Full Disjunctions: Polynomial-Delay Iterators in Action

Sara Cohen
Technion
Haifa, Israel
sarac@ie.technion.ac.il

Itzhak Fadida
Technion
Haifa, Israel
tzahi@tx.technion.ac.il

Yaron Kanza
University of Toronto
Toronto, Canada
yaron@cs.toronto.edu

Benny Kimelfeld
The Hebrew University
Jerusalem, Israel
bennyk@cs.huji.ac.il

Yehoshua Sagiv
The Hebrew University
Jerusalem, Israel
sagiv@cs.huji.ac.il

ABSTRACT

Full disjunctions are an associative extension of the outerjoin operator to an arbitrary number of relations. Their main advantage is the ability to *maximally* combine data from different relations while preserving all the original information. An algorithm for efficiently computing full disjunctions is presented. This algorithm is superior to previous ones in three ways. First, it is the first algorithm that computes a full disjunction with a polynomial delay between tuples. Hence, it can be implemented as an iterator that produces a stream of tuples, which is important in many cases (e.g., pipelined query processing and Web applications). Second, the total runtime is linear in the size of the output. Third, the algorithm employs a novel optimization that divides the relation schemes into biconnected components, uses a separate iterator for each component and applies outerjoins whenever possible. Combining efficiently full disjunctions with standard SQL operators is discussed. Experiments show the superiority of our algorithm over the state of the art.

1. INTRODUCTION

In many scenarios, databases that were built independently contain related information. The Internet has made many such databases accessible and enhanced the need for reliable and efficient data-integration techniques. The goal of data integration is to enable users to utilize all the available information from independent data sources. To that end, integration techniques should connect related pieces of information in a maximal way, that is, all and only related pieces should be combined without causing any loss of information.

Integration of two relations can be done using the *natural outerjoin* operator (or *outerjoin* for short) [4, 5, 9]. However, the outerjoin is not suitable for integrating more than two relations; for example, the outerjoin is not associative. The *full disjunction* [7] is an associative and commutative generalization of the outerjoin operator. Hence, it is suitable for integrating any number of relations. Intuitively, for a given set of relations, the full disjunction is obtained by joining maximal sets of related tuples (i.e., tuples that agree on all the common attributes). Since every given tuple is included in at least one tuple of the result, all the available information is preserved. In comparison, the tuples produced by a sequence of outerjoins include all the original information, but are not necessarily maximal, i.e., outerjoins cannot always combine all related pieces of information. Thus, in general, results of outerjoins may have redundant tuples or suffer from information loss (since some related tuples of the source relations are not joined in the result).

Our goal is to provide practically efficient algorithms for computing full disjunctions. Such algorithms are essential for incorporating full disjunctions into standard query languages. The next example demonstrates the importance of a full-disjunction operator as a primitive language construct.

Example 1.1. Suppose that the following relations are stored in a database containing touristic information: *Climates*(Country, Climate), *Accommodations*(Country, City, Hotel, Stars) and *Sites*(Country, City, Site). We would like to find a place to visit, based on the information of our database. The natural join of the three relations above may not yield the desired result, since a join may cause interesting data to be lost due to missing information in the tables. For some cities, for example, the database may contain no sites at all. For such cities, there is no corresponding tuple in the *Sites* relation. As another example, some national sites may be out of any specific city and, hence, the tuples of these sites will not join with any tuple of *Accommodations*. To prevent loss of information, we would like to be able to pose the following query.

```
SELECT Country,City,Stars,Site
FROM FD(Climates,Accommodations,Sites) AS F
WHERE F.Climate = 'tropical'
ORDER BY Stars
```

In this query, `FD(Climates,Accommodations,Sites)` is an operation that computes the full disjunction of its arguments. Intuitively, this query returns maximal information about tourist sites and accommodations in countries with tropical climate, ranked by the rating of the hotels. □

VLDB '06, September 12-15, 2006, Seoul, Korea.

When evaluating queries, such as that of Example 1.1, an efficient algorithm for computing full disjunctions is clearly needed. The best known algorithm for computing full disjunctions [2] runs in time that is quadratic in the size of the output. Since the output of a full disjunction is typically large (it includes, at least, all the tuples of the source relations), it is clearly desirable to reduce the dependency to optimum, i.e., achieve linear time in the size of the output (and, of course, polynomial time in the size of the input). One way of achieving linear dependency in the output is to devise an algorithm that generates the tuples of the result incrementally, where the time interval between generating a tuple of the result and generating the next tuple is small. Formally, we say that an algorithm runs in *polynomial delay* if the delay between successive tuples is polynomial only in the size of the input. An algorithm that runs with polynomial delay has a total time that is linear in the size of its output. In this paper, we present an algorithm for computing full disjunctions that is aimed at achieving small delays between tuples. In particular, we present the first algorithm for computing full disjunctions with polynomial delay.

