Conjunctive Queries with Comparisons

[Extended Abstract]

Qichen Wang
Hong Kong Baptist University
Hong Kong, China
qcwang@hkbu.edu.hk

Ke Yi
HKUST
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 patterns and analytical queries.

1. INTRODUCTION

The asymptotically optimal running time for evaluating a query is $\tilde{O}(N + \text{OUT})$, where N is the input size, OUT is the output size, and the $\tilde{O}$ notation suppresses a $\log^{O(1)} N$ factor. 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 [20] tells us that linear time can be achieved for α-acyclic *conjunctive queries (CQs)*, and recent negative results [2, 16] 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, which involves two types of predicates: type-1, involving attributes from one relation, and type-2, spanning 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. However, type-2 predicates have received less attention. The naive approach for handling this type of predicate is to compute the join first and then filter the results with the predicate. A common query optimization technique is *predicate pushdown*, where the predicate is pushed to 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, with $\leq, <, \geq, >$ as comparisons, and $\neq$ as inequality. Such queries are related to OLAP queries, spatiotemporal queries, and machine learning over relational data. The following gives an example of a temporal query.

Example 1.1. *Consider a toy database that stores a collaboration network using the scheme $R(P_1, P_2, T)$. Any tuple $t = (p_1, p_2, t) \in R$ indicates that the person p_1 has collaboration with p_2 starting from time t. The following query with a type-2 predicate, written in a rule-based form, will try to join three relations R_1, R_2, R_3, which all follow the schema R. The query will find all (p_1, p_2, p_3, p_4) combinations, such that p_2 has collaboration with both p_1 and p_3, where first with p_1 and then with p_3; meanwhile, p_3 has collaboration with p_4:*

$$R_1(p_1, p_2, t_1), R_2(p_2, p_3, t_2), R_3(p_3, p_4, t_3), t_1 \leq t_2.$$

This query (the CQ part) is α-acyclic. For the query plan $(R_1 \bowtie R_2) \bowtie R_3$, the predicate $t_1 \leq t_2$ 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 $t_1 \leq t_2$. In this case, $\text{OUT} = 0$ but the intermediate join size $|R_1 \bowtie R_2|$ can be as large as $\Omega(N^2)$.

A simple idea [12, 19] to reduce the time to (near) linear is to first push down the predicate, and then compute the sub-query

$$R_1(p_1, p_2, t_1), R_2(p_2, p_3, t_2), t_1 \leq t_2$$

without computing the join $R_1 \bowtie R_2$: Group the tuples in R_1 and R_2 by p_2. For each group, sort the t_1 values in ascending order. Then for each t_2, scan the sorted list until meeting some $t_1 > t_2$. 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 as in [20]. □

The predicate $t_1 \leq t_2$, as we define more formally in Section 4, is a *short* comparison. The following example features a *long* one:

EXAMPLE 1.2. *Consider the following query with predicate $x_1 \leq x_4$:*

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

Note that a long comparison like $x_1 \leq x_4$ cannot be pushed down. By decomposing the query to multiple parts, each plugged into an appropriately chosen generalized hypertree decomposition (GHD) [7] and combined with the idea from Example 1.1, Abo Khamis et al. [12] 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 [20], 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. Such an algorithm can also handle non-full CQCs (i.e., join-selection-projection queries). Due to the space constraint, the details are left in [18]. For queries outside this class, we show in Section 6 how to combine our techniques with the GHD framework to obtain improved running times over [12].

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, Abo Khamis et al. [12] also try to reduce the length, but only use the GHD framework to group 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 7) show that our algorithm offers an order-of-magnitude improvement over SparkSQL, especially for queries with highly selective type-2 predicates.

2. RELATED WORK

Recently, evaluating type-2 predicates over a CQ has regained interest, with several papers [9,12,17] addressing the issue. In particular, Abo Khamis et al. [12] 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 [19], who also generalized it to α-acyclic CQs with multiple comparisons, but all comparisons must be short. Unaware of his paper, the recent works [9, 12, 17] rediscovered Willard's idea and then extended it in various ways. Idris et al. [9] study the dynamic version of the problem; in the static setting, their algorithm is essentially the same as Willard's. Tziavelis et al. [17] 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, but the logarithmic factor is larger than Willard's. All these papers [9,17,19] only consider short comparisons. Abo Khamis et al. [12] 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 [12] are provided in Section 6). They also generalize their framework to handle aggregation queries, which are important for machine learning tasks.

