

# Incrementally Computing Ordered Answers of Acyclic Conjunctive Queries<sup>\*</sup>

Benny Kimelfeld and Yehoshua Sagiv  
{bennyk,sagiv}@cs.huji.ac.il

The Selim and Rachel Benin School of Engineering and Computer Science  
The Hebrew University of Jerusalem  
Edmond J. Safra Campus  
Jerusalem 91904, Israel

**Abstract.** Evaluations of SQL queries with the `ORDER BY` clause is considered. The naive approach of first computing the result and then sorting the tuples is not suitable for Web applications, since the result could be very large while users expect to get quickly the top- $k$  tuples. Tractability, in this case, amounts to enumerating answers in sorted order with *polynomial delay*, under *query-and-data* complexity. It is proved that an efficient algorithm for finding the top-ranked tuple of a conjunctive query is a sufficient (and not just necessary) condition for tractability. Several classes of orders are shown to have this property when queries are acyclic.

## 1 Introduction

Computing queries incrementally has become an increasingly important issue; for example, in Web applications users want to see the first few pages of results as quickly as possible. This task is more challenging if a ranking function is involved and users are interested in the top- $k$  answers. One approach [3, 4] is to extend SQL with the `STOP AFTER` operator that limits the generated tuples to the top- $k$ . An efficient processing of this operator is done by enriching query plans with counters and sorters [3] and range partitioning [4]. Other papers proposed algorithms for computing the top- $k$  results of ranked joins, including the  $J^*$  algorithm [14], the rank-join algorithm [11] and the top- $k$ -oriented variant of the hash-join algorithm [9].

The goal of this paper is to investigate the complexity of computing answers of queries in ranked order. To discern between tractable and intractable cases, one has to measure the running time as a function of the combined size of the input and the output, using *query-and-data complexity* (that is, the size of the query is unbounded). Since queries may have exponentially many answers, an algorithm is efficient if it computes all the answers in *polynomial total time*, i.e., the runtime is bounded by a polynomial in the combined size of the query, the data and the result. Computing efficiently the top- $k$  answers requires a stronger notion, namely, the runtime should be polynomial in the size of the input and

---

<sup>\*</sup> This research was supported by The Israel Science Foundation (Grant 893/05).

the value  $k$ . An even stronger notion is that of *polynomial delay* [12], that is, the elapsed time between generating two successive tuples is polynomial only in the size of the input (i.e., the query and data).

The algorithms mentioned above are not efficient from a theoretical point of view, since the delay in producing the top answer (or any answer thereafter) could be exponential. We show that for a large class of queries, answers can be computed in ranked order with polynomial delay. We believe that this result could lead to better algorithms for computing the top- $k$  answers than those proposed thus far.

Two factors determine whether the top- $k$  answers can be computed efficiently. First, the *non-emptiness* problem must be tractable. In other words, if it is hard to determine whether the result of a query is nonempty, then it is impossible to enumerate the answers with polynomial delay (regardless of which order is used). Second, for some (rather simple) ranking functions, it is hard to enumerate the answers in ranked order even if non-emptiness is easy to check.

In this paper, we consider the class of *acyclic conjunctive queries*, which is one of the largest classes with an efficient algorithm for checking non-emptiness. We first show that, with respect to all conjunctive queries, the important property of ranking functions is the ability to find the top answer efficiently (i.e., in polynomial time in the size of the the query and the data). Clearly, this is a necessary condition for enumerating in ranked order with polynomial delay. In the case of general conjunctive queries, it is (rather surprisingly) also a sufficient condition. We identify several types of ranking functions that have this property with respect to acyclic conjunctive queries. Our results hold even if projections are used and duplicates should be removed, and even if the ranking depends also on attributes that are not included in the final projection. Note that a naive handling of duplicate elimination (i.e., removing them at the end of the computation) cannot yield an enumeration with polynomial delay or even in polynomial total time.

## 2 Motivation

For motivating our work, consider the database scheme of Figure 1 and the SQL query of Figure 2. This query finds apartments for sale and sorts the result according to a linear combination of the average temperature, the price and the distance from London. Note that the result includes only some of the attributes that determine the order; in other words, there are attributes that appear in the `ORDER BY` clause but not in the `SELECT` clause. But this fact does not affect the complexity of evaluating the given query, since duplicates are not removed (i.e., `DISTINCT` is not specified in the `SELECT` clause). That is, the difference between applying projection as soon as possible and applying it in the last step is negligible. If, however, duplicates have to be removed and projection is performed only in the last step (followed by duplicate elimination), then intermediate results could be much larger than the final one, leading to a difference of an exponential factor in the runtime.