Computing full disjunctions with short delays between tuples is in itself an important goal (regardless of the total computation time), since there are applications that can benefit from getting tuples of the result as a stream that starts flowing as soon as possible, before the whole computation is complete. A motivating example is a parallel system, where one processor computes the tuples of a full disjunction and another processor receives these tuples for further processing. A short delay is required to achieve a proper utilization of the second processor. As another example, consider a server on the Web that computes a full disjunction and then sends the result over the Internet. While the computation is going on, tuples are sent as soon as they are produced, thereby reducing latency and preventing bandwidth bottlenecks. A third important example is when a user inspects the tuples one at a time, as soon as they are generated, and may even decide at any moment that no more tuples are needed and the computation can terminate. In such a scenario, the system should not let the user wait for too long when she is asking for the next tuple. Thus, minimizing the delay is important and it may even save resources, since computing the whole result is not always required.

Related Work. In some cases, full disjunctions can be formulated as sequences of outerjoins. In [13], it was shown that the full disjunction of a given set of relations is equivalent to some sequence of outerjoins if and only if the relation schemes form a γ-acyclic hypergraph [6]. Based on this result, an algorithm for efficiently computing full disjunctions of γ-acyclic sets of relations by a sequence of outerjoins was proposed in [13]. An algorithm that computes full disjunctions of an arbitrary set of relations was presented in [12]. The above two algorithms run in polynomial time in the size of the input and the output. In other words, all the tuples of the result are returned in time that is polynomial in the combined size of the given relations and their full disjunction. Note, however, that these algorithms do not return any tuple until all processing is complete (and cannot be easily adapted to do so).

In [2], an incremental algorithm for computing full disjunctions was presented. In this algorithm, the delay is short while the first several tuples are generated, but it gets longer as the computation goes on. Formally, the computation is in *incremental polynomial time*, i.e., the delay between the ith and the $(i+1)$st tuples of the result is polynomial in the combined size of the input and the first i tuples that are generated. Note, however, that the substantial slowdown between successive tuples is not suitable in the scenarios described earlier. The algorithm of [2] can also be adapted to generating tuples in ranked order provided that the ranking function is "monotonically c-determined."

Contributions. Our main contribution is the algorithm BiComNLOJ for efficiently computing full disjunctions in the general case. This algorithm is superior to previous ones in three ways. Foremost, it is the first algorithm that computes a full disjunction with polynomial delay. Second, the total runtime is linear in the size of the output, whereas the best previously known algorithm [2] runs in time that is quadratic in the size of the output. Third, by dividing the scheme graph of the relations into biconnected components, our algorithm can further optimize the runtime (and shorten the delay) by incorporating outerjoins into the computation.

Before discussing BiComNLOJ, we first present two algorithms, namely NLOJ and PDelayFD, that we build upon to derive this general algorithm. Section 3 describes the algorithm NLOJ that uses left-deep outerjoins to compute a full disjunction of an acyclic[1] set of relations. This algorithm is highly efficient with linear delay (in the size of the input) between results. The algorithm PDelayFD (Section 4) computes a full disjunction for an arbitrary set of relations with a polynomial delay between tuples. We note that PDelayFD is already a significant improvement over the state of the art [2], since it runs in polynomial delay and in total time that is linear (rather than quadratic) in the size of the output. Finally, BiComNLOJ, which combines the efficiency of NLOJ with the generality of PDelayFD, is presented (Section 5). The combination of NLOJ and PDelayFD is rather intricate, since special techniques must be applied to retain the polynomial delay between tuples.

In this paper, we also show how full disjunctions can be integrated with standard SQL operators (Section 6). In particular, we show how and when selection, projection and order-by can be integrated directly into the processing of a full disjunction.

All algorithms discussed in this paper, as well as the algorithm of [2], were implemented as block-based algorithms within an open-source database system. Extensive experimentation was performed to prove the superiority of BiComNLOJ over the state-of-art algorithm ([2]) in the total computation time as well as the delays between tuples in the result (Section 7). Our experimentations also show that the technique employed in BiComNLOJ of dividing the scheme graph into biconnected components is of great importance for deriving an algorithm that can be efficiently integrated into a real database system.

2. PRELIMINARIES

In this section, we present our data model, we define the concept of a full disjunction, and we discuss enumeration algorithms that act as iterators.