Koutris et al. [13] and Abo Khamis et al. [10] 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 $[n] = \{1, \ldots, n\}$. Let $\boldsymbol{R}$ be a relational database. A *conjunctive query with comparison (CQC)* has the form

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

where $R_1, R_2, \ldots$ are relations in $\boldsymbol{R}$, and $\bar{x}_1, \ldots, \bar{x}_n$ 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.

EXAMPLE 3.1. *The query*

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

fits the definition of a CQC by setting $a_1 = 1, b_1 = 3, f_1(\bar{x}_1) = x_1 - x_2, g_1(\bar{x}_3) = x_3 x_4 + 2$;

$a_2 = 2, b_2 = 4, f_2(\bar{x}_2) = \min\{2x_2, x_7\}, g_2(\bar{x}_4) = x_6$;
$a_3 = 3, b_3 = 5, f_3(\bar{x}_3) = x_2, g_3(\bar{x}_5) = x_8$. □

Note that for 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 6 (cf. Example 6.2).

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. An inequality $f_j(\bar{x}_{a_j}) \ne g_j(\bar{x}_{b_j})$ can be written as $f_j(\bar{x}_{a_j}) > g_j(\bar{x}_{b_j}) \vee f_j(\bar{x}_{a_j}) < g_j(\bar{x}_{b_j})$.

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 $f_j(t(\bar{x}_{a_j}))$, $g_j(t(\bar{x}_{b_j}))$.

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 $w(p) \in \mathbb{S}$. The problem has two versions. 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 solve both versions [3, 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 [4] 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 the input size $N = \sum_i |R_i(\bar{x}_i)|$ and output size $\text{OUT} = |q(\boldsymbol{R})|$ 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 [5], 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, 8, 21] 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.

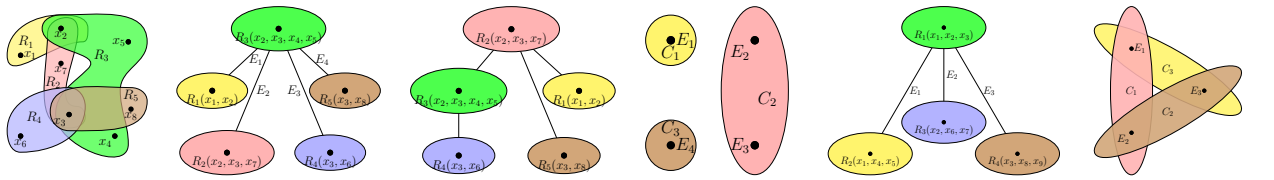
For a CQC q, we consider a second hypergraph, called its *comparison hypergraph*, which is defined after fixing a join tree T of q. After fixing T, for each comparison C_j, we set its two incident relations R_{a_j}, R_{b_j} such that among all valid (a_j, b_j) pairs, $P_T(a_j, b_j)$ is the shortest. 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*.

Then the comparison hypergraph of q induced by a given join tree T, denoted as $\mathcal{C}(q, T)$, is defined as follows. The vertices of $\mathcal{C}(q, T)$ correspond to the edges of T, while its hyperedges correspond to the comparisons in q. More precisely, a vertex in $\mathcal{C}(q, T)$, namely an edge (u, v) of T, belongs to a hyperedge of $\mathcal{C}(q, T)$, 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 $\mathcal{C}(q, T)$, and a *self-comparison*, i.e., one where $a_j = b_j$, becomes an empty hyperedge. Meanwhile, some vertices in $\mathcal{C}(q, T)$ 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 $\mathcal{C}(q, T)$ 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 $\mathcal{C}(q, T)$ are the edges of T, so the Berge-acyclicity of $\mathcal{C}(q, T)$ 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.



(a) Relational hypergraph (b) Join tree (c) Another join tree (d) Comparison hypergraph of 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.

EXAMPLE 4.1. *The query*

$$ans(x_1, x_2, x_3) \leftarrow R_1(x_1, x_2, x_3), R_2(x_1, x_4, x_5), R_3(x_2, x_6, x_7), R_4(x_3, x_8, x_9), C_1 : x_4 \leq x_6, \quad C_2 : x_7 \leq x_8, \quad C_3 : x_9 \leq x_5$$

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 its 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 $\mathcal{C}(q, T)$. Each edge (u, v) in T corresponds to a vertex in $\mathcal{C}(q, T)$, and we use $d(u, v)$ to denote its *degree* in $\mathcal{C}(q, T)$, i.e., the number of hyperedges of $\mathcal{C}(q, T)$ that contain (u, v). The degree of $\mathcal{C}(q, T)$ is the maximum degree of all $(u, v) \in T$, and we define the degree of q as the minimum degree of $\mathcal{C}(q, T)$ 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 $\mathcal{C}(q, T)$. On the other hand, the join tree in Figure 1(c) would lead to a $\mathcal{C}(q, T)$ 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 to 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 [20]. This algorithm has been extended to perform *constant delay enumeration (CDE)* [2]. 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 $x_1 \leq x_4$. If these tuples are not skipped during 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.* □