<ul style="list-style-type: none"> <li>- <i>Apartments</i> (id, city, street, price, bedrooms)</li> <li>- <i>Climate</i> (city, avgTemp)</li> <li>- <i>Distances</i> (fromCity, toCity, distance)</li> </ul>
--

**Fig. 1.** A database scheme

Two criteria determine whether queries can be evaluated without generating intermediate results that are too large. First, testing *non-emptiness* of queries (regardless of the order) must be a tractable problem. Since not all conjunctive queries satisfy this requirement [5], we state our actual tractability results for *acyclic conjunctive queries* [1, 2, 15] that have this property. Second, there must be an efficient algorithm for finding the top-ranked (i.e., first) tuple of the result, according to the specified order. Consider, for example, the following query.

```

SELECT *
FROM  $R_1, \dots, R_n$ 
ORDER BY ABS( $R_1.A_1 + \dots + R_n.A_n - K$ )

```

$R_1, \dots, R_n$  are names of relations that have the attributes  $A_1, \dots, A_n$ , respectively, and  $K$  is a number. **ABS** is the absolute-value operator. Testing non-emptiness of this query is easy, since it is merely a Cartesian product. It is intractable, however, to compute the top-ranked tuple.<sup>1</sup>

Our goal is to find sufficient conditions for efficiently enumerating all tuples of the result in sorted order. Informally, efficiency means that the delays between generating successive tuples are small. We show that for general conjunctive queries, tractability of finding the top-ranked tuple is a sufficient (and not just necessary) condition for efficiently enumerating in sorted order. This result holds even if duplicates are eliminated and attributes of the **ORDER BY** clause are not required to appear in the **SELECT** clause. The proof is by means of an algorithm that reduces the problem of enumerating in sorted order to the problem of finding the top-ranked tuple with respect to the given order. (Technically, we must find the top-ranked tuple with respect to the query that is obtained from the original one by adding all attributes to the **SELECT** clause.)

Next, we consider acyclic conjunctive queries and show that for two classes of orders, the top-ranked answer can be found efficiently. The first class consists

<sup>1</sup> This can be easily proved by a reduction from the *subset-sum* problem.

<pre> SELECT A.city, A.bedrooms, A.price FROM Apartments A, Climate C, Distances D WHERE A.city = C.city = D.fromCity AND D.toCity='London' ORDER BY C.avgTemp*10 - D.distance - A.price/1000 DESC </pre>
---

**Fig. 2.** An SQL query

of *monotonic orders* that have the following property. Consider a tuple  $t$  and all subsets  $S$  of the attributes that appear in the ORDER BY clause. Informally, if for some subset  $S$ , replacing the values of  $S$  in  $t$  with new ones yields a tuple that has at least the same rank as  $t$ , then this replacement can only improve every tuple of the result that is equal to  $t$  on  $S$  (whenever the modified tuple is also in the result). As an example, the following orders are monotonic:

- ORDER BY  $C_1(R_{i_1}.A_{i_1}) + \dots + C_k(R_{i_k}.A_{i_k})$  [DESC] (**Linear combination**)
- ORDER BY  $R_{i_1}.A_{i_1}$  [DESC],  $\dots$ ,  $R_{i_k}.A_{i_k}$  [DESC] (**Lexicographic**)

The second class of orders is called *c-determined*, where  $c$  is a fixed positive integer (i.e.,  $c$  does not depend on the input). Informally, in a  $c$ -determined order, the position of a tuple in the order is determined by a set of  $c$  or fewer values. For example, the following two orders are 1-determined and 3-determined, respectively. Note that in the first order,  $k$  is unbounded (i.e., not assumed to be fixed).

- ORDER BY MAX( $C_1 R_{i_1}.A_{i_1}, \dots, C_k R_{i_k}.A_{i_k}$ ) DESC
- ORDER BY  $R_1.A_1 * R_2.A_2 + R_3.A_3$

### 3 Formal Setting

Our goal is to identify queries and orders that have efficient algorithms for producing answers in the specified ordering. We consider *acyclic conjunctive queries*, since they satisfy the necessary requirement, namely, answers can be generated efficiently in an arbitrary order. This section defines the main concepts.

#### 3.1 Databases and Conjunctive Queries

We assume that **Dom** is a countable domain of atomic values. A *relation scheme*  $R$  has an associated *arity*, denoted by  $\text{arity}(R)$ . A *relation instance* (or simply, a *relation*) over a relation scheme  $R$  is a finite subset of  $\mathbf{Dom}^{\text{arity}(R)}$ . The elements of a relation are *tuples*. A *database scheme*  $S$  is a set of relation schemes. A *database instance* (or simply, a *database*)  $D$  over a database scheme  $S$  consists of a relation instance, denoted by  $D[R]$ , for each relation scheme  $R$  of  $S$ . We write  $D' \subseteq D$  if  $D'$  is obtained from  $D$  by deleting some tuples.