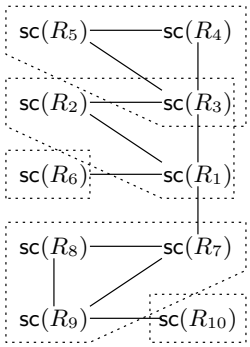
2.1 Relations and Schemes

In this work, we use the relational data model. Relations,

[1]Our notion of acyclicity is defined in Section 2.1.

rel.	attributes
R_1	A, B, C, D
R_2	B, E
R_3	A, E, F, G
R_4	F, H
R_5	G, H
R_6	C, I
R_7	D, J, K
R_8	K, L
R_9	J, L, M
R_{10}	M, N

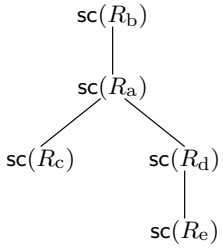
(a) Schemes of $\mathcal{R}_1$



(b) Scheme graph $\mathcal{G}_{\mathsf{sc}}(\mathcal{R}_1)$

rel.	attributes
R_{a}	A, B, C, D, E, F, G
R_{b}	A, E, F, G, H
R_{c}	C, I
R_{d}	D, J, K, L, M
R_{e}	M, N

(c) Schemes of $\mathcal{R}_{1\mathrm{a}}$



(d) Scheme graph $\mathcal{G}_{\mathsf{sc}}(\mathcal{R}_{1\mathrm{a}})$

Figure 1: Schemes of relation sets $\mathcal{R}_1$ and $\mathcal{R}_{1\mathbf{a}}$

tuples and schemes are defined in the usual way[2]. Given a tuple t and an attribute A of t, we use $t[A]$ to denote the value of t for A. We use $\perp$ to denote the *null* value. Given a relation R, we use $\mathsf{sc}(R)$ to denote the *scheme* of R, i.e., the set comprising all the attributes of R. Given a tuple t_1 of a relation R_1 and a tuple t_2 of another relation R_2, we say that t_1 and t_2 are *join consistent* if they *agree* (i.e., are equal and nonnull) on common attributes, that is, $t_1[A] = t_2[A] \neq \perp$ holds for every attribute A in $\mathsf{sc}(R_1) \cap \mathsf{sc}(R_2)$.

We use $\mathcal{R}$ (possibly with a subscript) to denote a set of relations. Consider a set $\mathcal{R} = \{R_1, \ldots, R_n\}$ of relations. The *scheme graph* of $\mathcal{R}$, denoted $\mathcal{G}_{\mathsf{sc}}(\mathcal{R})$, is the undirected graph that is defined as follows. The nodes of $\mathcal{G}_{\mathsf{sc}}(\mathcal{R})$ are the schemes $\mathsf{sc}(R_1), \ldots, \mathsf{sc}(R_n)$. There is an edge between two schemes if they share one or more common attributes. More formally, $\mathcal{G}_{\mathsf{sc}}(\mathcal{R})$ contains an edge between $\mathsf{sc}(R_i)$ and $\mathsf{sc}(R_j)$ if $i \neq j$ and $\mathsf{sc}(R_i) \cap \mathsf{sc}(R_j) \neq \emptyset$. We say that $\mathcal{R}$ is *connected* if $\mathcal{G}_{\mathsf{sc}}(\mathcal{R})$ is connected. $\mathcal{R}$ is *cyclic* if $\mathcal{G}_{\mathsf{sc}}(\mathcal{R})$ contains a cycle; otherwise, it is *acyclic*. Note that our notion of acyclicity implies γ-acyclicity, but the two are not equivalent.

EXAMPLE 2.1. The scheme graphs of two sets of relation $\mathcal{R}_1 = \{R_1, \ldots, R_{10}\}$ and $\mathcal{R}_{1\mathrm{a}} = \{R_{\mathrm{a}}, \ldots, R_{\mathrm{e}}\}$ is depicted in Figure 1. The dotted polygons in Figure 1(b) should be ignored for now—their meaning will be explained later. Note that both $\mathcal{R}_1$ and $\mathcal{R}_{1\mathrm{a}}$ are connected, $\mathcal{R}_1$ is cyclic and $\mathcal{R}_{1\mathrm{a}}$ is acyclic. □

2.2 Full Disjunctions

Consider a set of relations $\mathcal{R} = \{R_1, \ldots, R_n\}$. A *tuple set* of $\mathcal{R}$ is any set of tuples $T = \{t_1, \ldots, t_m\}$ consisting of at most one tuple from each relation (hence, $m \leq n$). We say that T is *connected* if the tuples of T belong to a connected subset of $\mathcal{R}$; that is, if $\{R_{i_1}, \ldots, R_{i_m}\}$ is connected, where each R_{i_j} is the relation that contains t_j. We say that T is *join consistent* if every two tuples in T are join consistent; that is, for every $1 \leq j < k \leq m$, the tuples t_j and t_k are join consistent. For clarity, we assume that different tuples from the same relation are not join consistent. $JC(T)$ denotes that T is join consistent and $JCC(T)$ denotes that T is both join consistent and connected.

[2]The algorithms in this paper assume that relations are sets of tuples, but they can be easily adapted to the case where relations are multisets of tuples.

