we consider inequalities or comparisons[2]. The following gives an example.

Example 1.1. Consider the following query with a type-2 predicate, written in a rule-based form:

$$R_1(x_1, x_2), R_2(x_2, x_3), R_3(x_3, x_4), x_1 \le x_3.$$

This query (the CQ part) is α-acyclic. For the query plan $(R_1 \bowtie R_2) \bowtie R_3$, the predicate $x_1 \le x_3$ can be pushed to after $R_1 \bowtie R_2$. However, the running time of this query plan (with or without predicate pushdown) is no longer linear, since the predicate may make the output size significantly smaller than the join size. To see this, just imagine the case where no join results satisfy the predicate $x_1 \le x_3$. In this case, OUT = 0 but the intermediate join size $|R_1 \bowtie R_2|$ can be as large as $\Omega(N^2)$.

A simple idea [15, 23] to reduce the time to (near) linear is to first push down the predicate, and then compute the sub-query

$$R_1(x_1, x_2), R_2(x_2, x_3), x_1 \le x_3$$

without computing the join $R_1 \bowtie R_2$: Group the tuples in R_1 and R_2 by x_2. For each group, sort the x_1 values in ascending order. Then for each x_3, scan the sorted list until meeting some $x_1 > x_3$. The cost is thus $\tilde{O}(|R_1| + |R_2|)$, plus the actual size of the sub-query result, hence linear. The last join with R_3 preserves linearity following the same argument[3] as in [24]. □

The predicate $x_1 \le x_3$, as we define more formally in Section 4, is a *short* comparison. The following example features a *long* one:

Example 1.2. Suppose we change the predicate $x_1 \le x_3$ to $x_1 \le x_4$ in Example 1.1:

$$R_1(x_1, x_2), R_2(x_2, x_3), R_3(x_3, x_4), x_1 \le x_4.$$

Note that a long comparison like $x_1 \le x_4$ cannot be pushed down. By decomposing the query to multiple parts, each plugged into an

[1] The $\tilde{O}$ notation suppresses a $\log^{O(1)} N$ factor.

[2] In this paper, we consider $\neq$ as inequality and $\le, <, \ge, >$ as comparisons.

[3] More rigorously, dangling tuples in R_1 and R_2 should be removed by semijoins before computing the sub-query.

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9249-5/22/06…$15.00
https://doi.org/10.1145/3514221.3517830

appropriately chosen *generalized hypertree decomposition (GHD)* [9] and combined with the idea from Example 1.1, Khamis et al. [15] are able to reduce the running time of this query to $\tilde{O}(N^{1.5} + \text{OUT})$, but it is not clear if the algorithm is practical. □

In this paper, we improve the running time of the query above to linear. More precisely, after $\tilde{O}(N)$-time preprocessing, our algorithm can enumerate the query results with constant *delay* (formal definition given in Section 4.2). This immediately implies $\tilde{O}(N + \text{OUT})$ total time; in addition, it means that the Boolean query (i.e., deciding if the query result is empty) can be answered in $\tilde{O}(N)$ time.

Our techniques are not restricted to this particular query. We are able to achieve linear time for a natural class of acyclic *conjunctive queries with comparisons (CQCs)*. On a high level, we require the relations to satisfy the α-acyclicity condition as in [24], while the comparisons should be *Berge-acyclic*, another popular definition of acyclicity for hypergraphs. The formal definition is given in Section 4, followed by our main algorithm for full CQCs described in Section 5. We extend this algorithm to handle non-full CQCs (i.e., join-selection-projection queries) in Section 6. For queries outside this class, in Section 7, we show how to combine with the GHD framework to obtain improved running times over [15].

Our algorithm consists of a series of reductions, each reducing the "length" of a long comparison, until it becomes a short one. In some sense, Khamis et al. [15] also try to reduce the length, but they only use the GHD framework, which groups multiple relations into bags, inevitably leading to superlinear running times. The key in the reductions is that we cannot just rewrite the query, but also transform the data (in linear time). The transformation will happen twice: once in the reduction, and once in "unwinding" the reduction.

Besides asymptotic improvements, our algorithm is also very practical. In fact, the transformations use some standard relational operations that are supported in all DBMSs. To verify its practical performance, we implement our algorithm in Spark, which gives us the additional benefits of parallelism, scalability, and fault tolerance. Our implementation uses only standard RDD operations without any modification to the Spark core. Experimental results (Section 8) show that our algorithm offers an order-of-magnitude improvement over SparkSQL, especially for queries with highly selective type-2 predicates.

2 RELATED WORK

Selection, projection, and join are the 3 most fundamental operators in relational databases, forming the backbone of most SQL queries. While conjunctive queries (i.e., queries composed of joins and projections) and type-1 predicates for selection have been studied extensively in the literature, type-2 predicates have received little attention, despite their frequent appearances in OLAP queries and spatiotemporal databases. Recently, they have regained interest, with several papers [11, 15, 21] addressing the issue. In particular, Khamis et al. [15] make a good case by showing that many machine learning tasks over relational data can be formulated as queries with type-2 comparisons.

The idea in Example 1.1 is perhaps the first technique (other than predicate pushdown) for dealing with type-2 comparisons. It was proposed by Willard [23], who also generalized it to α-acyclic CQs with multiple comparisons, but all comparisons must be short. The recent works [11, 15, 21] extended Willard's algorithm in various ways. Idris et al. [11] study the dynamic version of the problem; in the static setting, their algorithm is essentially the same as Willard's[4]. Tziavelis et al. [21] study ranked enumeration of full acyclic CQCs with only short comparisons; their algorithm for the unranked version also achieves $\tilde{O}(N + \text{OUT})$ time. All these papers [11, 21, 23] only consider short comparisons. Khamis et al. [15] combine GHDs and Willard's technique to handle long comparisons as shown in Example 1.2, but the running time is superlinear (more examples comparing our result and [15] are provided in Section 7). They also generalize their framework to handle aggregation queries, which are important for machine learning tasks.

Another common relational operator is the union. As observed by Carmeli and Kröll [5], the union of CQs is no harder than the constituting CQs, and the same observation holds for the union of CQCs. The interesting discovery in [5] is that certain unions of CQs are actually easier than the CQs. Thus, our algorithm can potentially solve some unions of CQCs in linear time even if the constituting CQCs are hard (i.e., not acyclic), which is an interesting future direction.

Koutris et al. [16] and Khamis et al. [13] study CQs with inequalities ($\neq$). Such a predicate, e.g., $x_1 \neq x_4$, can be written as the disjunction of two comparisons: $x_1 < x_4 \vee x_1 > x_4$, which turns the query into a union of CQCs. Then by the argument above, our algorithm can also handle such queries. However, they are interested in the *combined complexity* where the query size is not taken as a constant. Under this setting, this simple conversion results in exponentially (in the number of inequalities) many CQCs, thus our result is not directly comparable to theirs.

3 PRELIMINARIES

3.1 Conjunctive Queries with Comparisons

We follow the notation in [1]. Let $[n] = \{1, \dots, n\}$. Let $\boldsymbol{R}$ be a relational database. A *conjunctive query with comparison (CQC)* has the form

$$ans(\bar{y}) \leftarrow R_1(\bar{x}_1), \dots, R_n(\bar{x}_n), C_1, \dots, C_m \tag{1}$$

where $R_1, R_2, \dots$ are relations in $\boldsymbol{R}$, and $\bar{x}_1, \dots, \bar{x}_n$ are their variables/attributes. We use $var(q) = \bar{x}_1 \cup \cdots \cup \bar{x}_n$ to denote the set of variables appearing in the body of the query q. Without loss of generality, we assume that there are no self-joins; for queries with self-joins, one can always make logical copies of the relation. It is required that the output attributes $\bar{y} \subseteq var(q)$. If $\bar{y} = var(q)$, the query is said to be *full*; in this case we may omit writing the head $ans(\bar{y})$. Let $\mathbf{dom}(x)$ be the domain of variable x, and let $\mathbf{dom}(\bar{x})$ be the Cartesian product of all $\mathbf{dom}(x)$'s for $x \in \bar{x}$. Each C_j, for $j \in [m]$, is a *comparison* of the form $f_j(\bar{x}_{a_j}) \leq g_j(\bar{x}_{b_j})$, where a_j (resp. b_j) $\in [n]$, and f_j (resp. g_j) is a function mapping $\mathbf{dom}(\bar{x}_{a_j})$ (resp. $\mathbf{dom}(\bar{x}_{b_j})$) to $\mathbb{R}$.

Examples 1.1 and 1.2 are simple examples of full CQCs. Below we give a more complicated, non-full CQC.

[4] Actually, as stated, the result in [11] is worse than Willard's when there are two or more comparisons between two relations, but this can be fixed.

Example 3.1. The query

$$\begin{aligned} ans(x_1, x_2, x_3, x_4, x_7) \leftarrow & R_1(x_1, x_2), R_2(x_2, x_3, x_7), \\ & R_3(x_2, x_3, x_4, x_5), R_4(x_3, x_6), R_5(x_3, x_8), \\ & C_1 : x_1 - x_2 \le x_3 x_4 + 2, \\ & C_2 : \min\{2x_2, x_7\} \le x_6, \;\; C_3 : x_2 \le x_8 \end{aligned}$$

fits the definition of a CQC by setting $a_1 = 1, f_1(\bar{x}_1) = x_1 - x_2, b_1 = 3, g_1(\bar{x}_3) = x_3 x_4 + 2$; $a_2 = 2, f_2(\bar{x}_2) = \min\{2x_2, x_7\}, b_2 = 4, g_2(\bar{x}_4) = x_6$; $a_3 = 3, f_3(\bar{x}_3) = x_2, b_3 = 5, g_3(\bar{x}_5) = x_8$. □

Note that for a comparison C_j, the indices a_j, b_j of the two involved relations might not be unique. For instance, for the query above, a_3 could also be 1 or 2. In general, a_j (resp. b_j) can be any i such that the variables in f_j (resp. g_j) are contained in $\bar{x}_i$. After fixing any valid a_j, b_j, we say that C_j is *incident* to R_{a_j} and R_{b_j}. While we consider comparisons incident to two relations in the bulk of the paper, we show how comparisons involving more than two relations can be handled in Section 7 (cf. Example 7.2).

We now define the semantics of CQCs. Given a set of variables $\bar{x}$, a tuple t over $\bar{x}$ is an assignment of values from $\mathbf{dom}(\bar{x})$ to $\bar{x}$. For any $\bar{y}$, define $t(\bar{y})$ as the tuple restricted to the variables in $\bar{y}$. Given a CQC q in the form of (1), the query results of q on a database instance $\boldsymbol{R}$ are:

$$q(\boldsymbol{R}) = \left\{ t(\bar{y}) \;\middle|\; \begin{array}{l} t \text{ is a tuple over } var(q), \\ t(\bar{x}_i) \in R_i(\bar{x}_i)\ \forall i \in [n], \\ f_j(t(\bar{x}_{a_j})) \le g_j(t(\bar{x}_{b_j}))\ \forall j \in [m] \end{array} \right\}. \quad (2)$$

The restriction of considering only comparisons in the form of $f_j(\bar{x}_{a_j}) \le g_j(\bar{x}_{b_j})$ is without loss of generality: the comparison $f_j(\bar{x}_{a_j}) \ge g_j(\bar{x}_{b_j})$ can be written as $-f_j(\bar{x}_{a_j}) \le -g_j(\bar{x}_{b_j})$; the comparison $f_j(\bar{x}_{a_j}) < g_j(\bar{x}_{b_j})$ can be written as $f_j(\bar{x}_{a_j}) + \varepsilon \le g_j(\bar{x}_{b_j})$ for infinitesimally small ε[5]. An inequality $f_j(\bar{x}_{a_j}) \ne g_j(\bar{x}_{b_j})$ can be written as $f_j(\bar{x}_{a_j}) > g_j(\bar{x}_{b_j}) \vee f_j(\bar{x}_{a_j}) < g_j(\bar{x}_{b_j})$. Note that this converts a CQC into a union of disjoint CQCs. This does not affect the asymptotic running time, since the query size is taken as a constant. Alternatively, since an inequality predicate usually has high selectivity, it will be more efficient to evaluate the CQC ignoring these inequalities first and only check them during enumeration. This works better in practice, although the theoretical running time cannot be guaranteed.