A *conjunctive query* (abbr. CQ)  $Q$  over a database scheme  $S$  has the form

$$Q(\mathbf{u}): -R_1(\mathbf{u}_1), R_2(\mathbf{u}_2), \dots, R_k(\mathbf{u}_k)$$

and it satisfies the following: (1)  $R_1, \dots, R_n$  are (not necessarily distinct) relation schemes of  $S$ ; (2)  $\mathbf{u}$  is a list of *variables*; (3) Each  $\mathbf{u}_i$  is a list of *terms* that has a length of  $\text{arity}(R_i)$ , where a term is either a constant of **Dom** or a variable; and (4) Each variable in  $\mathbf{u}$  appears in some  $\mathbf{u}_i$ . We say that each  $R_i(\mathbf{u}_i)$  is a *conjunct* of  $Q$  and  $Q(\mathbf{u})$  is the *head* of  $Q$ . The set of variables appearing in  $Q$  is denoted by  $\text{var}(Q)$ . When we consider a CQ  $Q$  and a database  $D$ , we implicitly assume that  $Q$  and  $D$  are defined over the same database scheme.

Consider a conjunctive query  $Q$  and a database  $D$ . An *assignment* for  $var(Q)$  is a mapping  $\varphi : var(Q) \rightarrow \mathbf{Dom}$ . We use  $\varphi \mathbf{u}_i$  to denote the tuple that is obtained from  $\mathbf{u}_i$  by replacing every variable  $X$  with  $\varphi(X)$  (and leaving constants unchanged). A *homomorphism from  $Q$  to  $D$*  is an assignment  $\varphi$  for  $var(Q)$ , such that  $\varphi \mathbf{u}_i \in D[R_i]$  for each conjunct  $R_i(\mathbf{u}_i)$  of  $Q$ . We use  $Hom(Q, D)$  to denote the set of all homomorphisms from  $Q$  to  $D$ . An *answer* is a tuple  $\varphi \mathbf{u}$ , where  $\varphi$  is a homomorphism of  $Hom(Q, D)$ . The *result* of applying  $Q$  to  $D$ , denoted by  $Q(D)$ , is the set of all the answers, i.e.,  $Q(D) = \{\varphi \mathbf{u} \mid \varphi \in Hom(Q, D)\}$ .

Acyclic conjunctive queries [1, 2] (abbr. ACQs) are conjunctive queries that have join trees. Formally, we say that a conjunctive query  $Q$  is *acyclic* if there exists a tree  $T$  (called a *join tree*), such the nodes of  $T$  are the conjuncts of  $Q$  and the following holds. For every variable  $X \in var(Q)$ , the subgraph of  $T$  that is induced by the conjuncts containing  $X$  is connected (i.e., it is a subtree of  $T$ ). It has been shown [15] that acyclic conjunctive queries can be computed efficiently, i.e., in time that is polynomial in the combined size of the query, the database and the result, while for general conjunctive queries, it is NP-complete just to determine whether the result is nonempty [5].

### 3.2 Orders over Query Answers

Consider a conjunctive query  $Q$  and a database  $D$ . Our goal is to enumerate all the tuples of  $Q(D)$  according to an order that is specified, for example, in the ORDER BY clause of an SQL query. Recall that the ORDER BY clause may include attributes that do not necessarily appear in the SELECT clause. In other words, we cannot assume that the specified order is defined merely over the tuples of the result, since these tuples are obtained after projecting out attributes that may determine the position of a tuple in the desired ordering. Thus, an order over the answers to a query should be defined in terms of a more general order on the *ways* to derive these answers, i.e., homomorphisms. As an example, consider the SQL query of Figure 2. The corresponding conjunctive query is

$$Q(C, B, P) :- \text{Apartments}(I, C, S, P, B), \text{Climate}(C, T), \\ \text{Distances}(C, \text{'London'}, D)$$

and the ranking function is  $10\varphi(T) - \varphi(D) - \varphi(P)/1000$ . The values  $\varphi(T)$  and  $\varphi(D)$ , however, do not appear in the result. Things become more complicated if the DISTINCT operator is used in order to eliminate duplicates. In this case, the position of a tuple is determined by the first appearance that this tuple would have, had the DISTINCT operator not been used.

Formally, we assume that there is an underlying *order*  $\succeq$  over homomorphisms that is defined for a given CQ  $Q$  and a database  $D$ . The order  $\succeq$  is a binary relation over  $Hom(Q, D)$  that is reflexive, transitive and total (i.e., every two homomorphisms are comparable).