	A	B	C
t_1	1	10	1
t_2	2	21	2
t_3	1	$\perp$	3

(a) R_{11}

	A	B	D
u_1	1	10	1
u_2	2	22	2

(b) R_{12}

	A	E	F
v_1	1	11	1
v_2	2	20	2

(c) R_{13}

	A	E	G
w_1	1	12	1
w_2	2	20	2

(d) R_{14}

Figure 2: Relations R_{11}, R_{12}, R_{13} and R_{14}

DEFINITION 2.2. *Let $\mathcal{R}$ be a set of relations. MaxJCC($\mathcal{R}$) is the set of all tuple sets T of $\mathcal{R}$, such that* (1) *JCC(T),* and (2) *T is maximal, that is, no tuple set of $\mathcal{R}$ that is both join consistent and connected also properly contains T.*

EXAMPLE 2.3. Let $\mathcal{R} = \{R_{11}, R_{12}, R_{13}, R_{14}\}$, where R_{11}, R_{12}, R_{13} and R_{14} are the relations of Figure 2. Consider the tuple set $T = \{t_1, u_1, v_1\}$. It is easy to see that $JCC(T)$ holds, since *(1)* the scheme graph of $\{R_{11}, R_{12}, R_{13}\}$ is connected *and (2)* t_1, u_1 and v_1 agree on their common attributes. Furthermore, T is in $MaxJCC(\mathcal{R})$, since $JCC(T')$ does not hold for any tuple set T' that properly contains T. The reader can easily verify that $MaxJCC(\mathcal{R})$ comprises the following six tuple sets: $\{t_1, u_1, v_1\}$, $\{t_2, v_2, w_2\}$, $\{t_3, v_1\}$, $\{u_2, v_2, w_2\}$, $\{t_1, u_1, w_1\}$ and $\{t_3, w_1\}$. □

Consider a join-consistent tuple set T. Let S be a scheme that contains all the attributes that appear in the tuples of T and perhaps additional attributes. We use $embed_S(T)$ to denote the tuple over S that is obtained by first applying the natural join to the tuples of T and then adding columns with nulls for the remaining attributes of S. In other words, $embed_S(T)$ is the tuple t, such that for all attributes A of S, if T contains a tuple t' with the attribute A, then $t[A] = t'[A]$; otherwise, $t[A] = \perp$. We use $embed_S(t)$ as a shorthand notation for $embed_S(\{t\})$.

As an example, let $T = \{t_1, u_1, v_1\}$ be the tuple set from Example 2.3. Suppose that S is the set of all attributes appearing in the relations R_{11}, R_{12}, R_{13} and R_{14} of Figure 2, i.e., $S = \{A, B, C, D, E, F, G\}$. Then, $embed_S(T)$ is the tuple $t = (A : 1, B : 10, C : 1, D : 1, E : 11, F : 1, G : \perp)$. Note that $t[G] = \perp$, since G does not appear in R_{11}, R_{12} or R_{13}.

Given a set of relations $\mathcal{R}$, the full disjunction of $\mathcal{R}$ is obtained by first applying the natural join to the tuples of each $T \in MaxJCC(\mathcal{R})$ and then taking the outer union. Following is a formal definition.

DEFINITION 2.4 (FULL DISJUNCTION). *Let $\mathcal{R}$ be a set*

	A	B	C	D	E	F	G
s_1	1	10	1	1	11	1	$\perp$
s_2	2	21	2	$\perp$	20	2	2
s_3	1	$\perp$	3	$\perp$	11	1	$\perp$
s_4	2	22	$\perp$	2	20	2	2
s_5	1	10	1	1	12	$\perp$	1
s_6	1	$\perp$	3	$\perp$	12	$\perp$	1

Figure 3: $FD(\{R_{11}, R_{12}, R_{13}, R_{14}\})$

of relations and S be the union of the schemes of the relations in $\mathcal{R}$. The full disjunction *of $\mathcal{R}$, denoted $FD(\mathcal{R})$, is the relation $\{embed_S(T) \mid T \in MaxJCC(\mathcal{R})\}$.*

As an example, consider again the relations R_{11}, R_{12}, R_{13} and R_{14}, presented in Figure 2. The full disjunction of these relations consists of 6 tuples and is shown in Figure 3. Tuple s_1, for instance, is the tuple $embed_S(\{t_1, u_1, v_1\})$, where $S = \{A, B, C, D, E, F, G\}$.

Our definition of full disjunctions differs from that of [13] in that we allow the source relations to contain null values. When source relations do not have null values, the two definitions are equivalent.

2.3 Enumeration Algorithms

In this paper, we develop enumeration algorithms for computing full disjunctions. An *enumeration algorithm* E is an algorithm that acts as an *iterator* [10] but is written as an ordinary algorithm, i.e., **next**() and **hasNext**() functions are not defined explicitly (in Section 2.4 we explain how iterators are used in enumeration algorithms). An enumeration algorithm generates, for a given input x, a sequence $e_1, \ldots, e_m$ of results. Each element e_i is produced by the operation **output** $(\cdot)$. We say that $E(x)$ *enumerates* the set S if, in the execution of $E(x)$, every element of S is produced exactly once. Next, we discuss measures of efficiency for enumeration algorithms.