When the variables of $f_j(\bar{x}_{a_j})$ and $g_j(\bar{x}_{b_j})$ are understood from the context, given a tuple t whose attributes contain $\bar{x}_{a_j}$ and/or $\bar{x}_{b_j}$, we often simply write $f_j(t)$, $g_j(t)$ instead of $f_j(t(\bar{x}_{a_j}))$, $g_j(t(\bar{x}_{b_j}))$.

Two syntactically different CQCs may be semantically equivalent. For instance, the query in Example 3.1 will remain semantically the same if C_1 is changed to $x_1 - 2 \le x_2 + x_3 x_4$, which might change the incident relations of C_1 (although in this example, it doesn't). As a more subtle example, C_2 can be rewritten as a disjunction $2x_2 \le x_6 \vee x_7 \le x_6$, so the query is equivalent to the union of two CQCs, both of which could be syntactically easier than the original query. The query containment problem for CQCs, even when all comparisons are between two variables (e.g., C_3 in Example 3.1) is already Π_2^p-complete [22]. Thus in this paper, we just focus on evaluating a CQC as given.

[5]The use of an infinitesimally small ε is for theoretical convenience. All our algorithms work if $\ge$ ($\le$) is replaced by $<$ ($>$) directly. In the implementation, we use an abstract comparator and instantiate it with the actual comparison operator in the query.

3.2 Orthogonal Range Searching

We will make use of some classical results on orthogonal range searching. Let $(\mathbb{S}, \oplus)$ be a commutative semigroup, where $\mathbb{S}$ is the ground set and $\oplus$ is its "addition" operator. Let P be a set of N points in d-dimensional space, where each point $p \in P$ is associated with a weight $w(p) \in \mathbb{S}$. There are two versions of the problem. In the aggregation version, one aims at building a data structure on P such that for any orthogonal query rectangle B, the sum $\bigoplus_{p \in P \cap B} w(p)$ can be returned efficiently. In the reporting version, the goal is to report all points in $P \cap B$. Multi-dimensional range trees can be used to solve both versions [4, 6]. In particular, all our queries will be one-sided, i.e., the constraint is in the form of $(-\infty, x]$ or $[x, \infty)$ in each dimension. For such queries, a range tree with fractional cascading [7] can be built in $O(N \log^{\max\{d-1,1\}} N)$ time so that any aggregation query can be answered in $O(\log^{\max\{d-1,1\}} N)$ time and any reporting query can be answered in $O(\log^{\max\{d-1,1\}} N + |P \cap B|)$ time.

3.3 Complexity Measures

We adopt the standard RAM model of computation and measure the running time in terms of data complexity, i.e., the query size $|Q|$ is treated as a constant, while using the input size $N = \sum_i |R_i(\bar{x}_i)|$ and output size $\mathrm{OUT} = |q(\boldsymbol{R})|$ as asymptotic parameters. Note that OUT can be much smaller than that of the CQ without the comparisons.

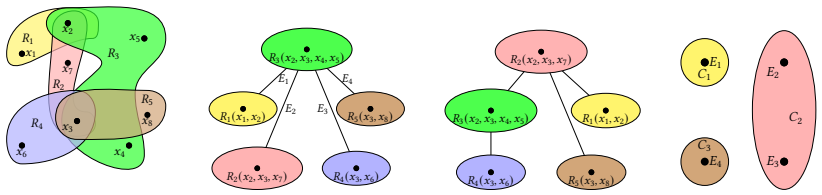
We also require a linear-space index structure that can support key lookups in constant time, and enumerate all tuples corresponding to a given key with constant delay. A standard implementation of such an index is a hash table [8], which can also be built in expected linear time.

4 ACYCLICITY OF CQCS

4.1 Acyclic CQs and CQCs

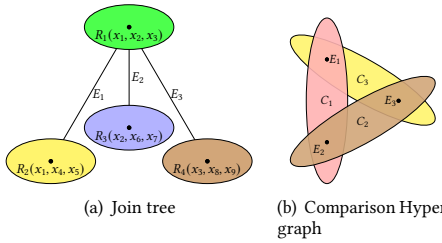
The acyclicity of a CQ q is defined by the α-acyclicity of its *relation hypergraph*, denoted $\mathcal{R}(q)$. The vertices of $\mathcal{R}(q)$ correspond to the variables and its hyperedges correspond to the relations. For example, Figure 1(a) shows the relation hypergraph of the query in Example 3.1. The CQ is said to be acyclic if $\mathcal{R}(q)$ is α-acyclic, i.e., the relations of q admit a *join tree*. A join tree is a tree T with n vertices corresponding to the relations $\{R_i(\bar{x}_i)\}_i$. For any $i, j \in [n]$, let $P_T(i, j)$ denote the unique path between i and j in the join tree T. It is required that $\bar{x}_i \cap \bar{x}_j \subseteq \bar{x}_k$ for every node $k \in P_T(i, j)$. Join trees are not unique, and we use $\mathcal{T}(q)$ to denote the set of all valid join trees of q. One can use the GYO algorithm [1, 10, 25] to find all its join trees. For example, Figure 1(b) and 1(c) give two possible join trees of the query in Example 3.1.

For a CQC q, we consider a second hypergraph, called its *comparison hypergraph*, which is defined after fixing a join tree T of q. After fixing T, for each comparison C_j, we set its two incident relations R_{a_j}, R_{b_j} such that among all valid (a_j, b_j) pairs, $P_T(a_j, b_j)$ is the shortest. For instance, for comparison C_3 in Example 3.1, we would set $a_3 = 3, b_3 = 5$ if using the join tree in Figure 1(b), while set $a_3 = 2, b_3 = 5$ if using the join tree in Figure 1(c). A comparison is said to be *short* if it is incident to two adjacent nodes of the join tree, otherwise, *long*.



(a) Relational hypergraph (b) Join tree (c) Another join tree (d) The comparison hypergraph for the join tree in (b)

Figure 1: Relational hypergraph, join trees, and comparison hypergraph for the query in Example 3.1.



(a) Join tree (b) Comparison Hypergraph

Figure 2: Join tree and comparison hypergraph for query in Example 4.1.

Then the comparison hypergraph of q induced by a given join tree T, denoted as $C(q,T)$, is defined as follows. The vertices of $C(q,T)$ correspond to the edges of T, while its hyperedges correspond to the comparisons in q. More precisely, a vertex in $C(q,T)$, namely an edge (u,v) of T, belongs to a hyperedge of $C(q,T)$, namely a comparison $f_j(\bar{x}_{a_j}) \le g_j(\bar{x}_{b_j})$, if $(u,v) \in P_T(a_j,b_j)$ (abusing notation, we use P_T to denote either the set of nodes or the set of edges on the path depending on the context). Thus, a short comparison becomes a singleton hyperedge in $C(q,T)$, and a *self-comparison*, i.e., one where $a_j = b_j$, becomes an empty hyperedge. Meanwhile, some vertices in $C(q,T)$ may not belong to any hyperedge.

Figure 1(d) shows the comparison hypergraph of the CQC in Example 3.1 after fixing the join tree in Figure 1(b).

We say that a CQC q is *acyclic*, if its relation hypergraph $\mathcal{R}(q)$ is α-acyclic, and there exists a join tree T such that $C(q,T)$ is *Berge-acyclic*. Such a T is said to *support* the comparisons in q. Recall that a hypergraph is Berge-acyclic if and only if there is at most one simple path between any two vertices. Recall that the vertices in $C(q,T)$ are the edges of T, so the Berge-acyclicity of $C(q,T)$ means that there is at most one way to go from any one edge of T to another edge via a sequence of steps, where each step is covered by a comparison. Berge-acyclicity is more restrictive than α-acyclicity: the former implies the latter, but not vice versa. Note that singleton and empty hyperedges (i.e., short and self comparisons) do not affect the Berge-acyclicity of a hypergraph.

Examples 1.1, 1.2, 3.1 are all acyclic CQCs; below we give one that is not.

Example 4.1. The query

$$\begin{aligned} ans(x_1,x_2,x_3) \leftarrow & R_1(x_1,x_2,x_3), R_2(x_1,x_4,x_5), \\ & R_3(x_2,x_6,x_7), R_4(x_3,x_8,x_9), \\ & C_1 : x_4 \le x_6,\ \ C_2 : x_7 \le x_8,\ \ C_3 : x_9 \le x_5 \end{aligned}$$

is not an acyclic CQC, although its relational hypergraph is acyclic, as witnessed by the join tree in Figure 2(a), which is in fact the only possible join tree. However, the comparison hypergraph induced by this join tree is Figure 2(b), which is not Berge-cyclic. □

Given an acyclic CQC q and a join tree T supporting its comparisons, consider the induced comparison hypergraph $C(q,T)$. Each edge (u,v) in T corresponds to a vertex in $C(q,T)$, and we use $d(u,v)$ to denote its *degree* in $C(q,T)$, i.e., the number of hyperedges of $C(q,T)$ that contain (u,v). The degree of $C(q,T)$ is the maximum degree of all $(u,v) \in T$, and we define the degree of q as the minimum degree of $C(q,T)$ over all join trees T supporting the comparison of q, denoted as d_q. For example, the degree of CQC in Example 3.1 is 1, with the join tree in Figure 1(b) being the T that attains the minimum degree of $C(q,T)$. On the other hand, the join tree in Figure 1(c) would lead to a $C(q,T)$ of degree 2 (the edge between R_2 and R_3 would be contained in two hyperedges). The degree of q, as well as the supporting join tree T, can be found by enumerating all possible join trees of q using the GYO algorithm. This takes time exponential in the size of the query, but independent of the size of the data. Henceforth, we assume that the degree of q and the supporting join tree T are given.