Orders on homomorphisms determine orders on answers as follows. Consider a query with the head  $Q(\mathbf{u})$  and a database  $D$ . Let  $\varphi_1, \varphi_2 \in Hom(Q, D)$ . If  $\varphi_1 \succeq \varphi_2$  holds, then it means that  $\varphi_1$  generates a “better” answer, i.e.,  $\varphi_1 \mathbf{u}$

should precede  $\varphi_2 \mathbf{u}$ . Formally,  $\succeq$  implies an order  $\succeq_{Q,D}$  over the tuples of  $Q(D)$  that is defined as follow. Two tuples  $t_1, t_2 \in Q(D)$  satisfy  $t_1 \succeq_{Q,D} t_2$  if for every homomorphism  $\varphi_2 \in \text{Hom}(Q, D)$  satisfying  $\varphi_2 \mathbf{u} = t_2$ , there exists a homomorphism  $\varphi_1 \in \text{Hom}(Q, D)$  satisfying  $\varphi_1 \mathbf{u} = t_1$ , such that  $\varphi_1 \succeq \varphi_2$ . In other words,  $t_1 \succeq_{Q,D} t_2$  if the maximal homomorphisms  $\varphi_1$  and  $\varphi_2$  that produce  $t_1$  and  $t_2$ , respectively, satisfy  $\varphi_1 \succeq \varphi_2$ . Thus, the goal is to enumerate the answers of  $Q(D)$  in decreasing order, i.e., if  $t_1 \succeq_{Q,D} t_2$ , then  $t_1$  is printed before  $t_2$ .

### 3.3 Measuring Efficiency

We want to enumerate all the answers of  $Q(D)$  in the order  $\succeq_{Q,D}$ . A related problem is that of finding a maximal (i.e., top-ranked) homomorphism of  $\text{Hom}(Q, D)$ . An algorithm for this problem is deemed efficient if its runtime is polynomial (in the size of  $Q$  and  $D$ ). Note that we assume that  $\varphi' \succeq \varphi$  can be tested efficiently.

For the problem of enumerating all the answers of  $\text{Hom}(Q, D)$ , polynomial time is not a suitable measure of efficiency, since the output size could be much larger (e.g., exponentially larger) than the input size. For this type of problems, there are several notions of efficiency [12]. The strongest one among them is enumeration with *polynomial delay*, that is, after the algorithm prints the  $(i-1)$ st answer, it generates the next ( $i$ th) answer in time that is polynomial in the input size. Polynomial delay implies an efficient evaluation of the top- $k$  answers, since we can stop the execution after printing the first  $k$  answers and the runtime is only linear in  $k$  (and polynomial in the input).

By definition, the result of a conjunctive query is a set whereas an SQL query may produce duplicates (unless the `DISTINCT` operator is used). It is sufficient, however, to develop an efficient algorithm for enumerating tuples of the result assuming that duplicates are eliminated. If duplicates should be retained, then one can simply use this algorithm as if all the attributes appear in the `SELECT` clause (equivalently, as if all the variables of  $\text{var}(Q)$  appear in the head) and apply the required projection just before printing each tuple.

## 4 Ordered Evaluation with Polynomial Delay

### 4.1 The Main Theorem

In this section, we prove our main theorem. In the next section, we describe families of orders that satisfy the first condition of this theorem, assuming that the CQ  $Q$  is acyclic. We start with some notation and definitions.

Consider an ACQ  $Q$ , with the head  $Q(\mathbf{u})$ , and a database  $D$ . We denote by  $Q^+$  the query that is obtained from  $Q$  by adding the conjunct  $R_X(X)$  for each variable  $X$  of  $\mathbf{u}$ . Note that  $R_X$  is a relation scheme with arity 1. The database  $D^{Q(\mathbf{u})}$  is obtained from  $D$  by adding a relation instance  $I_X$  for each  $R_X$  as follows.  $I_X$  comprises all constants  $d$  of  $D$ , such that there is a conjunct  $R_i(\mathbf{u}_i)$  of  $Q$ , the variable  $X$  is in  $\mathbf{u}_i$  and the constant  $d$  appears in the relation  $R_i[D]$  in a column that corresponds to  $X$ . In other words, the relation for  $R_X$  contains

(at least) all constants that could potentially be the image of  $X$  under some homomorphism from  $Q$  to  $D$ . The following proposition is rather straightforward.

**Proposition 1.** *Let  $Q$  be a conjunctive query and  $D$  be a database. Then,*

- $Q(D) = Q^+(D^{Q(\mathbf{u})})$ ;
- $\text{Hom}(Q, D) = \text{Hom}(Q^+, D^{Q(\mathbf{u})})$ ; and
- $Q$  is acyclic if and only if  $Q^+$  is acyclic.

According to the above proposition, if we are interested in evaluating a CQ  $Q$  over a database  $D$  or finding a maximal homomorphism of  $\text{Hom}(Q, D)$ , we can instead solve these problems with respect to the query  $Q^+$  and the database  $D^{Q(\mathbf{u})}$ . Furthermore, this transformation preserves acyclicity of  $Q$ , namely, if  $Q$  is a cyclic, then so is  $Q^+$ .

Our main result is the next theorem that uses the above proposition in order to show the following. An efficient algorithm for finding a maximal homomorphism is sufficient for efficiently enumerating all answers and all homomorphisms of a CQ  $Q$  with respect to a database  $D$ . Formally, we need to make the natural assumption that if an algorithm is efficient with respect to a CQ  $Q$  and a database  $D$ , then it is also correct and efficient with respect to every database  $\hat{D}$  that is obtained from  $D$  by deleting some tuples (i.e.,  $\hat{D} \subseteq D$ ). Note that when considering  $\hat{D}$  instead of  $D$ , we assume that the order on the homomorphisms of  $\text{Hom}(Q, \hat{D})$  is the same as the one defined on  $\text{Hom}(Q, D)$ .