Polynomial time complexity is not a suitable yardstick of efficiency when analyzing an enumeration algorithm. This is because the size of the output could be much larger (e.g., exponentially larger) than the size of the input. In [11], several definitions of efficiency for enumeration algorithms are discussed. The weakest definition is *polynomial total time,* where the running time is polynomial in the combined size of the input and the output. Two stronger definitions consider the time that is needed for generating the ith element, after the first $i-1$ elements have already been created. *Incremental polynomial time* means that the ith element is generated in time that is polynomial in the combined size of the input and the first $i-1$ elements. The strongest definition is *polynomial delay,* where the ith element is generated in time that is polynomial only in the input. It is easy to see that every algorithm that enumerates with polynomial delay also runs in total time that is linear in the size of the output. For instance, if the delay is $\mathcal{O}(d)$, where d is polynomial in the size of the input, and there are m elements in the result, then the runtime will be $\mathcal{O}(m \cdot d)$.

Note that, in general, SQL queries cannot be evaluated in polynomial total time, i.e., the evaluation may take exponential time if the size of the query is not fixed. For example, the join of a set of relations cannot be evaluated in polynomial total time, since it is NP-complete just to determine whether the result of the join is nonempty [1]. It was shown that full disjunctions can be computed in polynomial total time [12] and even in incremental polynomial time [2].

2.4 Iterators

We use iterators in our enumeration algorithms. An iterator is a programming construct that makes it possible to iterate over the results of an enumeration algorithm, and even pause the computation while retaining the internal state.

Iterators are needed in order to realize polynomial delay when enumeration algorithms use each other. If an enumeration algorithm $\hat{E}$ calls another enumeration algorithm E and then waits until E generates all of its output, the delay could be exponential. Instead, E should halt after generating and returning the first result to $\hat{E}$. Later, E should resume its operation, in order to generate the next result when $\hat{E}$ is ready to get it. Formally, an *iterator* is an object that operates on top of a specific execution of an enumeration algorithm. The iterator outputs each result of the underlying algorithm only upon an explicit **next** request.

Consider an enumeration algorithm E and an input x. An iterator I over $E(x)$ is constructed by the operation

$$I \leftarrow \textbf{new } \mathsf{Iterator}(E, x).$$

Suppose that $E(x)$ enumerates $\{e_1, \ldots, e_m\}$. When executing I.**next**() for the first time, the execution of $E(x)$ starts and continues until the first result e_1 is generated. Specifically, the command **output** (Y) in the execution of $E(x)$ returns both the value of Y and the control of the execution to the procedure that called I.**next**(). Note that the value of Y is returned to the user (i.e., printed) only when **output** (Y) is executed by the outermost procedure. When I.**next**() is executed again, the execution of $E(x)$ is resumed until e_2 is generated and so on. Hence, the elements $e_1, \ldots, e_m$ are enumerated by repeatedly executing m times the command I.**next**(). The cost of executing these m calls to I.**next**() is the total cost of executing $E(x)$.

The operation I.**hasNext**() returns **true** if there are more elements to be generated; otherwise, it returns **false**. Note that I can implement **hasNext**() by actually continuing the execution of $E(x)$ to check if another result is generated. In our algorithms, however, **hasNext**() can be implemented by a simple inspection of the data structures.

3. ACYCLIC SETS OF RELATIONS

Full disjunctions are an associative generalization of binary outerjoins to any number of relations. *Left-deep sequences of outerjoins* (or just *left-deep outerjoins* for short) are also a generalization of outerjoins to any number of relations, but they are not associative. Nevertheless, we will show that in common cases, namely, when the given set of relations is acyclic, full disjunctions can be formulated as left-deep outerjoins. For these cases, a polynomial-delay algorithm is presented in this section. This algorithm forms the basis of our algorithm for the general case (presented in Section 5). We note in passing that [13] showed how to compute full disjunctions for a special case using outerjoins. However, their algorithm did not run with polynomial delay. See Section 1 for details.