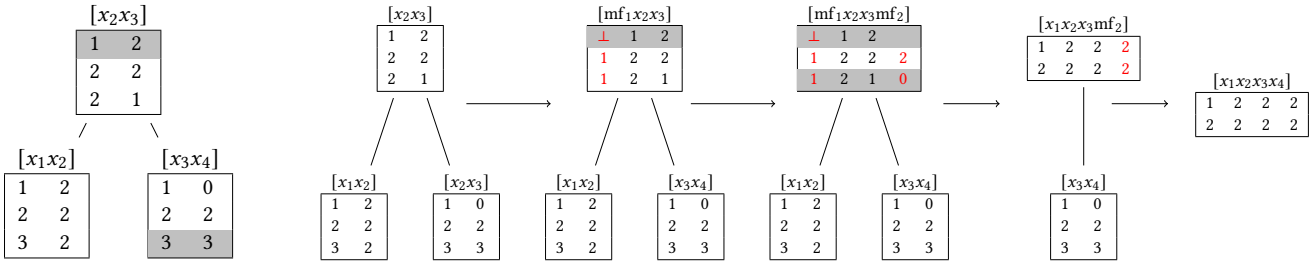
Finally, it is easy to see that an α-acyclic CQ is just a special acyclic CQC of degree 0 by our definition.

4.2 Constant-delay Enumeration

Acyclic full CQs can be evaluated in $\tilde{O}(N + \text{OUT})$ time by the well-known Yannakakis algorithm [24]. This algorithm has been extended to perform *constant delay enumeration (CDE)* [3]. A CDE data structure is one that can be built, ideally in $\tilde{O}(N)$ time, from which the query results $q(\boldsymbol{R})$ can be enumerated (without repetition) with constant delay, i.e., the time between the start of the enumeration process and enumerating the first result in $q(\boldsymbol{R})$, the time between enumerating any two consecutive results, and the time between the last result and the end of the enumeration process are all bounded by a constant. In this paper, we relax the requirement slightly, by allowing the delay to be $\tilde{O}(1)$. Note that a CDE data structure immediately leads to an $\tilde{O}(N+\text{OUT})$-time algorithm for computing $q(\boldsymbol{R})$, but not necessarily vice versa.

The design of the CDE algorithm is based on the simple observation that, after the Yannakakis algorithm has completed the semi-join reductions that remove all dangling tuples, every remaining tuple is guaranteed to produce at least one join result. Thus, the join results $q(\boldsymbol{R})$ can be enumerated with constant delay by performing a pre-order traversal along the join tree T equipped with appropriate hash tables. However, the algorithm fails on CQCs, because the semi-join reductions cannot ensure that the tuples will satisfy the comparisons. Thus, during the enumeration process, some tuples that do not satisfy the comparisons need to be skipped, breaking the $\tilde{O}(1)$-delay requirement.

Example 4.2. Figure 3(a) illustrates the issue for the query in Example 1.2 using the join tree R_1-R_2-R_3. The tuples in white are those after the semijoin reduction. However, the tuple $(3,2)$ in R_1, $(2,1)$ in R_2, and $(1,0)$ in R_3 do not appear in any valid query results due to the predicate $x_1 \le x_4$. If these tuples are not skipped during



(a) Standard semijoin reduction

(b) New Approach

Figure 3: A running example for the query in Example 4.2.

the enumeration, this could lead to an $O(N)$-delay. For example, starting from the tuple $(2, 1)$ in R_2, the pre-order traversal has to check all tuples in R_1 against $(1, 0)$ in R_3 without enumerating any valid query results. □

5 FULL ACYCLIC CQCs

In this section, we address the issue in Example 4.2 and present a CDE algorithm for a full acyclic CQC q. Our algorithm will use a series of reductions. For each reduction $q \to q', \boldsymbol{R} \to \boldsymbol{R}'$, we will ensure that

(1) q' is still an acyclic CQC;
(2) $\boldsymbol{R}'$ can be computed from $\boldsymbol{R}$ in $\tilde{O}(N)$ time; and
(3) given a CDE structure of $q'(\boldsymbol{R}')$ and some other data structures on $\boldsymbol{R}$ that can be built in $\tilde{O}(N)$ time, we can enumerate $q(\boldsymbol{R})$ with delay $\tilde{O}(1)$;

The base case is when q has only one relation and no comparisons, for which the CDE structure is just the relation itself.

The simplest reduction is to remove all self-comparisons (i.e., type-1 predicates). For this reduction, q' is just q after dropping all self-comparisons, while $\boldsymbol{R}'$ is $\boldsymbol{R}$ after filtering each relation with all the self-comparisons on that relation. This clearly satisfies the three properties above. We will always perform this reduction when applicable. Thus, when describing the other reduction rules below, we may assume that q has no self-comparisons. Other reductions will each remove one relation from q, thus it takes at most $2(n-1)$ reductions to reach the base case for a CQC over n relations. As n is taken as a constant, the overall preprocessing cost would be $\tilde{O}(N)$ and the enumeration delay would be $\tilde{O}(1)$.

5.1 Reducible Relations

Given an acyclic CQC q and a join tree T, a leaf node (relation) R is one with only one neighbor in T. That neighbor is called its *parent*, denoted R_p.

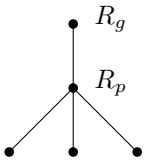
We will always perform a reduction from a *reducible* relation, defined as follows.

Definition 5.1. For an acyclic CQC q and a join tree T supporting its comparisons, a relation R is *reducible* if (1) R is a leaf in T; and (2) R is incident to at most one long comparison.

The following structural result is important for our development.

Lemma 5.2. *Given any acyclic CQC q and any join tree T supporting its comparisons, there exists a reducible relation.*

Proof. The lemma is trivially true if q has only two relations, since there are no long comparisons. When T has at least 3 nodes, it is easy to see that T must have a substructure shown in the figure below, where the nodes in $\boldsymbol{R}_s$ are all leaves, $|\boldsymbol{R}_s| \geq 1$, R_p is their common parent, and R_g is another node adjacent to R_p, but not in $\boldsymbol{R}_s$. Note that R_g might also be a leaf. We will show that at least one node in $\boldsymbol{R}_s$ is reducible.



First, if there is a node in $\boldsymbol{R}_s$ with at most one long comparison, then that node is reducible by definition. Below, we show that the other case, i.e., every node in $\boldsymbol{R}_s$ has ≥ 2 long comparisons, is not possible.

Recall that the comparison hypergraph $C(q, T)$ is Berge-acyclic. In $C(q, T)$, a hyperedge corresponds to a comparison and a vertex in $C(q, T)$ corresponds to an edge in T. Let C' be the subgraph of $C(q, T)$, whose vertices are just (R_g, R_p) and (R_p, R_s) for all $R_s \in \boldsymbol{R}_s$, and whose hyperedges are only those corresponding to long comparisons. Berge-acyclicity is monotone, i.e., any subgraph of a Berge-acyclic hypergraph is still Berge-acyclic, so C' is also Berge-acyclic (note that α-acyclicity is not monotone).

Consider the following cases:

Case (1): At least two long comparisons (hyperedges) from $\boldsymbol{R}_s$ contain the vertex (R_g, R_p). This contradicts the Berge-acyclicity of C', since that would cause each edge on the join tree to be connected to at least two comparisons, and each comparison to be connected to at least two edges, making the comparison hypergraph C' Berge-cyclic.

Case (2): No long comparisons (hyperedges) from $\boldsymbol{R}_s$ contain the vertex (R_g, R_p). This implies that $|\boldsymbol{R}_s| \geq 2$, while each long comparison is between two nodes in $\boldsymbol{R}_s$. Now consider the subgraph of C' containing only the vertices (R_p, R_s) for all $R_s \in \boldsymbol{R}_s$ and these long comparisons. Each hyperedge (now an ordinary edge) contains just two vertices while every vertex has degree ≥ 2, so this becomes an ordinary graph with $|\boldsymbol{R}_s|$ vertices and $\geq |\boldsymbol{R}_s|$ edges, which must cyclic, contradicting the Berge-acyclicity of C' again.

Case (3): Exactly one long comparison from $\boldsymbol{R}_s$ contains the vertex (R_g, R_p). Similar to Case (2), consider the subgraph of C' containing only the vertices (R_p, R_s) for all $R_s \in \boldsymbol{R}_s$ and the long comparisons in $\boldsymbol{R}_s$. The only difference is that one node has degree

≥ 1 while the rest has degree ≥ 2. Nevertheless, this still means that there are at least $|\boldsymbol{R}_s|$ edges, as needed for the contraction. □

Let R be a reducible relation. We apply different reductions depending on the number of incident comparisons on R, as described below.

5.2 No Incident Comparisons

If R has no incident comparisons, then we perform a standard semijoin reduction as in the Yannakakis algorithm, i.e., we replace R_p with $R'_p := R_p \ltimes R$ and then remove R. The correctness of this reduction has been proved in [3], but we rephrase the arguments under our framework, i.e., it satisfies the three properties stated at the beginning of Section 5. Property (1) follows from the fact that $T' = T - \{R\}$ is a valid join tree of q'. Property (2) is obvious. For property (3), suppose we have a CDE structure for $q'(\boldsymbol{R}')$. Note that R is not in $\boldsymbol{R}'$, but the semijoin ensures that every query result in $q'(\boldsymbol{R}')$ can join with at least one tuple in R. Thus, after enumerating a tuple t' from $q'(\boldsymbol{R}')$, all we have to do is to enumerate $R \bowtie t'$. This can be done with constant delay by building a hash table on R using $\bar{z} := var(R) \cap var(R_p)$ as the key. This hash table is the "other data structures" needed in property (3), which can be built in $O(N)$ time.

5.3 One Incident Comparison

Suppose $C : f(var(R)) \leq g(\bar{x}_b)$, for some $b \in [n]$, is the only comparison incident to R, which may be either long or short. Define

$$R'_p(var(R_p), \mathrm{mf}) := \left\{ \left(t_p, \min_{t \in R, t \bowtie t_p \neq \emptyset} f(t) \right) \middle| t_p \in R_p \right\}, \qquad (3)$$

where mf is a new helper attribute. We perform the reduction:

- $\boldsymbol{R} \to \boldsymbol{R}'$: replace R_p with R'_p;
- $q \to q'$: drop R, and replace C with $C' : \mathrm{mf} \leq g(\bar{x}_b)$.

Note that since mf is an attribute in R'_p, comparison C' is now between R'_p and R'_b. If C is a short comparison, C' will become a self-comparison. If so, we will apply the self-comparison-removal reduction immediately.

Now we argue for the correctness of this reduction.

Property (1): Property (1) follows from the same reason as in Section 5.2. The only difference is that the reduction in Section 5.2 does not change the attributes of any relation, while here we have added a new attribute mf to R'_p. But since this new attribute does not appear in any other relation, $T' = T - \{R\}$ is still a valid join tree of q'. The comparison hypergraph $\mathcal{C}(q, T)$ is still Berge-acyclic, because the reduction removes one edge from T and shortens one comparison. This corresponds to removing one node from $\mathcal{C}(q, T)$ and shrinking a hyperedge. So $\mathcal{C}(q', T')$ is still Berge-acyclic due to monotonicity.

Property (2): To achieve property (2), we cannot compute R'_p by definition. Instead, we first hash R using $\bar{z} := var(R) \cap var(R_p)$ as the key. For each unique key $\bar{z} = \kappa$, we keep the minimum $f(t)$ over all tuples $t \in R$ such that $t(\bar{z}) = \kappa$. Then, for each $t_p \in R_p$, we can look up the hash table to find the corresponding value for mf. This process takes $O(|R| + |R_p|)$ time.