**Theorem 1.** *Consider a CQ  $Q$ , a database  $D$  and an order  $\succeq$  on  $\text{Hom}(Q, D)$ . The following are equivalent.*

1. *There is a polynomial-time algorithm for solving the following problem. Given the CQ  $Q$  and a database  $\hat{D} \subseteq D^{Q(\mathbf{u})}$ , find a maximal homomorphism of  $\text{Hom}(Q^+, \hat{D})$ . (The running time is polynomial in the size of  $Q^+$  and  $\hat{D}$ .)*
2. *There are enumeration algorithms that run with polynomial delay for the following two problems. Given the CQ  $Q$  and a database  $\hat{D} \subseteq D^{Q(\mathbf{u})}$ , enumerate all the answers of  $Q^+(\hat{D})$  and all the homomorphisms of  $\text{Hom}(Q^+, \hat{D})$  in ranked order. (The delay is polynomial in the size of  $Q^+$  and  $\hat{D}$ .)*

Clearly, the second part implies the first. So, we now prove the other direction. Our proof is for the case where we want to enumerate  $Q(D)$  (or, equivalently,  $Q^+(D^{Q(\mathbf{u})})$ ). For that, we describe the algorithm  $\text{SORTEDEVAL}_{\succeq}(Q, D)$  of Figure 3 that enumerates the answers of  $Q(D)$  in ranked order, using the subroutine  $\text{MAXIMIZE}_{\succeq}(Q^+, \hat{D})$  that finds a maximal homomorphism of  $\text{Hom}(Q^+, \hat{D})$ , where  $\hat{D} \subseteq D^{Q(\mathbf{u})}$ . Our algorithm is an adaptation of a procedure<sup>2</sup> by Lawler [13] for computing the top- $k$  solutions to discrete optimization problems. Proving the theorem for an arbitrary  $\hat{D} \subseteq D^{Q(\mathbf{u})}$  requires straightforward modifications to the algorithm. Note that in order to use  $\text{SORTEDEVAL}_{\succeq}(Q, D)$  for enumerating all the homomorphisms of  $\text{Hom}(Q, D)$ , we need to modify  $Q$  so that the head includes all the variables of  $Q$ .

<sup>2</sup> This procedure generalizes an algorithm of Yen [16] for enumerating simple paths.

```

SORTEDEVAL $\succeq$ ( $Q, D$ )
1   $\mathbf{u} \leftarrow$  the tuple of variables from the head of  $Q$ 
2   $Queue \leftarrow$  an empty priority queue, with priority based on  $\succeq$ 
3   $\varphi \leftarrow \text{MAXIMIZE}_{\succeq}(Q^+, D^{Q(\mathbf{u})})$ 
4  if  $\varphi \neq \perp$ 
5    then  $Queue.\text{INSERT}(\langle \emptyset, \emptyset, \varphi \rangle)$ 
6  while  $Queue$  is not empty
7    do  $\langle P, N, \varphi \rangle \leftarrow Queue.\text{REMOVETOP}()$ 
8       $\{X_1, \dots, X_k\} \leftarrow$  the variables that appear in  $\mathbf{u}$  but not in  $P$ 
9      for  $i \leftarrow 1$  to  $k$ 
10       do  $P_i \leftarrow P \cup \{X_1 = \varphi(X_1), \dots, X_{i-1} = \varphi(X_{i-1})\}$ 
11          $N_i \leftarrow N \cup \{X_i \neq \varphi(X_i)\}$ 
12          $D_i \leftarrow D^{Q(\mathbf{u})}$ 
13         for all  $(X = a) \in P_i$ 
14           do remove from  $D_i[R_X]$  all tuples except for  $\langle a \rangle$ 
15         for all  $(Y \neq b) \in N_i$ 
16           do remove from  $D_i[R_Y]$  the tuple  $\langle b \rangle$ 
17          $\varphi_i \leftarrow \text{MAXIMIZE}_{\succeq}(Q^+, D_i)$ 
18         if  $\varphi_i \neq \perp$ 
19           then  $Queue.\text{INSERT}(\langle P_i, N_i, \varphi_i \rangle)$ 
20       PRINT( $\varphi\mathbf{u}$ )

```

**Fig. 3.** Incrementally computing answers in sorted order

The algorithm  $\text{SORTEDEVAL}_{\succeq}(Q, D)$  uses two types of *constraints*. A *positive* constraint has the form  $X = a$  and a *negative* constraint has the form  $Y \neq b$ , where both  $X$  and  $Y$  are variables from the head of  $Q$  while  $a$  and  $b$  are constants of  $D$ . We use  $P$  and  $P_i$  to denote sets of positive constraints whereas  $N$  and  $N_i$  denote sets of negative constraints. A homomorphism  $\varphi$  *satisfies* the sets of constraints  $P$  and  $N$  if  $\varphi(X) = a$  for all constraint  $X = a$  of  $P$  and  $\varphi(Y) \neq b$  for every constraint  $Y \neq b$  of  $N$ .