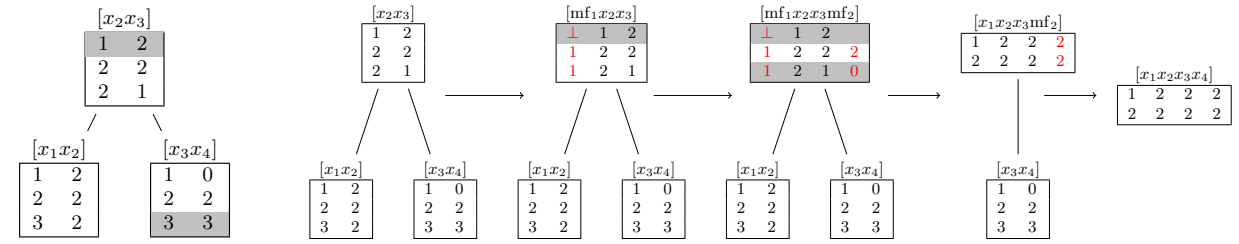
5. FULL ACYCLIC CQCs

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 $\tilde{O}(1)$.



(a) Standard semijoin reduction

(b) New Approach

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

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.*

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 semi-join 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 [2], but we rephrase the arguments under our framework, i.e., it satisfies the three properties stated at the beginning of Section 5. Detailed analysis can be found in [18].

5.3 One Incident Comparison

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

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

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 $C' : \mathrm{mf} \le g(\bar{x}_b)$.

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.

For the symmetric case where C is $f(\bar{x}_a) \le g(var(R))$ 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. Such a reduction can be written in SQL as follows:*

```
SELECT x2, x3, min(x1) as mf1
FROM R1, R2
WHERE R1.x2 = R2.x2
```

Suppose we reduce R_3 next (we could also reduce R_2, which is now reducible). This appends mf_2 *to R_2. The reduction can be written in SQL as follows, where R'_2 represents the relation R_2 after applying the first reduction:*

```
SELECT x2, x3, mf1, max(x4) as mf2
FROM R2', R3
WHERE R2'.x3 = R3.x3
```

After this step, the comparison becomes a self-comparison $\mathrm{mf}_1 \le \mathrm{mf}_2$. *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 $x_1 \le 2$. 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. Such procedure can be considered as evaluating the following SQL query (assume R''_2 to be the relation after applying condition $\mathrm{mf}_1 \le \mathrm{mf}_2$*):*

```
SELECT x1, x2, x3
FROM R2'', R1
WHERE R2''.x2 = R1.x2 and R1.x1 ≤ R2.mf2
```

Then, for each partial join result from $R_1 \bowtie R_2$, we find all join tuples in R_3 that satisfy $x_1 \le x_4$. Similarly, as all tuples in R_3 are grouped by x_3 and sorted in descending order of x_4, such tuples can be retrieved with constant delay. □

5.4 Two or More Incident Comparisons

Now we consider the general case. Let R be a *reducible* relation with $d \ge 2$ 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 $f_j(var(R)) \le g_j(\bar{x}_{b_j})$, where $b_j \in [n]$ for each $j \in [d]$. If any C_j 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 sim-

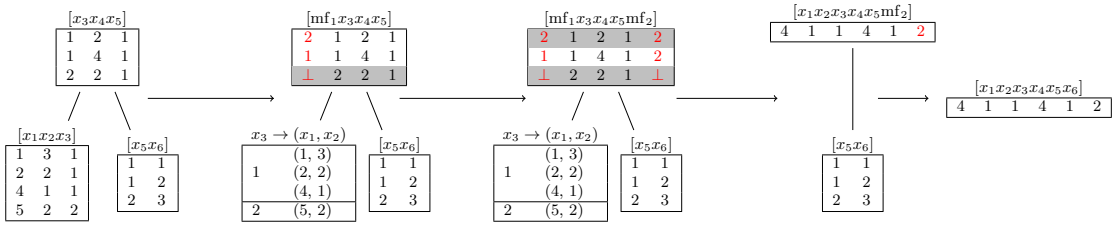


Figure 4: A running example for the query in Example 5.4

ilar to the one-incident-comparison case. Define

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

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 $C'_1 : \mathrm{mf} \le g_1(\bar{x}_{b_1})$.