Property (3): Suppose we have a CDE structure for $q'(\boldsymbol{R}')$. We, in addition, create a hash table on R using $\bar{z}$ as the key, and the value associated with $\bar{z} = \kappa$ is a list of tuples $t \in R$ such that $t(\bar{z}) = \kappa$, stored in the increasing order of $f(t)$. This hash index can be computed in $O(|R| \log |R|)$ time. After enumerating a query result $t' \in q'(\boldsymbol{R}')$, we look up the hash index for the list associated with $\kappa = t'(\bar{z})$. Since t' must satisfy the comparison $\mathrm{mf} \leq g(var(R'))$ in q', while the first tuple t in the list has $f(t) = \mathrm{mf}$, $t \bowtie t'$ must be a valid query result of $q(\boldsymbol{R})$, which will be enumerated (to be technically correct, we should enumerate $\pi_{var(q)}(t \bowtie t')$ to project out the helper attribute mf). Then, we scan the list in the increasing order of $f(\cdot)$, enumerating $t \bowtie t'$ for every t in the list until we encounter a t with $f(t) > g(t')$. When this happens, we stop the enumeration and move on to the next tuple from $q'(\boldsymbol{R}')$.

We need to show that the enumeration algorithm above is sound and complete. The soundness is obvious. For completeness, consider any query result $r \in q(\boldsymbol{R})$. Let $\kappa = r(\bar{z})$, $t = r(var(R))$, and $t' = r(var(q')) \bowtie (\mathrm{mf} = \min_{s \in R, s(\bar{z}) = \kappa} f(s))$. Note that $t \in R$. Because r satisfies C, i.e., $f(t) \leq g(t')$, and $f(t) \geq t'(\mathrm{mf})$, we have $t'(\mathrm{mf}) \leq g(t')$, i.e., t' must satisfy C'. Thus $t' \in q'(\boldsymbol{R}')$, so it will be enumerated from the CDE structure of $q'(\boldsymbol{R}')$. When this happens, t will be retrieved from the list associated with κ and $r = t \bowtie t'$ will be enumerated.

To see that the delay is $O(1)$, observe that for every $t' \in q'(\boldsymbol{R}')$, we examine $k+1$ tuples in the list while enumerating k query results, for some $k \geq 1$. Thus, the delay is $O(1)$. Note that it is important to ensure $k \geq 1$; otherwise, we may see an unbounded number of $t' \in q'(\boldsymbol{R}')$ without enumerating any query result from $q(\boldsymbol{R})$.

For the symmetric case where C is $f(\bar{x}_a) \leq g(var(R))$ for some $a \in [n]$, we change $\min f(t)$ to $\max g(t)$ in (3), and the list associated with each $\bar{z} = \kappa$ will be stored in the decreasing order of $g(\cdot)$.

Example 5.3. Figure 3(b) illustrates how the reduction works on the query in Example 1.2. Suppose we reduce R_1 first (the other reducible relation is R_3). This appends helper attribute mf_1 to R_2 while removing the tuple $(1, 2)$ in R_2. Suppose we reduce R_3 next (we could also reduce R_2, which is now reducible). This appends mf_2 to R_2. After this step, the comparison becomes a self-comparison $\mathrm{mf}_1 \leq \mathrm{mf}_2$. The next immediate reduction checks this self-comparison on R_2, removing the tuple $(2, 1)$. Now we have reached the base case with only R_2 and no comparisons.

To enumerate the query results, we rewind the reductions. After enumerating each tuple from R_2 (in this example, only one tuple remains in R_2), we first find all tuples in R_1 with $x_2 = 2$ and $x_1 \leq 2$. By using the hash table of R_2 on x_2 and visiting the tuples in sorted order of x_1, these tuples can be retrieved with constant delay. Then, for each partial join result from $R_1 \bowtie R_2$, we find all join tuples in R_3. For the last step, no more comparison needs to be checked. □

5.4 Two or More Incident Comparisons

Now we consider the general case. Let R be a *reducible* relation with $d \geq 2$ incident comparisons, which include at most one long comparison. Let $C_1, \ldots, C_d$ be the d comparisons incident on R. Without loss of generality, we assume each C_j has the form $f_j(var(R)) \leq g_j(\bar{x}_{b_j})$, where $b_j \in [n]$ for each $j \in [d]$. If any C_j

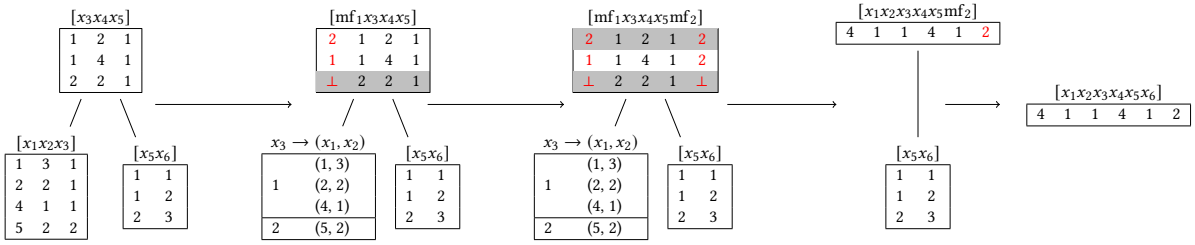


Figure 4: A running example for the query in Example 5.4

has the form $f_j(\bar{x}_{a_j}) \le g_j(var(R))$, the case can be handled symmetrically. Suppose C_1 is the only comparison that might be long, while $C_2, \ldots, C_d$ are all short comparisons. The reduction is similar to the one-incident-comparison case. Define

$$R'_p(var(R_p), \text{mf}) := \left\{ \left(t_p, \min_{t \in R, \sigma_{C_2 \wedge \cdots \wedge C_d} t \bowtie t_p \neq \emptyset} f_1(t) \right) \middle| t_p \in R_p \right\}, \tag{4}$$

where mf is a new helper attribute. Note that $C_2, \ldots, C_d$ are all short comparisons, i.e., they are between t and t_p, so the selection condition in (4) is well defined. In particular, if for a $t_p \in R_p$, no t satisfies the condition under the min, t_p will not be included in R'_p.

The reduction is defined as:

- $\boldsymbol{R} \to \boldsymbol{R'}$: replace R_p with R'_p;
- $q \to q'$: drop R and $C_2, \cdots, C_d$, and replace C_1 with C'_1 : $\text{mf} \le g_1(\bar{x}_{b_1})$.

Again, C'_1 may be a self-comparison (if C_1 is short), which would be immediately removed next.

Property (1) follows from the same reasoning as in Section 5.3. However, it requires more work to achieve property (2) and (3).

Property (2): To compute R'_p efficiently, we as before first hash R on $\bar{z}$. However, unlike the one-incident-comparison case, here it is no longer sufficient to just keep the minimum $f(t)$ for each unique key $\bar{z} = \kappa$, due to the additional comparison conditions $C_2 \wedge \cdots \wedge C_d$ imposed on $t \bowtie t_p$ in (4). Instead, we treat each tuple $t \in R$ as a $(d-1)$-dimensional point $p(t) := (f_2(t), \cdots, f_d(t))$ with weight $w(t) := f_1(t)$. For each tuple $t_p \in R_p$, we treat it as an orthogonal range query $B(t_p) := (-\infty, g_2(t_p)] \times \cdots \times (-\infty, g_d(t_p)]$. Now, we see that the mf value for each $t_p \in R_p$ is exactly the answer to the range-aggregation query $B(t_p)$ on the points $\{p(t) \mid t \in R, t \bowtie t_p \neq \emptyset\}$, where we use min as the semigroup addition.

This observation immediately leads to the following algorithm to compute R'_p: We, as before, hash R on $\bar{z}$. For each unique key $\bar{z} = \kappa$, we build a $(d-1)$-dimensional orthogonal range searching structure on $\{p(t) \mid t \in R, t(\bar{z}) = \kappa\}$. The cost to build these range searching structures is $O(|R| \log^{\max\{d-2,1\}} |R|)$. Next, for each $t_p \in R_p$, we perform a range-min query on $\{p(t) \mid t \in R, t(\bar{z}) = t_p(\bar{z})\}$, which takes time $O(|R_p| \log^{\max\{d-2,1\}} |R|)$ in total.

Property (3): Suppose we have a CDE structure for $q'(\boldsymbol{R'})$. We in addition create a hash table on R using $\bar{z}$ as the key, and the value associated with $\bar{z} = \kappa$ is a d-dimensional orthogonal range searching structure on $\{p(t) \mid t \in R, t(\bar{z}) = \kappa\}$, where $p(t)$ is now a d-dimensional point $\hat{p}(t) := (f_1(t), \cdots, f_d(t))$. The cost of building these structures is $O(|R| \log^{\max\{d-1,1\}} |R|)$. After enumerating a $t' \in q'(\boldsymbol{R'})$, we perform a range-reporting query on the range searching structure associated with $\kappa = t'(\bar{z})$ using range $\hat{B}(t') = (-\infty, g_1(t')] \times \cdots \times (-\infty, g_d(t')]$. For each t reported, we enumerate $t \bowtie t'$.

The soundness and completeness of this enumeration algorithm follow from similar reasons as in Section 5.3. To bound the delay, the key observation is that each range-reporting query $\hat{B}(t')$ must return at least one tuple. Specifically, the tuple $t_{\min} = \arg\min_{t \in R, t(\bar{z}) = t'(\bar{z})} f(t)$ must be inside the range, i.e., it satisfies all the constraints $f_j(t) \le g_j(t')$. To see this, let $t'_p = t'(var(R'_p)) \in R'_p$ and $t_p = t'_p(var(R_p)) \in R_p$. Note that t'_p just has one extra attribute mf compared with t_p. When constructing R'_p, we performed a range-min query with $B(t_p)$ on $\{t \in R \mid t(\bar{z}) = t'(\bar{z})\}$. Recall that $C_2, \ldots, C_d$ are all short comparisons, so $g_j(t') = g_j(t_p)$, which means that $\hat{B}(t')$ is the same as $B(t_p)$ except that the former has an extra constraint $f_1(t) \le g_1(t')$. Furthermore, $t'_p(\text{mf})$ takes the minimum $f_1(t)$ among all tuples in $\{t \in R \mid t(\bar{z}) = t'(\bar{z})\}$. Thus, if $t_{\min}$ were outside $\hat{B}(t')$, no other tuple would be inside. This means that $B(t_p)$ would have been empty, in which case t'_p would not have been included in R'_p and t' would not be enumerated from $q'(\boldsymbol{R'})$. Therefore, the range query spends $O(\log^{d-1} |R| + k)$ time to enumerate k query results for some $k \ge 1$. So the delay is $O(\log^{d-1} |R|)$.

Example 5.4. Consider the following query with 2 incident comparisons on R_1:

$$R_1(x_1, x_2, x_3) \bowtie R_2(x_3, x_4, x_5) \bowtie R_3(x_5, x_6), x_1 \le x_4, x_2 < x_6$$

Figure 4 shows a running example of this query. Suppose we reduce R_1 first. We first build a 1D range searching structure on R_1 for each unique x_3 such that, given any $t = (x_3, x_4, x_5) \in R_2$, we can find the minimum x_2 in R_1 with a matching x_3 while satisfying the comparison $x_1 \le x_4$. This becomes the helper attribute mf_1 in R_2. Note that the tuple $(2, 2, 1)$ in R_2 has $\text{mf}_1 = \perp$ as no tuple in R_1 satisfies $x_3 = 2$ and $x_1 \le x_4$. Now we drop R_1 and $x_1 \le x_4$, while rewriting $x_2 < x_6$ into $\text{mf}_1 < x_6$. Next, we reduce R_3 as in the one-incident-comparison case, which will append mf_2 to R_2. Now we drop R_3 and rewrite $\text{mf}_1 < x_6$ into a self-comparison $\text{mf}_1 < \text{mf}_2$, reaching the base case.