$\text{SORTEDEVAL}_{\succeq}$  uses a priority queue,  $Queue$ . The operations on  $Queue$  take logarithmic time in the size of  $Queue$  (hence, polynomial time in  $Q$  and  $D$ ). An element of  $Queue$  is a triplet  $\langle P, N, \varphi \rangle$ , where  $P$  and  $N$  are sets of positive and negative constraints, respectively, and  $\varphi \in \text{Hom}(Q, D)$  is a maximal homomorphism among all those satisfying  $P$  and  $N$ . Priority in  $Queue$  is based on  $\succeq$ ; that is, the top element of  $Queue$  is a triplet  $\langle P, N, \varphi \rangle$ , such that  $\varphi \succeq \varphi'$  for all triplets  $\langle P', N', \varphi' \rangle$  in  $Queue$ . A triplet  $\langle P, N, \varphi \rangle$  represents the subset of  $Q(D)$  comprising all answers that are produced by homomorphisms satisfying  $P$  and  $N$ , where  $\varphi$  is the “best” such homomorphism. The triplets of  $Queue$  represent disjoint subsets that cover all the answers that have not yet been printed.

The first element inserted into  $Queue$  (Lines 3–5) represents the whole set  $Q(D)$ . In the loop of Lines 7–20, each iteration starts by removing the top ele-

ment  $\langle P, N, \varphi \rangle$  from *Queue*. The answer  $\varphi \mathbf{u}$  is printed in Line 20. Let  $X_1, \dots, X_k$  be the variables that appear in  $\mathbf{u}$  but not in  $P$  (intuitively, these are the free variables of the answers represented by  $\langle P, N, \varphi \rangle$ ). In Lines 8–19, we partition all the answers represented by  $\langle P, N, \varphi \rangle$ , except for  $\varphi \mathbf{u}$ , into  $k$  disjoint subsets by creating the elements  $\langle P_i, N_i, \varphi_i \rangle$ ,  $1 \leq i \leq k$ , and inserting them into *Queue* (whenever  $\varphi_i$  exists, i.e.,  $\varphi_i \neq \perp$ ). The set  $P_i$  is obtained from  $P$  by adding the constraints  $X_j = \varphi(X_j)$  for  $j = 1, \dots, i - 1$ . The set  $N_i$  is obtained from  $N$  by adding the constraint  $X_i \neq \varphi(X_i)$ . The maximal homomorphism  $\varphi_i$ , under  $P_i$  and  $N_i$ , is generated in Lines 13–17 by executing  $\text{MAXIMIZE}_{\succeq}(Q^+, D_i)$ , where  $D_i$  is the database that is obtained by changing  $D^{Q(\mathbf{u})}$  so that all constraints are satisfied: (1) For each constraint  $X = a$  of  $P_i$ , we remove from  $D^{Q(\mathbf{u})}[R_X]$  all tuples except for  $\langle a \rangle$ , and (2) for each constraint  $Y \neq b$  of  $N_i$ , we remove the tuple  $\langle b \rangle$  from  $D^{Q(\mathbf{u})}[R_Y]$ . The following lemma shows the correctness of the algorithm  $\text{SORTEDEVAL}_{\succeq}(Q, D)$ . It also shows that this algorithm enumerates with polynomial delay if  $\text{MAXIMIZE}_{\succeq}$  runs in polynomial time.

**Lemma 1.** *Given a CQ  $Q$ , the algorithm  $\text{SORTEDEVAL}_{\succeq}(Q, D)$  enumerates  $Q(D)$  in ranked order. The delay is polynomial provided that  $\text{MAXIMIZE}_{\succeq}$  runs in polynomial time.*

## 4.2 Tractable Orders

In this Section, we present two families of orders that satisfy the first part of Theorem 1 and, hence, also the second part, assuming that the CQ  $Q$  is acyclic.