We use $R_1 \overset{\circ}{\bowtie} R_2$ to denote the outerjoin of R_1 and R_2. When R_1 and R_2 are connected, the outerjoin and the full disjunction of the two relations are the same. If, however, R_1 and R_2 do not share any attribute and both are nonempty, then the outerjoin is the Cartesian product whereas

```
NLOJ((R_1, ..., R_n))
 1: if n = 1 then
 2:    output all tuples of R_n
 3: else
 4:    S := sc(R_1) ∪ ··· ∪ sc(R_n)
 5:    I := new Iterator(NLOJ, (R_1, ..., R_{n-1}))
 6:    while I.hasNext() do
 7:       t̂ := I.next()
 8:       for all tuples t ∈ R_n, s.t. JC({t, t̂}) do
 9:          mark t
10:          output (embed_S({t, t̂}))
11:       if ∄t ∈ R_n, s.t. JC({t, t̂}) then
12:          output (embed_S(t̂))
13:    for all unmarked tuples t ∈ R_n do
14:       output (embed_S(t))
```

Figure 4: Computing $\overset{\circ}{\bowtie}(R_1, \ldots, R_n)$

the full disjunction is the outerunion of the two relations. We use $\overset{\circ}{\bowtie}(R_1, \ldots, R_n)$ to denote the left-deep outerjoin of $R_1, \ldots, R_n$. Formally, $\overset{\circ}{\bowtie}(R_1, R_2) \stackrel{\text{def}}{=} (R_1 \overset{\circ}{\bowtie} R_2)$, and for $n > 2$, recursively,

$$\overset{\circ}{\bowtie}(R_1, R_2, R_3, \ldots, R_n) \stackrel{\text{def}}{=} \overset{\circ}{\bowtie}(R_1 \overset{\circ}{\bowtie} R_2, R_3, \ldots, R_n).$$

For example,

$$\overset{\circ}{\bowtie}(R_1, R_2, R_3, R_4) \stackrel{\text{def}}{=} ((R_1 \overset{\circ}{\bowtie} R_2) \overset{\circ}{\bowtie} R_3) \overset{\circ}{\bowtie} R_4.$$

Consider a connected set of relations $\mathcal{R} = \{R_1, \ldots, R_n\}$. A *connected-prefix ordering* of $\mathcal{R}$ is an ordering $R_1, \ldots, R_n$ of the relations of $\mathcal{R}$, such that $\{R_1, \ldots, R_i\}$ is connected for all $1 \leq i \leq n$. Note that every connected set of relations has a connected-prefix ordering. Moreover, this ordering can easily be created (e.g., by a DFS traversal of the scheme graph). As an example, one connected-prefix ordering of the relation set $\mathcal{R}_{1a}$ of Figure 1(d) is $R_{\mathrm{c}}, R_{\mathrm{a}}, R_{\mathrm{b}}, R_{\mathrm{d}}, R_{\mathrm{e}}$.

Proposition 3.1 shows that when the scheme graph is a tree, a connected-prefix ordering yields a left-deep outerjoin that is equivalent to the full disjunction. This can be proven using the results of [13].

PROPOSITION 3.1. *Let $\mathcal{R}$ be a connected and acyclic set of relations. If $R_1, \ldots, R_n$ is a connected-prefix ordering of $\mathcal{R}$, then $FD(\mathcal{R}) = \overset{\circ}{\bowtie}(R_1, \ldots, R_n)$.*

We now describe the algorithm NLOJ (Nested-Loop OuterJoin) for computing left-deep outerjoins with polynomial delay. This algorithm is shown in Figure 4. The input to the algorithm NLOJ consists of a tuple $(R_1, \ldots, R_n)$ of relations, where the order of the relations is a connected-prefix ordering. The output is the relation $\overset{\circ}{\bowtie}(R_1, \ldots, R_n)$. Essentially, this algorithm is similar to the nested-loop join, but it also generates tuples that cannot be joined. In order to simplify the presentation, NLOJ is presented as a recursive algorithm. In our implementation of NLOJ (and of all the other algorithms considered in this paper), we used a nonrecursive version, to improve the efficiency.

When the input consists of only one relation (i.e., $n = 1$), all the tuples of that relation are enumerated (Line 2). When $n > 1$, all the tuples of $\overset{\circ}{\bowtie}(R_1, \ldots, R_{n-1})$ are recursively enumerated (Lines 5–14) by the recursive iterator I constructed

```
PDelayFD(R)
1: let R be an arbitrary relation in R
2: Q_{¬R} ← an empty queue
3: for all t ∈ R do
4:    TupExtFD(R, R, t, Q_{¬R})
5: RelExcFD(R, R, Q_{¬R})
```

Figure 5: Computing $FD(\mathcal{R})$

in Line 5. For each enumerated tuple $\hat{t}$, the algorithm iterates, in Line 8, over all the tuples $t \in R_n$ that can be joined with $\hat{t}$. In Lines 9–10, each such tuple t is marked and its natural join with $\hat{t}$ is sent to the output. If no such tuple t exists, then in Line 12 the algorithm outputs $\hat{t}$ after padding it with nulls for the attributes of $\mathsf{sc}(R_n)$ that are missing from $\hat{t}$. Finally, in Lines 13–14, the algorithm outputs all the tuples of R_n that are not marked (i.e., those tuples that are never chosen in Line 8). Each such tuple t is padded with nulls for the attributes of $R_1, \ldots, R_n$ that it does not include.