To enumerate the query results, we rewind the reductions. Starting from each tuple in R_2, we first find all join tuples in R_1 with a matching x_3 while satisfying $x_1 \le x_4$, using the range searching structures. Then, for each partial join result in $R_1 \bowtie R_2$, we find all join tuples in R_3 using a hash table and visiting the tuples in sorted order of x_6.

5.5 Putting Things Together

For a given acyclic CQC q and a join tree T supporting its comparisons, we perform a series of reductions, each on an arbitrarily chosen reducible relation R. It should be clear that d, the number of comparisons incident to R is never larger than d_q, the degree of q. This is because each reduction reduces one leaf node of T, and shrinks one long comparison, while dropping a number of short comparisons.

To analyze the total cost, recall the following results from the previous subsections.

(1) if $d = 0$, the preprocessing takes $O(N)$ time, and the enumeration delay is $O(1)$;
(2) if $d = 1$, the preprocessing takes $O(N \log N)$ time, and enumeration delay is $O(1)$;
(3) if $d \geq 2$, the preprocessing takes $O(N \log^{d-1} N)$ time, and enumeration delay is $O(\log^{d-1} N)$.

After a series of reductions, the preprocessing times and the enumeration delays add up. But since the query size is considered as a constant, this does not affect the asymptotic result, summarized as follows.

Theorem 5.5. *A full acyclic CQC q with degree d_q can be enumerated with delay $O(\log^{\max\{d_q-1,0\}} N)$ delay after $O(N \log^{d_q - \mathbb{I}(d_q \geq 2)} N)$ preprocessing time*[6].

Corollary 5.6. *A full acyclic CQC q can be computed in $\tilde{O}(N + \text{OUT})$ time.*

5.6 Hard Queries

In this section, we argue that the neither the α-acyclicity condition for the relational hypergraph nor the Berge-acyclicity of the comparison graph can be removed, if one aims at the $\tilde{O}(N + \text{OUT})$ running time for CQCs. For the first condition, it is well known that if a CQ is not α-acyclic, it must be at least as hard as the triangle query, which has a lower bound:

Theorem 5.7 ([20]). *The triangle query*

$$R_1(x_1, x_2), R_2(x_2, x_3), R_3(x_1, x_3)$$

requires time $\Omega(N^{4/3-\varepsilon})$ for every constant $\varepsilon > 0$ when $\text{OUT} \geq N$*, under the 3SUM conjecture.*

Since CQs are special CQCs, a CQC whose relational hypergraph is not α-acyclic is thus at least as hard as the triangle query. For the second condition, consider the following query:

Example 5.8. Consider the query

$$R_1(x_1, x_2), R_2(x_2, x_3), R_3(x_3, x_4), C_1 : x_1 \leq x_4, C_2 : x_1 \geq x_4.$$

It is clear that its relational hypergraph is α-acyclic. It has only one join tree T: R_1-R_2-R_3. On this join tree, its comparison hypergraph has two hyperedges, both containing the two edges of T. Note that this comparison hypergraph is α-acyclic but not Berge-acyclic. On the other hand, it is easy to see that this query is equivalent to the triangle query. □

[6] $\mathbb{I}(\cdot)$ is the indicator function.

The query above implies that the second condition cannot be dropped or relaxed to α-acyclicity. Note that, however, this only means that some CQCs not satisfying our acyclic conditions are hard (i.e., cannot be solved in $\tilde{O}(N + \text{OUT})$ time). It does not mean that *every* CQC not satisfying our acyclic conditions is hard. In fact, if we change C_2 in Example 5.8 to $x_1 \leq x_4 + 1$, this will make the query easy as C_2 has become a redundant comparison, although syntactically, it is still not acyclic by our definition. This means that our acyclic condition is a sufficient, but not necessary, condition for a CQC to be easy. In practice, one may use some query rewrite rules to take care of some common cases, e.g., removing redundant comparisons, rewriting or decomposing a comparison (see the discussion following Example 3.1), and see if the resulting CQC is acyclic.

6 NON-FULL CQCs

We consider non-full CQCs in this section. We use $q(\boldsymbol{R})$ to denote the results of evaluating q on $\boldsymbol{R}$ when the output attributes are the given $\bar{y}$, and $q^*(\boldsymbol{R})$ to denote the full query results, i.e., when the output attributes are $var(q)$.

Free-connex CQs. A naive way to evaluating such a non-full CQ or CQC q is to first compute $q^*(\boldsymbol{R})$, and then perform a (distinct) projection. This would take time $\tilde{O}(N + |q^*(R)|)$, assuming q^* is acyclic. However, as $\text{OUT} = |q(R)| \ll |q^*(\boldsymbol{R})|$ due to the projection, this method can lead to running times much longer than necessary. It does not guarantee any bounded-delay enumeration, either. Ideally, like on full queries, one would want $\tilde{O}(1)$-delay after $\tilde{O}(N)$-time preprocessing for non-full queries as well. Unfortunately, this is not achievable when the output attributes can be an arbitrary subset of $var(q)$, even without comparisons. In fact, the query $ans(x_1, x_3) \leftarrow R_1(x_1, x_2), R_2(x_2, x_3)$ already encompasses Boolean matrix multiplication, and the best algorithm for this problem is not significantly better than the naive algorithm above. Nevertheless, the literature has identified a subclass of non-full CQs that can be enumerated with $\tilde{O}(1)$ delay after $\tilde{O}(N)$-time preprocessing, known as *free-connex* CQs [3, 12].

There are a number of equivalent definitions of free-connex CQs, and we adopt the following one. Given a CQ q on an database $\boldsymbol{R}$, we can add an auxiliary relation $\hat{R}_i := \pi_{\bar{z}} R_i$ to $\boldsymbol{R}$ and correspondingly the atom $\hat{R}_i(\bar{z})$ to q, where $\bar{z}$ is any subset of $var(R_i)$. It is obvious that adding such a relation does not change the query results. We thus obtain an *extended* query after adding any number of such auxiliary relations. A CQ q is *free-connex* if it has an extended query $\hat{q}$ that admits a join tree $\hat{T}$ with a designated root node, such that (1) all nodes $R \in \hat{T}$ where $var(R) \subseteq \bar{y}$ form a connected component in $\hat{T}$ containing its root (they are said to form the *connex subset*), and (2) $\bar{y} = \cup_R var(R)$ where the union is over all R in the connex subset.

Free-connex CQCs. We extend the definition above to CQCs. Similarly, a CQC q is first extended to some $\hat{q}$ by adding any number of auxiliary relations. We say that q is a free-connex CQC if $\hat{q}$ has a join tree $\hat{T}$ that, in addition to conditions (1) and (2) above, also satisfies (3) $C(\hat{q}, \hat{T})$ is Berge-acyclic (i.e., $\hat{T}$ *supports* $\hat{q}$). When deciding the incident relations of each comparison, we as before choose the (a_j, b_j) pair such that $P_{\hat{T}}(a_j, b_j)$ is the shortest in $\hat{T}$. Note that this

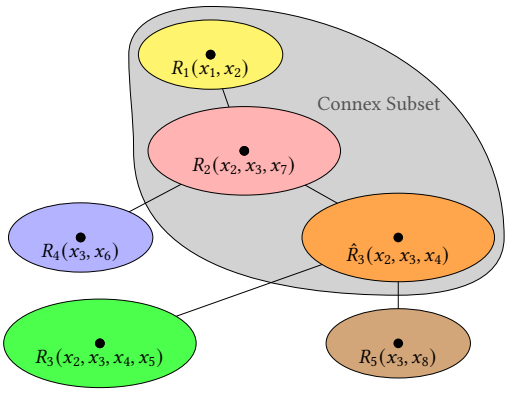


Figure 5: A free-connex join tree for the extended query.

means that some of the comparisons will be incident to the auxiliary relations. The degree of a free-connex CQC q, still denoted as d_q, is the minimum degree of $C(\hat{q}, \hat{T})$ over all join trees $\hat{T}$ supporting some extended query $\hat{q}$ of q. Note that a free-connex CQ is just a special free-connex CQC with degree 0.

Example 6.1. The query from Example 3.1 is a free-connex CQC, with an extended query being

$$\begin{aligned} ans(x_1, x_2, x_3, x_4, x_7) \leftarrow & R_1(x_1, x_2), R_2(x_2, x_3), \hat{R}_3(x_2, x_3, x_4) \\ & R_3(x_2, x_3, x_4, x_5), R_4(x_3, x_6), R_5(x_3, x_8) \\ & C_1 : x_1 - x_2 \le x_3 x_4 + 2, \\ & C_2 : \min\{2x_2, x_7\} \le x_6, \quad C_3 : x_2 \le x_8, \end{aligned}$$

and the supporting join tree $\hat{T}$ is shown in Figure 5, whose degree is 1. Note that when using this join tree, C_1 is incident on R_1 and the auxiliary relation $\hat{R}_3$. □

Given a free-connex CQC q, we first extend it to $\hat{q}$ as described above. As argued earlier, q and $\hat{q}$ have the same query answer, so it suffices to describe how to solve $\hat{q}$. To simplify notation, below we still use q to denote the extended query, and T its supporting join tree.

We follow the same framework as for full acyclic CQCs, and perform a series of reductions. Note that for a full query, the join tree T does not have a designated root, but it does for a free-connex query by definition. A reducible relation in this case is still defined as a leaf relation in T that has at most one incident long comparison. As T is rooted, its root is not a leaf by definition (unless T has only one node), so it is not reducible even if it has only one incident long comparison. Nevertheless, although Lemma 5.2 is stated for unrooted join trees, it is straightforward to verify that the proof also works for rooted join trees, and a reducible relation still exists for any free-connex CQC.

As before, each reduction will remove one reducible relation. Given a reducible relation R with d incident comparisons $C_1, \cdots, C_d$, we perform the same reduction as in Section 5.4, and we show below how the 3 properties are satisfied. First, property (1) shall be changed into "q' is a free-connex CQC". This follows easily from the fact that if R is in the connex subset, so is its parent R_p. Property (2) still holds because we use the same algorithm to compute $\boldsymbol{R}'$.

To preserve property (3), we need to differentiate the cases depending on whether R belongs to the connex subset or not. Recall that R belongs to the connex subset iff $var(R) \subseteq \bar{y}$.

Case (1): R belongs to the connex subset. For this case, we build the same data structures, and then use the same algorithm to enumerate $q(\boldsymbol{R})$ from a CDE structure for $q'(\boldsymbol{R}')$.

Case (2): R does not belong to the connex subset. In this case, we can actually show that $q(\boldsymbol{R}) = q'(\boldsymbol{R}')$, thus property (3) is trivially satisfied without the need to build additional data structures and further enumeration.