**Monotonic orders.** This family includes many widely used orders, such as sums of single-variable functions (or products, if the functions are positive) and lexicographic orders. In order to define monotonic orders, we need to consider partial assignments, i.e., mappings from some of the variables of the given query to constants of the database. First, consider two partial assignments  $\varphi$  and  $\varphi'$  that are defined on the sets of variables  $V$  and  $V'$ , respectively. The *combined* assignment  $\varphi' \oplus \varphi$  maps variables of  $V'$  according to  $\varphi'$  and variables of  $V \setminus V'$  according to  $\varphi$  (i.e.,  $\varphi'$  takes precedence when mapping variables that are in the domains of both assignments). Now, an order  $\succeq$  is *monotonic* if for all partial assignments  $\varphi, \varphi_1$  and  $\varphi_2$ , such that  $\varphi_1$  and  $\varphi_2$  are defined on the same set of variables, if  $\varphi_1 \succeq \varphi_2$  then  $\varphi_1 \oplus \varphi \succeq \varphi_2 \oplus \varphi$ .<sup>3</sup> As an example, suppose that each variable  $X$  is associated with a ranking function  $\text{rank}_X$ . Then, the following definitions of  $\text{rank}(\varphi)$  imply monotonic orders (regardless of the specific form of  $\text{rank}_X$ ). Note that  $V$  denotes the domain of  $\varphi$ .

$$\text{rank}(\varphi) = \sum_{X \in V} \text{rank}_X(\varphi(X)),$$

<sup>3</sup> For simplicity's sake, the definition of monotonic orders is more restrictive than necessary. A more general definition can be obtained by taking into account the structure of the join tree of the given query.

$$\text{rank}(\varphi) = \max\{\text{rank}_X(\varphi(X)) \mid X \in V\},$$

$$\text{rank}(\varphi) = \min\{\text{rank}_X(\varphi(X)) \mid X \in V\}.$$

The class of monotonic orders also contains the *lexicographic orders*. For example, every order specified by the SQL command `ORDER BY  $R_1.A_1, \dots, R_k.A_k$`  (with an optional `DESC` attached to each attribute) is lexicographic. We define this type of orders on homomorphisms.<sup>4</sup> Given a CQ  $Q$ , a lexicographic order  $\succeq$  is specified by a sequence  $(X_1, \dots, X_k)$  of distinct variables of  $Q$ . We assume that for every variable  $X_i$ , there is an order  $>_i$  over the values (of the given database  $D$ ) that  $X_i$  could be mapped to. Two homomorphisms  $\varphi_1, \varphi_2 \in \text{Hom}(Q, D)$  satisfy  $\varphi_1 \succeq \varphi_2$  if either (1)  $\varphi_1$  and  $\varphi_2$  agree on every variable  $X_i$  or (2) the minimal  $i$ , such that  $\varphi_1(X_i) \neq \varphi_2(X_i)$ , satisfies  $\varphi_1(X_i) >_i \varphi_2(X_i)$ .

We briefly sketch how to find a maximal homomorphism, under a monotonic order, for a given ACQ  $Q$  and a database  $D$ . Note that the running time is polynomial in the size of  $Q$  and  $D$ . First, we construct a rooted join tree  $T$  of  $Q$  (recall that the nodes of  $T$  are the conjuncts of  $Q$ ). Then, we traverse  $T$  bottom up. For each conjunct  $R(\mathbf{u}_i)$  and tuple  $t \in D[R]$ , we compute the maximal partial assignment  $\varphi_t$  to the variables of the subtree rooted at  $R(\mathbf{u}_i)$ , such that  $\varphi_t$  maps  $\mathbf{u}_i$  to  $t$ . A detailed description is beyond the scope of this paper.

**$c$ -determined orders.** This type of orders generalizes the one given in [6] and is defined as follows. Given a fixed positive integer  $c$ , an order  $\succeq$  over homomorphisms is  *$c$ -determined* if for all  $\varphi \in \text{Hom}(Q, D)$ , there exists a set  $C_\varphi$  of  $c$  or fewer variables, such that every homomorphism  $\varphi' \in \text{Hom}(Q, D)$  that agrees with  $\varphi$  on the variables of  $C_\varphi$  must satisfy  $\varphi' \succeq \varphi$ . For example, an order that is defined by a ranking function over 3 variables, e.g., the one used in the query of Figure 2, is 3-determined. As another example, an order that is defined by taking the maximal rank over all variables (as defined above) is 1-determined. Note that a  $c$ -determined order is not necessarily monotonic and vice versa (even if the order is determined by a fixed set of variables  $C$ , i.e.,  $C_\varphi = C$  for all  $\varphi$ ).

Given an ACQ  $Q$ , a database  $D$  and a  $c$ -determined order  $\succeq$ , we can find a maximal homomorphism as follows. We consider every subset  $C$  of  $\text{var}(Q)$ , with  $c$  or fewer variables, and every possible assignment  $\psi$  that maps  $C$  to constants from  $D$ . For each  $\psi$ , we compute a homomorphism  $\varphi_C$  (if it exists) that agrees with  $\psi$  on  $C$ . The desired homomorphism is the maximal one among all the  $\varphi_C$ .

The following theorem summarizes the above discussion.

**Theorem 2.** *Consider an ACQ  $Q$  and database  $D$ . For the following orders, finding a maximal homomorphism is polynomial in  $Q$  and  $D$ . Hence, all answers can be generated in ranked order with polynomial delay.*

- *Monotonic (e.g., lexicographic) orders.*
- *$c$ -determined orders, where  $c$  is a fixed positive integer.*

<sup>4</sup> We omit both the definition of these orders on partial assignments and a proof that they are monotonic, due to a lack of space.