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

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. The reduction can be written in SQL as:

```
SELECT x3, x4, x5, min(x2) as mf1
FROM R1, R2
WHERE R2.x3 = R1.x3 and R1.x1 ≤ R2.x4
```

In order to solve the reduction query efficiently, 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 $\mathrm{mf}_1 = \perp$ 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 $\mathrm{mf}_1 < x_6$. 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 $\mathrm{mf}_1 < x_6$ into a self-comparison $\mathrm{mf}_1 < \mathrm{mf}_2$, 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 $x_1 \le x_4$. The enumeration procedure can be written as:

```
SELECT x1, x2, x3, x4, x5
FROM R1, R2''
WHERE R2''.x3 = R1.x3 and R1.x1 ≤ R2''.x4 and
      R1.x2 ≤ R2''.mf2
```

and we can further expedite the execution by 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, 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 $d \ge 2$, 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 $O(N \log^{d_q - \mathbb{I}(d_q \ge 2)} N)$ preprocessing time, where $\mathbb{I}(\cdot)$ is the indicator function.*

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

6. GENERALIZED HYPERTREE DECOMPOSITIONS WITH CQCS

Generalized hypertree decompositions (GHDs) [7] provide a powerful framework for dealing with general CQs [11] and CQCs [12]. 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, Abo Khamis et al. [12] show that a CQC q can be enumerated with $\tilde{O}(1)$ delay after $\tilde{O}(N^{\mathsf{width}(q)})$ preprocessing time, where

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

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 $\mathcal{G}(q) \subseteq \mathcal{G}^*(q)$, $\mathsf{width}^*(q) \leq \mathsf{width}(q)$ 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 6.1. *Consider the following CQC:*

$$R_1(x_1, x_2), R_2(x_2, x_3), R_3(x_3, x_4), R_4(x_4, x_5),$$
$$C_1 : x_1 \leq x_4, C_2 : x_1 \geq x_5.$$

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 [12]. 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 6.2. *Consider the following CQC, which finds all "dumbbells" in a graph, whose edges are stored in a relation R We impose a comparison involving the weights associated with the edges of the two triangles that make up the dumbbell.*

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

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 [15], 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 [12]. □

To keep the presentation accessible, we have only stated the general result where a single GHD is used. It has been shown [11, 12, 14] that the width can be further reduced by using multiple GHDs. Our algorithm offers improvements in this case as well.

EXAMPLE 6.3. *Revisit the query in Example 6.1. 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 $\deg_{R_2}(t, x_2) \geq \sqrt{N}$ 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 6.1. But now all tuples in R_2 are light, so we have $|R_1 \bowtie R_2| \leq N^{1.5}$. 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. 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 [12] because neither GHD used above is allowed in [12]. 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 6.1, 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 6.4. *The following query is a non-full version of Example 6.1:*

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

This query cannot be handled by our algorithm directly 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 [12] is still $\tilde{O}(N^{1.5})$ while using two GHDs. □

7. EXPERIMENTS

7.1 Experimental Setup

Prototype implementation.

We have implemented our algorithms in a system prototype on top of Spark, 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

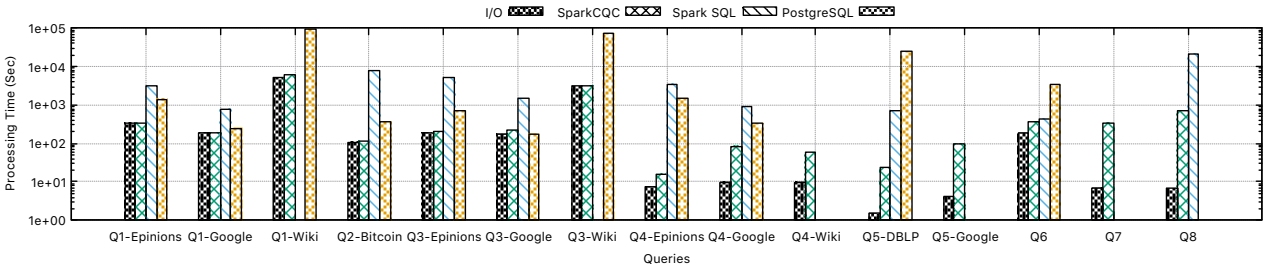
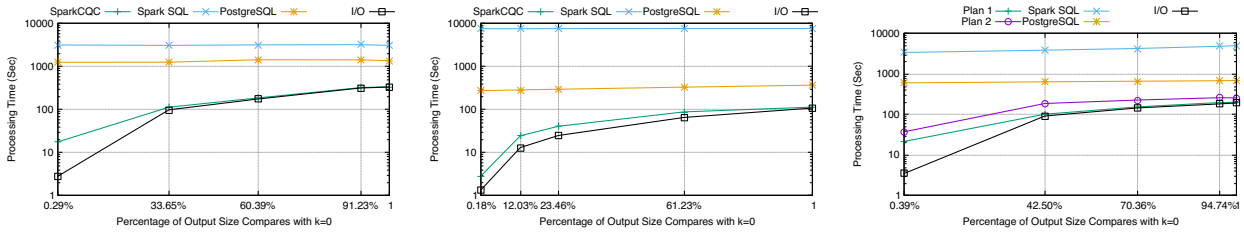