If R does not belong to the connex subset, then q' has the same output variables $\bar{y}$ as q. Consider the full queries q^* and q'^* corresponding to q and q', respectively. We know that property (3) holds on q^* and q'^*, which implies that $\pi_{\bar{y}} q^*(\boldsymbol{R}) = \pi_{\bar{y}} q'^*(\boldsymbol{R}')$. Thus, we have $q(\boldsymbol{R}) = \pi_{\bar{y}} q^*(\boldsymbol{R}) = \pi_{\bar{y}} q'^*(\boldsymbol{R}') = q'(\boldsymbol{R}')$, as desired.

Therefore, the complexity of free-connex CQCs is the same as that of full acyclic CQCs. In practice, free-connex CQCs can be evaluated even faster due to the simplification in case (2) above.

Theorem 6.2. *A free-connex CQC q with degree d_q can be enumerated with delay $O(\log^{\max\{d_q - 1, 0\}} N)$ delay after $O(N \log^{d_q - \mathbb{I}(d_q \ge 2)} N)$ preprocessing time.*

7 GENERAL CQCs

7.1 Generalized Hypertree Decompositions

Generalized hypertree decompositions (GHDs) [9] provide a powerful framework for dealing with general CQs [14] and CQCs [15]. By putting multiple relations into a *bag*, GHDs convert a non-acyclic or non-free-connex query into an acyclic free-connex one. The overhead is a larger preprocessing time, since each bag must be precomputed. Thus, one should use the GHD that minimizes the maximum precomputation time over all bags, which leads to various definitions of *width*.

More formally, Khamis et al. [15] show that a CQC q can be enumerated with $\tilde{O}(1)$ delay after $\tilde{O}(N^{\mathsf{width}(q)})$ preprocessing time, where

$$\mathsf{width(q)} = \min_{\mathcal{T} \in \mathcal{G}(q)} \max_{v \in \mathcal{T}} w(v).$$

Here, $\mathcal{G}(q)$ denotes the set of all GHDs of q that has only short comparisons, and $w(v)$ is the *width* of a bag v in the GHD $\mathcal{T}$. By using our algorithm to compute the GHD, we achieve $\tilde{O}(1)$ delay after $\tilde{O}(N^{\mathsf{width}^*(q)})$ preprocessing time, where

$$\mathsf{width}^*\mathsf{(q)} = \min_{\mathcal{T} \in \mathcal{G}^*(q)} \max_{v \in \mathcal{T}} w(v),$$

where $\mathcal{G}^*(q)$ is now the set of GHDs of q that meet our acyclic conditions. Since $\mathcal{G}(q) \subseteq \mathcal{G}^*(q)$, $\mathsf{width}^*(q) \le \mathsf{width}(q)$ for any q. The actual improvement depends on the query q, as well as $w(v)$, which in turns depends on the given degree constraints of the input (including cardinality constraints, functional dependencies, and PK constraints). The exact definition of $w(v)$ is very technical; below we illustrate the improvements on a few representative examples.

Example 7.1. Consider the following CQC:

$$\begin{aligned} & R_1(x_1, x_2), R_2(x_2, x_3), R_3(x_3, x_4), R_4(x_4, x_5), \\ & C_1 : x_1 \le x_4, C_2 : x_1 \ge x_5. \end{aligned}$$

This CQC is not acyclic (the comparison hypergraph is not Berge-acyclic). One valid GHD is $u_1 = \{R_1, R_2\}$, $u_2 = \{R_3\}, u_3 = \{R_4\}$. On this GHD, C_1 is incident to u_1 and u_2, C_2 is incident to u_1 and u_3, so the comparison hypergraph is now Berge-acyclic. After precomputing $R_1 \bowtie R_2$, the query becomes an acyclic CQC, which can be handled by our algorithm. The preprocessing time increases to $\tilde{O}(N + |R_1 \bowtie R_2|)$, which is $\tilde{O}(N^2)$ in the worst case. If x_2 is a primary key (PK) of R_1 or R_2, then the preprocessing time becomes $\tilde{O}(N)$. On the other hand, this GHD cannot be used in [15]. They can only use $u_1 = \{R_1\}, u_2 = \{R_2, R_3, R_4\}$; $u_1 = \{R_1, R_2\}, u_2 = \{R_3, R_4\}$; or $u_1 = \{R_1, R_2, R_3\}, u_2 = \{R_4\}$, all of which lead to $\tilde{O}(N^2)$ preprocessing time, even if x_2 is a PK. □

Example 7.2. Consider the following CQC, which finds all "dumbbells" in a graph, whose edges are stored in a relation R (see Figure 6(b) for the hypergraph representation). We impose a comparison involving the weights associated with the edges of the two triangles that make up the dumbbell.

$$R(x_1, x_2, w_1), R(x_2, x_3, w_2), R(x_1, x_3, w_3), R(x_3, x_4),$$
$$R(x_4, x_5, w_4), R(x_5, x_6, w_5), R(x_4, x_6, w_6),$$
$$w_1 w_2 w_3 \le w_4 w_5 w_6.$$

This CQC is not acyclic due to two reasons: (1) the relational hypergraph is not α-acyclic, and (2) the comparison is not in the required form where either side should be defined on variables from one relation. Nevertheless, we can group the 7 relations (actually, 7 logical copies of the same physical relation) into 3 bags: two triangles and the "handle" of the dumbbell. Each triangle join can be computed in $O(N^{1.5})$ time [19], after which we apply our algorithm on the GHD, which is the same as Example 4.2, by treating $w_1 w_2 w_3$ and $w_4 w_5 w_6$ as new attributes of the two triangle bags. On the other hand, this CQC requires $\tilde{O}(N^2)$ time to preprocess in [15]. □

To keep the presentation accessible, we have only stated the general result where a single GHD is used. It has been shown [14, 15, 18] that the width can be further reduced by using multiple GHDs. Our algorithm offers improvements in this case as well.

Example 7.3. Revisit the query in Example 7.1. As mentioned, if there is no key constraint, our algorithm has $\tilde{O}(N^2)$ preprocessing time. It turns out that by decomposing the relations and using different GHDs for different parts, this can be further improved. For a variable x and a tuple t, let $\deg_R(t, x) = |\sigma_{x=t(x)}(R)|$ be the degree of t in R with respect to x. We partition the tuples of R_2 into the *heavy* ones and *light* ones: the former have $\deg_{R_2}(t, x_2) \ge \sqrt{N}$ while the latter $\deg_{R_2}(t, x_2) < \sqrt{N}$. For the light R_2 (together with R_1, R_3, R_4 in full), we use the same GHD $u_1 = \{R_1, R_2\}$, $u_2 = \{R_3\}, u_3 = \{R_4\}$ from Example 7.1. But now all tuples in R_2 are light, so we have $|R_1 \bowtie R_2| \le N^{1.5}$. For the heavy R_2, we use the GHD $u_1 = \{R_1\}, u_2 = \{R_2, R_3\}, u_3 = \{R_4\}$. Because there are at most $\sqrt{N}$ heavy values on x_2, we can bound $|R_2 \bowtie R_3|$ by $N^{1.5}$ as well. Thus, the total preprocessing time is $\tilde{O}(N^{1.5})$. Note that by setting R_4 to an identity relation (i.e., $x_4 = x_5$), this CQC degenerates into the triangle query in Example 5.8. This implies that the $\tilde{O}(N^{1.5})$ preprocessing time cannot be improved unless the triangle query can be improved, which is considered unlikely.

However, for this query, multiple GHDs do not help the algorithm of [15] because neither GHD used above is allowed in [15]. Interestingly, multiple GHDs do help them when x_2 is a PK of R_2 to reduce the time to $\tilde{O}(N^{1.5})$. However, as we see in Example 7.1, our algorithm can achieve $\tilde{O}(N)$ time with just one GHD in this case. □

The last example is on a non-full query:

Example 7.4. The following query is a non-full version of Example 7.1:

$$ans(x_2, x_4) \leftarrow R_1(x_1, x_2), R_2(x_2, x_3), R_3(x_3, x_4), R_4(x_4, x_5),$$
$$C_1 : x_1 \le x_4, C_2 : x_1 \ge x_5.$$

This query cannot be handled by our algorithm in Section 6 for two reasons: (1) the comparison hypergraph is not Berge-acyclic, and (2) it is not free-connex. However, we can use the GHD $u_1 = \{R_1\}, u_2 = \{R_2, R_3\}, u_3 = \{R_4\}$, which fixes the two issues simultaneously. The extended query of this GHD has an auxiliary relation $\hat{u}_2(x_2, x_4)$, which forms the free-connex subset. The preprocessing time increases to $\tilde{O}(N + |R_2 \bowtie R_3|)$, which is linear if x_3 is a PK of R_2 or R_3. In this case, the best time achievable in [15] is still $\tilde{O}(N^{1.5})$ while using two GHDs. □

7.2 Projection Expansion

Example 7.5. Consider the following non-full CQC:

$$q : ans(x_2, x_4) \leftarrow R_1(x_1, x_2, x_3), R_2(x_3, x_4, x_5), x_1 < x_5.$$

It is an acyclic CQC but not free-connex. Under the GHD framework, we would have to put R_1 and R_2 into one bag, thus this degenerates into the naive algorithm that first computes the full query, followed by a projection onto x_2, x_4. However, a better approach is to first compute the query

$$q' : ans(x_2, x_3, x_4) \leftarrow R_1(x_1, x_2, x_3), R_2(x_3, x_4, x_5), x_1 < x_5,$$

using our algorithm in Section 6, and then project onto x_2, x_4. Note that after adding x_3 to the output attributes, q' is now a free-connex CQC, so our algorithm can compute it in time $\tilde{O}(N + |q'(\boldsymbol{R})|)$, whereas using naive algorithm has running time of $\tilde{O}(N + |q^*(\boldsymbol{R})|)$, where q^* is the full query of q, where all attributes are output attributes. They are both larger than the true output size OUT = $|q(\boldsymbol{R})|$, but the former is certainly smaller on many instances. □

This simple optimization, which we call *projection expansion*, can be plugged into the GHD framework to handle more complicated queries. More generally, if the CQC associated with a bag of the GHD is acyclic but non-free-connex, we add a minimally necessary set of attributes to the projection so that the query becomes free-connex, and apply our algorithm in Section 6 instead of computing the full bag.

7.3 Pre-grouping

If the CQC associated with a GHD bag is non-acyclic, projection expansion cannot be applied. For this case, we design another optimization technique to avoid computing the full bag.

Example 7.6. Consider the following query:

$$ans(x_2, x_4) \leftarrow R_1(x_1, y_1, x_2, x_3), R_2(x_3, x_4, x_5, y_5), x_1 \le x_5, y_1 \le y_5.$$

Compared with Example 7.5, the comparison hypergraph of this query is not Berge-acyclic as it has two long comparisons. Making x_3 an output attribute does not help.