## 5 Space-Efficient Evaluation

The algorithm `SORTEDEVAL` of Figure 3 runs with polynomial delay, but its queue may grow linearly with respect to the number of answers. In this section, we briefly discuss an alternative technique that applies to certain types of orders. This technique has a polynomial delay and uses only a polynomial amount of space. It is based on the following proposition.

**Proposition 2.** *Let  $Q$  be an ACQ and  $D$  be a database. The answers of  $Q(D)$  can be enumerated (in an arbitrary order) with polynomial delay and polynomial space.*

If the order is either lexicographic or  $c$ -determined by a fixed set of variables and, moreover, it depends only on variables that appear in the head of  $Q$ , then we can use the above result as follows. We bind the variables of the head that determine the rank to constants from the database. Then, we sort the different bindings. In the case of a  $c$ -determined order, there is only a polynomial number of bindings. If the order is lexicographic, then we have to do it one variable at a time, in a recursive manner. For a given binding, we can determine whether there is any answer and, if so, enumerate all the answers using the above proposition. Additional details are beyond the scope of this paper.

## 6 Conclusion

We have investigated the complexity of incrementally computing SQL queries with the `ORDER BY` clause. We have stated actual tractability results for SQL queries that correspond to acyclic conjunctive queries, since solving efficiently the non-emptiness problem is a necessary condition for incremental computation with polynomial delay. In our framework, the `ORDER BY` clause is modeled as an order on homomorphisms from the conjunctive query to the database, since the `ORDER BY` clause may include attributes that do not appear in the `SELECT` clause.

Our main result holds for general conjunctive queries and it states that if one can find the top-ranked homomorphism in polynomial time, then all the tuples of the result can be enumerated in sorted order with polynomial delay (subject to the natural assumption that the given algorithm for finding a maximal homomorphism can be applied efficiently and correctly to any database that is obtained from the given one by deleting some tuples). For acyclic conjunctive queries, we have shown that  $c$ -determined and monotonic (which include lexicographic) orders satisfy this property. If all attributes of the `ORDER BY` clause also appear in the `SELECT` clause, then the computation requires only polynomial space if the order is either  $c$ -determined by a fixed set of variables  $C$  or lexicographic. Our results can be easily extended to unions of acyclic conjunctive queries. In this case, enumeration in sorted order runs in incremental polynomial time if duplicates are eliminated; otherwise, the delay is polynomial.

Some related work dealt with problems that are similar to, yet different from the one considered in this paper. In [7, 8], the goal is to find the top- $k$  objects,

where ranking is determined by a monotone function of several attributes and the ranking order on each attribute is given by a sorted list. The PREFER system [10] is aimed at finding the top- $k$  tuples of a relation, based on a combination of several ranking functions. Query processing is optimized by using sorted views.

In summary, the worst-case delay of our algorithms is much better (polynomial vs. exponential) than the delays of previous algorithms, but more work is needed in order to incorporate our algorithms in practical systems.

## References

1. C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the desirability of acyclic database schemes. *J. ACM*, 30(3):479–513, 1983.
2. P. A. Bernstein and N. Goodman. Power of natural semijoins. *SIAM J. Comput.*, 10(4):751–771, 1981.
3. M. J. Carey and D. Kossmann. On saying "enough already!" in SQL. In *SIGMOD*, pages 219–230, 1997.
4. M. J. Carey and D. Kossmann. Reducing the braking distance of an SQL query engine. In *VLDB*, pages 158–169, 1998.
5. A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, pages 77–90, 1977.
6. S. Cohen and Y. Sagiv. An incremental algorithm for computing ranked full disjunctions. In *PODS*, 2005.
7. R. Fagin. Combining fuzzy information from multiple systems. *J. Comput. Syst. Sci.*, 58(1):83–99, 1999.
8. R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003.
9. D. Habich, W. Lehner, and A. Hinneburg. Optimizing multiple top-k queries over joins. In *SSDBM*, pages 195–204, 2005.
10. V. Hristidis, N. Koudas, and Y. Papakonstantinou. PREFER: A system for the efficient execution of multi-parametric ranked queries. In *SIGMOD*, 2001.
11. I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. In *VLDB*, pages 754–765, 2003.
12. D. S. Johnson, M. Yannakakis, and C. H. Papadimitriou. On generating all maximal independent sets. *Information Processing Letters*, 27:119–123, March 1988.
13. E. L. Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science*, 18:401–405, 1972.
14. A. Natsev, Y. C. Chang, J. R. Smith, C. S. Li, and J. S. Vitter. Supporting incremental join queries on ranked inputs. In *VLDB*, pages 281–290, 2001.
15. M. Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, pages 82–94, 1981.
16. J. Y. Yen. Another algorithm for finding the  $k$  shortest loopless network paths. In "Proc. 41st Mtg. Operations Research Society of America", volume 20, page B/185, 1972.