Figure 5: Processing times of SparkCQC, SparkSQL, and PostgreSQL.



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

Figure 6: Processing times under different selectivity.

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.

Query processing engines compared.

We compare our algorithms with SparkSQL 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 [17] (called "Any-K"). Their algorithm only supports full CQCs with short comparisons, so we can only compare it on Query 6.

7.2 Datasets and Queries

We tested 5 graph pattern queries and 3 analytical queries. The characteristics and the formal definition of each query can be found in [18].

For graph pattern queries, we use some real graphs from SNAP (Stanford Network Analysis Project). 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.

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

7.3 Experimental Results

Running time comparison.

Figure 5 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.

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 $f(\bar{x}) + k \leq g(\bar{y})$, which leads to various selectivities by controlling the value of k. The experiment results are shown in Figure 6, 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.

8. ACKNOWLEDGMENTS

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

9. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases: The Logical Level.* Addison-Wesley

Longman Publishing Co., Inc., USA, 1st edition, 1995.

[2] G. Bagan, A. Durand, and E. Grandjean. On acyclic conjunctive queries and constant delay enumeration. CSL/EACSL, page 208–222. Springer-Verlag, 2007.

[3] B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, 1988.

[4] B. Chazelle and L. J. Guibas. Fractional cascading: II. Applications. *Algorithmica*, 1:163–191, 1986.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[6] M. de Berg, O. Cheong, M. J. van Kreveld, and M. H. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd ed. edition, 2008.

[7] G. Gottlob, M. Grohe, N. Musliu, M. Samer, and F. Scarcello. Hypertree decompositions: Structure, algorithms, and applications. In *WG*, page 1–15. Springer-Verlag, 2005.

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

[9] M. Idris, M. Ugarte, S. Vansummeren, H. Voigt, and W. Lehner. General dynamic yannakakis: conjunctive queries with theta joins under updates. *The VLDB Journal*, 29:619–653, 2020.

[10] M. Khamis, H. Ngo, D. Olteanu, and D. Suciu. Boolean tensor decomposition for conjunctive queries with negation. In *ICDT*, volume 127 of *LIPIcs*, pages 21:1–21:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[11] M. Khamis, H. Q. Ngo, and D. Suciu. What do shannon-type inequalities, submodular width, and disjunctive datalog have to do with one another? In *PODS*, page 429–444. ACM, 2017.

[12] M. A. Khamis, R. R. Curtin, B. Moseley, H. Q. Ngo, X. Nguyen, D. Olteanu, and M. Schleich. Functional aggregate queries with additive inequalities. *ACM Transactions on Database Systems*, 45(4), dec 2020.

[13] P. Koutris, T. Milo, S. Roy, and D. Suciu. Answering conjunctive queries with inequalities. *Theory of Computing Systems*, 61:2–30, 2017.

[14] D. Marx. Tractable hypergraph properties for constraint satisfaction and conjunctive queries. *Journal of the ACM*, 60(6), nov 2013.

[15] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms: [extended abstract]. In *PODS*, page 37–48. ACM, 2012.

[16] M. Patrascu. Towards polynomial lower bounds for dynamic problems. In *STOC*, page 603–610. ACM, 2010.

[17] N. Tziavelis, W. Gatterbauer, and M. Riedewald. Beyond equi-joins: Ranking, enumeration and factorization. In *VLDB*, volume 14, page 2599–2612. VLDB Endowment, 2021.

[18] Q. Wang and K. Yi. Conjunctive queries with comparisons. In *SIGMOD*, pages 108–121. ACM, 2022.

[19] D. E. Willard. An Algorithm for Handling Many Relational Calculus Queries Efficiently. *JCSS*, 65(2):295–331, Sept. 2002.

[20] M. Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, page 82–94. VLDB Endowment, 1981.

[21] C. T. Yu and M. Z. Ozsoyoglu. An algorithm for tree-query membership of a distributed query. In *COMPSAC*, pages 306–312. IEEE, 1979.