To deal with this query, we group the tuples in R_1 by (x_2, x_3). For each group, we build a 2D range query structure on the (x_1, y_1) pairs. Then we compute $J = \pi_{x_2,x_3} R_1 \bowtie R_2$. For each $(x_2, x_3, x_4, x_5, y_5) \in J$, we check if the range $(-\infty, x_5] \times (-\infty, y_5']$ contains any (x_1, y_1) pair in the range query structure associated with the group (x_2, x_3). If yes, we enumerate (x_2, x_4). The cost is thus $\tilde{O}(N + |J|)$. This is always is better than computing the full join, followed by checking the comparisons and projecting onto x_2, x_4, which has cost $\tilde{O}(N + |R_1 \bowtie R_2|)$. Note that we could also do the grouping on R_2, which leads to a cost of $|R_1 \bowtie \pi_{x_3,x_4} R_2|$. The better plan can be chosen using some join size estimation technqiues. □

8 EXPERIMENTS

8.1 Experimental Setup

Prototype implementation. We have implemented our algorithms in a system prototype on top of Spark [26], which we call *SparkCQC*. SparkCQC contains three components: a (standard) SQL parser, a query optimizer, and a core library. Recall that in each step of the reduction, we are free to choose any reducible relation. Our optimizer enumerates all possible reduction orders and tries to find the best plan.

The core library is written in Scala and contains functions that use standard RDD operations to implement the reduction/enumeration procedures described in the paper. This allows us to inherit all the benefits of Spark: good scalability through distributed processing, dynamic workload balancing, fault-tolerance, and the ability to work with a variety of data sources and sinks. For example, we could run a graph pattern query (with comparisons) over a graph stored in GraphX, or feed the query results of a CQC directly to Spark ML without writing to disk.

As a practical optimization, we did not use the multi-dimensional range searching structures. These data structures are needed to guarantee theoretical $\tilde{O}(N+\text{OUT})$ time, but we find that the hidden constant and logarithmic factors outweigh their benefits. Thus, we implemented a simpler 1D alternative: For the reduction phase, if there are two or more incident comparisons, we only choose one to perform the reduction while ignoring the rest; during the enumeration phase, we check all the neglected comparisons.

Query processing engines compared. We compare our algorithms with SparkSQL [2] and PostgreSQL. In order to get a sense of the hidden constant and logarithmic factors in the $\tilde{O}(N+\text{OUT})$ bound, for each query, we measured the time to read the input data from disk and write the output to disk. This I/O cost can be considered as the minimum cost required to answer the query. We also compare with the unranked version of [21] (called "Any-K"). Their algorithm only supports full CQCs with short comparisons, so we can only compare it on Query 6.

Experimental environment. All experiments were performed on a machine equipped with two Intel Xeon 2.1GHz processors each having 12 cores/24 threads, 416 GB memory, and 4x 4TB HDDs. The 4 HDDs were run over a RAID 5 with approximately 600MB/s read/write speed. The machine runs Linux, with Scala 2.12.13. The Spark version is 3.0.1 and the PostgreSQL version is 9.2.24. Each query was evaluated 10 times with each engine and we report the average running time. We ran each experiment with a 24-hour limit.

8.2 Datasets and Queries

We tested 5 graph pattern queries and 3 analytical queries described below. Figure 6 and Table 1 highlight their structures and characteristics.

Graph pattern queries. For graph pattern queries, we use some real graphs from SNAP (Stanford Network Analysis Project) [17]. Some statistics of these graphs are given in Table 2. We store the edges as a relation `G(src,dst)`, so a graph pattern query can be formulated as a CQ with self-joins on `G`. We created two other relations `O(node,deg)`, `I(node,deg)` that store the out-degrees and in-degrees of the nodes, respectively. The degrees will be used in the comparisons.

Q1 (SQL shown below) finds all length-3 paths `(A, B, C, D)` from the graph, such that the degree of `A` is less than the degree of `D`. This is an instantiation of Example 1.2[7].

```
SELECT G1.src as A, G2.src as B,
G3.src as C, G3.dst as D
FROM G G1, G G2, G G3, O O1, O O2
WHERE G1.dst = G2.src AND G2.dst = G3.src
AND G1.src = O1.node AND G3.dst = O2.node
AND O1.deg < O2.deg
```

For Q2, we perform the dumbbell query in Example 7.2. As mentioned, after first computing the two triangle queries (actually, we just need to compute it once), the query becomes Q1.

Q3 adds one more comparison to Q1, which requires the out-degree of `B` to be less than the in-degree of `D`. The added comparison is a short one, which preserves the acyclicity of the CQC, but turns it into a degree-2 query.

Q4 is a non-full version of Q1, where the output variables are set to `C,D`. Note that in SQL, this corresponds to changing the `SELECT` clause to `SELECT DISTINCT G3.src as C, G3.dst as D`.

Q5 is a non-full query featuring two long comparisons.

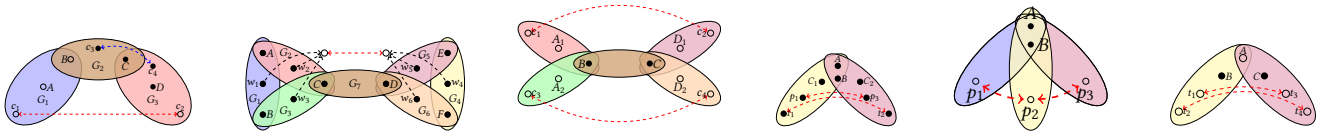
Analytical queries. We tested 3 analytical queries on TPC-E data. TPC-E is an online transaction processing benchmark that models a financial brokerage house.

Q6 (SQL shown below) is a self-join on the `Trade` relation that stores the trading information of customers. It finds all pairs of transactions that make a ≥20% profit within 90 days. Note that the two transactions must be made by the same customer (`CA_ID`) on the same stock (`S_SYMB`) for the profit to make sense. This is a degree-3 CQC. This query has only short comparisons, so it can be handled by Willard's algorithm [23] and Any-K [21]. Actually, our algorithm degenerates into Willard's algorithm on this query. However, we find that using 3-dimensional range query structures has poor practical performance, and the 1D alternative described above works better as verified by our experiments.

```
SELECT * FROM Trade T1, Trade T2
WHERE T1.TT = "BUY" and T2.TT = "SALE" and
T1.CA_ID = T2.CA_ID and T1.S_SYBM = T2.S_SYMB
and T1.T_DTS <= T2.T_DTS
and T1.T_DTS + interval '90' day >= T2.T_DTS
```

[7] Strictly speaking, this is a query over 5 relations, but we first compute `G1⋈O` and `G3⋈O` to turn the query into Example 1.2 for all query engines.

[8] For Q1 and Q3, all attributes are output attributes. The comparison between c_3 and c_4 only applies for Q3.



(a) Q1, Q3 and Q4[8] (b) Q2 (c) Q5 (d) Q6 (e) Q7 (f) Q8

Figure 6: The relational hypergraphs of queries. The dashed lines show the comparisons; solid dots are output attributes.

```
and T1.T_TRADE_PRICE*1.2 < T2.T_TRADE_PRICE
```

Q7 (SQL shown below) is also a self-join on the `Trade` relation, It finds all distinct pairs of customer and stock, such that the customer (`CA_ID`) had traded the stock (`S_SYMB`) at least three times, and the interval of each trade should be at least 90 days.

```
SELECT DISTINCT T1.CA_ID, T1.S_SYMB FROM
Trade T1, Trade T2, Trade T3
WHERE T1.S_SYMB = T2.S_SYMB
and T2.S_SYMB = T3.S_SYMB and
T1.CA_ID = T2.CA_ID and T2.CA_ID = T3.CA_ID
and T1.T_DTS + interval '90' day < T2.T_DTS
and T2.T_DTS + interval '90' day < T3.T_DTS
```

From TPC-E data, we created a new temporal relation `Hold(CK,SK,ST,ET)`. Each row indicates that a customer (`CK`) holds a security (`SK`) from start time (`ST`) to end time (`ET`). Q8 (SQL shown below) finds all pairs of customers who hold common securities within 10 days, as well as the number of such common securities. This is an instantiation of Example 7.6, where we can apply the pre-grouping technique. This technique actually answers the non-full query where the output attributes are `H1.CK,H2.CK,H1.SK`, so we just need another group-by aggregation to obtain the distinct count on `H1.SK`.

```
SELECT H1.CK, H2.CK, COUNT(DISTINCT H1.SK)
FROM Hold H1, Hold H2 WHERE H1.SK = H2. SK
and H1.ST < H2.ET - interval '10' day
and H2.ST < H1.ET - interval '10' day
and H1.CK <> H2.CK GROUP BY H1.CK, H2.CK
```

	acyclic	free-connex	full	degree	aggregation
Q_1	✓	✓	✓	1	
Q_2		✓	✓	1	
Q_3	✓	✓	✓	2	
Q_4	✓	✓		1	
Q_5	✓	✓		1	
Q_6	✓	✓	✓	3	
Q_7	✓	✓		1	
Q_8	✓			2	✓

Table 1: Characteristics of queries

8.3 Experimental Results

Running time comparison. Figure 7 shows the running times of the three systems on all tested queries. Q2 requires finding all the triangles first, so we only tested it on the smallest graph; the other graph pattern queries were tested on the 3 larger graphs. On the largest graph wiki, we used 16 workers; other experiments were done with a single worker. Missing results indicate that the system did not finish within the 24-hour time limit.

From the results, we can draw the following observations. (1) SparkCQC provides a speedup from 9x to 68x compared with Spark SQL, and 3x to 237x compared with PostgreSQL, even not considering some runs which did not finish within 24 hours. (2) In many cases, the running time of SparkCQC is close to the I/O time, which indicates that the constant factor in $\tilde{O}(N + \text{OUT})$ is actually quite small. (3) For Q8, the I/O time is much smaller, because Q8 is an aggregation query with a small output size. (4) For most queries, PostgreSQL has better performance than SparkSQL on a single worker, but SparkSQL will run faster with more workers.

Name	Edges	L3	Q1	Q3
Bitcoin	24,186	42,848,068	19,325,823	5,261,622
Epinions	508,837	3.7×10^9	2.0×10^9	1.2×10^9
Google	5,105,039	849,058,944	383,455,057	250,921,321
Wiki	28,511,807	2.4×10^{11}	1.6×10^{11}	9.0×10^{10}

Table 2: Graphs and their characteristics (L3 is the number of length-3 paths)

Selectivity. The selectivity of the comparison predicates is an important parameter for our algorithms, which directly affects OUT. However, it does not have a major impact on SparkSQL or PostgreSQL, as they cannot push down the predicates, except the short comparison in Q3.

We performed a set of experiments to verify this claim using Q1–Q3. We changed the comparisons in these queries to the form $f(\bar{x}) + k \le g(\bar{y})$, which leads to various selectivities by controlling the value of k. The experiment results are shown in Figure 8, where measure the selectivity as the ratio between OUT and the query size when $k = 0$. First, from the results we see that the running time of SparkCQC scales almost linearly as the selectivity, which is in turn proportional to OUT, which is expected as the algorithm runs in $\tilde{O}(N + \text{OUT})$ time and the output size often dominates the running time. On the other hand, SparkSQL and PostgreSQL cannot benefit from the smaller output size as claimed.

Parallel query processing. To verify the benefit of parallelism (among many others) of building our system on top of Spark, we assigned more cores to each system and re-ran Q1–Q3. The results are shown in Figure 9. As we can see, SparkCQC and SparkSQL obtain almost linear speedups as we increase the parallelism. This reflects another nice property of our algorithm that it is easily parallelizable as it does its most of work on a per-key basis. However, PostgreSQL cannot benefit much from multiple cores.