Note that NLOJ can be viewed as a pipelined computation of binary outerjoins. The iterator I produces the results of the previous stage in the pipeline. It is easy to show that this algorithm has an $\mathcal{O}(N)$ delay, where N is the total number of tuples in the relations. Furthermore, Line 8 forms the bottleneck of the computation. Thus, by using standard join techniques (e.g., ones that are based on indices), the delay can be substantially reduced. Such techniques are beyond the scope of this paper.

From Proposition 3.1, we conclude that NLOJ can be used for computing full disjunctions of acyclic and connected sets of relations. If the given set of relations is not connected, then we can separately compute the full disjunction of each connected component and pad each generated tuple with nulls for all the attributes of the other connected components. In other words, we compute the full disjunctions of the different components and then compute the outerunion of these full disjunctions. Given that N is the total number of tuples in the relations of $\mathcal{R}$, the delay of NLOJ is $\mathcal{O}(N)$ as formally stated in the following theorem.

THEOREM 3.2. *Given an acyclic set of relations $\mathcal{R}$ containing N tuples altogether, the algorithm* NLOJ *computes $FD(\mathcal{R})$ with $\mathcal{O}(N)$ delay.*

For some cyclic sets of relations, the full disjunction is still equivalent to a left-deep sequence of outerjoins; however, characterizing these cases is beyond the scope of this paper. In such cases, the full disjunction can be computed by NLOJ. In Section 5, we use NLOJ as part of an algorithm for computing full disjunctions in the general case.

4. GENERAL SETS OF RELATIONS

In this section, we present PDELAYFD which is the first algorithm that computes full disjunctions of general sets of relations with polynomial delay. By definition, having polynomial delay guarantees that the total time of this algorithm is linear in the size of the output. This improves the best known upper bound [2] for computing full disjunctions which is quadratic in the size of the output.

```
TupExtFD(R, R, t, Q_{¬R})
 1: C_t := ∅
 2: Q_t := {ExtendToMax(R, {t})}
 3: S := ⋃_{R'∈R} sc(R')
 4: while Q_t is not empty do
 5:    remove the top element T from Q_t
 6:    output (embed_S(T))
 7:    C_t.insert(T)
 8:    for all s ∈ R \ {R} do
 9:       let T_s be the maximal tuple set, s.t. s ∈ T_s,
          T_s ⊆ T ∪ {s} and JCC(T_s)
10:       T̂ := ExtendToMax(R, T_s)
11:       if t ∈ T̂ and T̂ ∉ C_t ∪ Q_t then
12:          Q_t.insert(T̂)
13:       if T̂ ∩ R = ∅ and T̂ ∉ Q_{¬R} then
14:          Q_{¬R}.insert(T̂)
```

Figure 6: Computing $FD_{|t}(\mathcal{R})$

```
ExtendToMax(R, T)
 1: T̂ := T
 2: for all R ∈ R do
 3:    if R ∩ T̂ = ∅ then visited[R]:=false
 4:    else visited[R]:=true
 5: while there is R ∈ R, s.t. visited[R]=false and
    R and T̂ share a common attribute do
 6:    visited[R]:=true
 7:    for all t ∈ R do
 8:       if JCC(T̂ ∪ {t}) then
 9:          T̂ := T̂ ∪ {t}
10:          break
11: return T̂
```

Figure 7: Extending a tuple set

Consider a relation $R \in \mathcal{R}$. The set comprising all tuple sets in $MaxJCC(\mathcal{R})$ that contain a given tuple $t \in R$ is denoted by $MaxJCC_{|t}(\mathcal{R})$. The subset of $FD(\mathcal{R})$ that corresponds to $MaxJCC_{|t}(\mathcal{R})$ is denoted by $FD_{|t}(\mathcal{R})$; that is, $FD_{|t}(\mathcal{R})$ is the relation $\{embed_S(T) \mid T \in MaxJCC_{|t}(\mathcal{R})\}$, where S is the union of all the schemes of the relations of $\mathcal{R}$. The set consisting of all tuple sets in $MaxJCC(\mathcal{R})$ that contain no tuple from R is denoted by $MaxJCC_{|\neg R}(\mathcal{R})$. The corresponding subset of $FD(\mathcal{R})$ is denoted by $FD_{|\neg R}(\mathcal{R})$. Clearly, $FD(\mathcal{R}) = FD_{|\neg R}(\mathcal{R}) \cup \bigcup_{t \in R} FD_{|t}(\mathcal{R})$.

We compute $FD(\mathcal{R})$ in the following manner. We first choose a relation $R \in \mathcal{R}$. We then enumerate $FD_{|t}(\mathcal{R})$ for all tuples $t \in R$. Finally, we enumerate $FD_{|\neg R}(\mathcal{R})$. This is basically the algorithm PDelayFD of Figure 5. To explain this algorithm, we examine the algorithms that it calls.