Efficiency of the 1D alternative. To verify the efficiency of the 1D alternative, we ran Query 6 under all systems, and the results are shown in Figure 10. We note that using the range trees has worse performance than SparkSQL, which does the equi-join first and checks the comparisons later. On the other hand, the 1D alternative achieves a 15% improvement. Any-K has an even worse performance than SparkSQL, as its log factor is larger than Willard's.

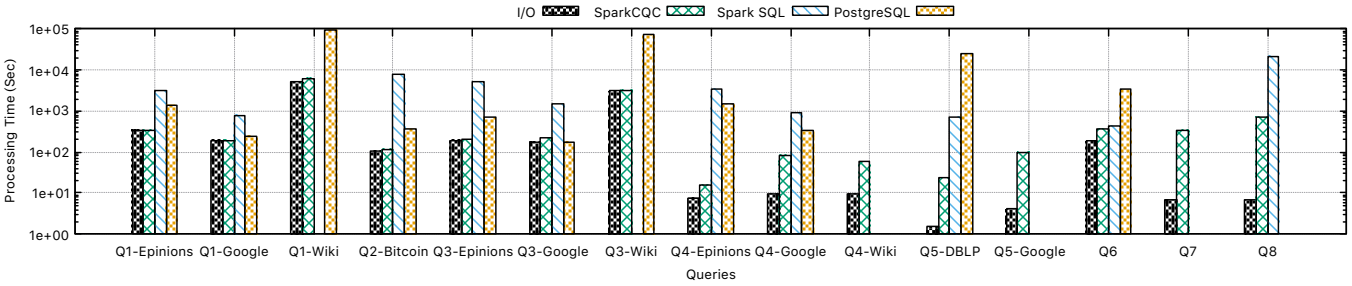
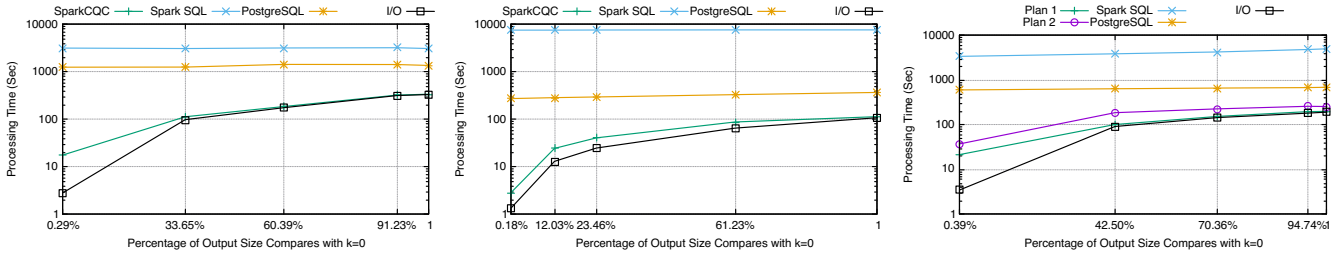
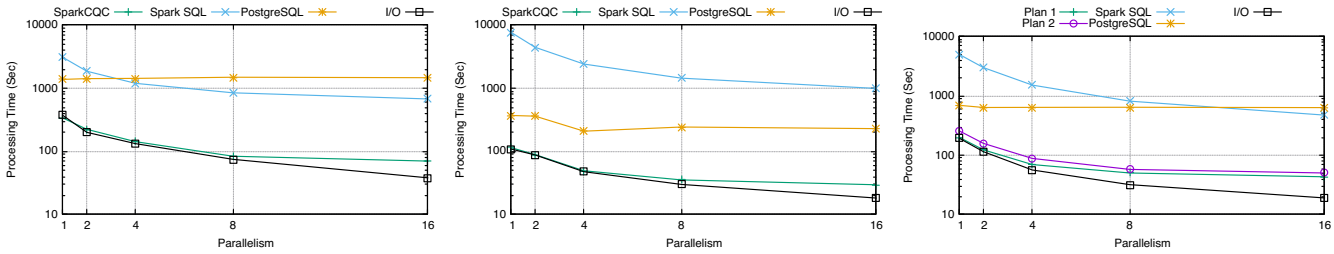


Figure 7: Processing times of SparkCQC, SparkSQL, and PostgreSQL.



(a) Q1-Epinions (b) Q2-Bitcoin (c) Q3-Epinions

Figure 8: Processing times under different selectivity.



(a) Q1-Epinions (b) Q2-Bitcoin (c) Q3-Epinions

Figure 9: Processing times under different parallelism.

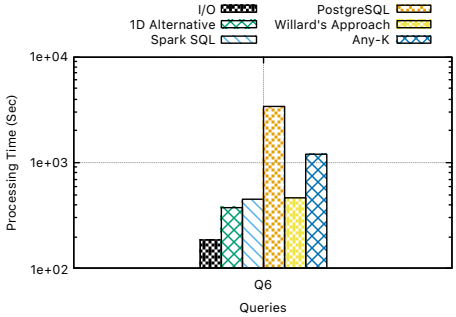


Figure 10: Processing times of different systems on Q6

Query optimization. Regardless of the reduction order, our algorithm always has running time $\tilde{O}(N+\text{OUT})$, but the hidden constant might differ for different plans. In the last set of experiments, we examine this difference with Q3, which has two reduction orders: Plan 1 reduces R_1 first while Plan 2 reduces R_3 first. In Figure 8(c) and 9(c), we see that Plan 1 is roughly better than Plan 2 by around a factor of 2 (or less). The reason is probably the following. In Plan 1, the first reduction is 1D, after which we obtain a query with two short comparisons, which is a 2D problem, and the last step is another 1D problem. In Plan 2, the first reduction is 2D, then we solve a 1D problem, but the last step is another 2D problem. Thus, our optimizer uses the simple heuristic that chooses the plan with the minimum "total dimensionality".

9 FUTURE WORK

There are interesting directions, both theoretical and practical, to extend this work. Theoretically, it is an intriguing question if our techniques can be applied to aggregation queries with comparisons. Currently, such queries can only have short comparisons [15]; long comparisons can only be dealt with using GHDs, leading to super-linear time. In practice, cost-based optimization can be added to our optimizer that chooses the optimal query plan (both reduction order and whether to use multi-dimensional range searching structures) based on actual data. It is also possible to merge SparkCQC into SparkSQL, as both translate SQL into RDD operations which are then executed by the same Spark core.

ACKNOWLEDGMENTS

This work has been supported by HKRGC under grants 16201318, 16201819, and 16205420.

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases: The Logical Level* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., USA.

[2] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1383–1394. https://doi.org/10.1145/2723372.2742797

[3] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. 2007. On Acyclic Conjunctive Queries and Constant Delay Enumeration *(CSL'07/EACSL'07)*. Springer-Verlag, 208–222.

[4] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. 2008. *Computational Geometry: Algorithms and Applications* (3rd ed. ed.). Springer-Verlag TELOS, Santa Clara, CA, USA.

[5] Nofar Carmeli and Markus Kröll. 2021. On the Enumeration Complexity of Unions of Conjunctive Queries. *ACM Transactions on Database Systems* 46, 2, Article 5 (May 2021), 41 pages. https://doi.org/10.1145/3450263

[6] Bernard Chazelle. 1988. A functional approach to data structures and its use in multidimensional searching. *SIAM J. Comput.* 17, 3 (1988), 427–462.

[7] Bernard Chazelle and Leonidas J. Guibas. 1986. Fractional cascading: II. Applications. *Algorithmica* 1 (1986), 163–191.

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

[9] Georg Gottlob, Martin Grohe, Nysret Musliu, Marko Samer, and Francesco Scarcello. 2005. Hypertree Decompositions: Structure, Algorithms, and Applications. In *WG*. Springer-Verlag, 1–15. https://doi.org/10.1007/11604686_1

[10] MH Graham. 1980. *On the universal relation.* Technical Report. University of Toronto. Computer Systems Research Group and Graham, MH.

[11] Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. 2020. General dynamic Yannakakis: conjunctive queries with theta joins under updates. *The VLDB Journal* 29 (2020), 619–653.

[12] Manas R. Joglekar, Rohan Puttagunta, and Christopher Ré. 2016. AJAR: Aggregations and Joins over Annotated Relations. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (San Francisco, California, USA) *(PODS '16)*. Association for Computing Machinery, New York, NY, USA, 91–106. https://doi.org/10.1145/2902251.2902293

[13] Mahmoud Abo Khamis, Hung Q. Ngo, Dan Olteanu, and Dan Suciu. 2019. Boolean Tensor Decomposition for Conjunctive Queries with Negation. In *22nd International Conference on Database Theory, ICDT 2019, March 26-28, 2019, Lisbon, Portugal (LIPIcs, Vol. 127)*, Pablo Barceló and Marco Calautti (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 21:1–21:19. https://doi.org/10.4230/LIPIcs.ICDT.2019.21

[14] 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 *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Chicago, Illinois, USA) *(PODS '17)*. Association for Computing Machinery, New York, NY, USA, 429–444. https://doi.org/10.1145/3034786.3056105

[15] Mahmoud Abo Khamis, Ryan R. Curtin, Benjamin Moseley, Hung Q. Ngo, Xuanlong Nguyen, Dan Olteanu, and Maximilian Schleich. 2020. Functional Aggregate Queries with Additive Inequalities. *ACM Transactions on Database Systems* 45, 4, Article 17 (dec 2020), 41 pages. https://doi.org/10.1145/3426865

[16] Paraschos Koutris, Tova Milo, Sudeepa Roy, and Dan Suciu. 2017. Answering Conjunctive Queries with Inequalities. *Theory of Computing Systems* 61 (2017), 2–30.

[17] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[18] Dániel Marx. 2013. Tractable Hypergraph Properties for Constraint Satisfaction and Conjunctive Queries. *J. ACM* 60, 6, Article 42 (nov 2013), 51 pages. https://doi.org/10.1145/2535926

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

[20] Mihai Patrascu. 2010. Towards Polynomial Lower Bounds for Dynamic Problems. In *Proceedings of the Forty-Second ACM Symposium on Theory of Computing* (Cambridge, Massachusetts, USA) *(STOC '10)*. Association for Computing Machinery, New York, NY, USA, 603–610. https://doi.org/10.1145/1806689.1806772

[21] Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. 2021. Beyond Equi-Joins: Ranking, Enumeration and Factorization. *Proc. International Conference on Very Large Data Bases* 14, 11 (jul 2021), 2599–2612. https://doi.org/10.14778/3476249.3476306

[22] Ron van der Meyden. 1997. The complexity of querying indefinite data about linearly ordered domains. *J. Comput. System Sci.* 54, 1 (1997), 113–135.

[23] Dan E Willard. 2002. An Algorithm for Handling Many Relational Calculus Queries Efficiently. *J. Comput. System Sci.* 65, 2 (Sept. 2002), 295–331.

[24] Mihalis Yannakakis. 1981. Algorithms for Acyclic Database Schemes. In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7* (Cannes, France) *(VLDB '81)*. VLDB Endowment, 82–94.

[25] Clement Tak Yu and Meral Z Ozsoyoglu. 1979. An algorithm for tree-query membership of a distributed query. In *COMPSAC 79. Proceedings. Computer Software and The IEEE Computer Society's Third International Applications Conference, 1979.* IEEE, 306–312. https://doi.org/10.1109/CMPSAC.1979.762509

[26] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for in-Memory Cluster Computing *(NSDI'12)*. USENIX Association, USA, 2.