We first describe the algorithm TupExtFD of Figure 6. The input of this algorithm consists of a set $\mathcal{R}$ of relations, a relation $R \in \mathcal{R}$, a tuple $t \in R$ and a queue $\mathcal{Q}_{\neg R}$. This algorithm computes $FD_{|t}(\mathcal{R})$ with polynomial delay. It uses (in Lines 2 and 10) the algorithm ExtendToMax of Figure 7 that, given a set $\mathcal{R}$ of relations and a join consistent and connected set T of tuples in $\mathcal{R}$, returns an arbitrary maximal set $\hat{T}$ of tuples in $\mathcal{R}$ that satisfy $JCC(\hat{T})$ and $T \subseteq \hat{T}$. In particular, if T is a singleton $\{t\}$, then ExtendToMax$(\mathcal{R}, T)$ returns an arbitrary tuple set of $MaxJCC_{|t}(\mathcal{R})$.

The algorithm ExtendToMax works as follows. It starts with T as the arbitrary set $\hat{T}$. It marks as visited all the relations of $\mathcal{R}$ that contain a tuple of T (Lines 2–4). Then, it iteratively extends $\hat{T}$ while keeping $\hat{T}$ connected and join consistent (Lines 5–10). In each iteration, a tuple is taken from a relation that has not yet been visited (hence, $\hat{T}$ never includes two tuples from the same relation). The tuple is chosen arbitrarily from a relation that has a shared attribute with $\hat{T}$, since tuples of other relations are not connected to $\hat{T}$. Only tuples that are connected and join consistent with $\hat{T}$ are added to $\hat{T}$ (Lines 8–9). The iterations continue until $\hat{T}$ cannot be extended anymore.

We continue now with the description of TupExtFD. The algorithm TupExtFD uses a repository $\mathcal{C}_t$ for storing tuple sets that have been printed and a queue $\mathcal{Q}_t$ for storing tuple sets awaiting to be printed. Initially, $\mathcal{C}_t$ is empty and $\mathcal{Q}_t$ is a singleton containing one tuple set of $MaxJCC_{|t}(\mathcal{R})$. Lines 5–14 are then executed repeatedly, until $\mathcal{Q}_t$ is empty. In Lines 5–7, a tuple set $T \in \mathcal{Q}_t$ is moved from $\mathcal{Q}_t$ to $\mathcal{C}_t$, transformed into the corresponding tuple of $FD(\mathcal{R})$ (by joining and adding nulls) and printed. Then, Lines 9–14 are repeatedly executed for each tuple s of some relation in $\mathcal{R}\setminus\{R\}$ (by a slight abuse of notation, we write $s \in \mathcal{R} \setminus \{R\}$). In these lines, the algorithm tries to add the tuple s to the set T. Since $JCC(T \cup \{s\})$ does not necessarily hold, the maximal among all subsets of $T' \subseteq T \cup \{s\}$ satisfying $s \in T'$ and $JCC(T')$ is produced. Let T_s denote this set. It is easy to show that T_s is unique.[3] In Line 10, T_s is extended to a maximal tuple set $\hat{T} \in MaxJCC(\mathcal{R})$ by executing ExtendToMax$(\mathcal{R}, T_s)$. In Lines 11–12, the algorithm tests whether t is contained in $\hat{T}$. If so, then $\hat{T}$ is added to $\mathcal{Q}_t$, provided that $\hat{T}$ is in neither $\mathcal{C}_t$ nor $\mathcal{Q}_t$. Lines 13–14 are used for enumerating $FD_{|\neg R}(\mathcal{R})$ and are discussed in detail later.

By now, we have described how to enumerate $FD_{|t}(\mathcal{R})$ with polynomial delay. Thus, $FD(\mathcal{R})$ can be enumerated with polynomial delay if $FD_{|\neg R}(\mathcal{R})$ can be likewise enumerated. However, the following proposition indicates that this is not likely to be true (and one has to expect an exponential first delay when computing $FD_{|\neg R}(\mathcal{R})$).

Proposition 4.1. *Testing whether $FD_{|\neg R}(\mathcal{R}) \neq \emptyset$ is NP-complete.*

To overcome this problem, we use the fact that the relation $FD_{|\neg R}(\mathcal{R})$ is computed only after generating the relations $FD_{|t}(\mathcal{R})$ for all $t \in R$. It turns out that we can *collect* enough information, during the enumeration of all the $FD_{|t}(\mathcal{R})$, in order to enumerate $FD_{|\neg R}(\mathcal{R})$ with polynomial delay. In particular, in Lines 13–14 of TupExtFD, we test whether the maximal tuple set $\hat{T}$ is in $MaxJCC_{|\neg R}(\mathcal{R})$. If so, then we store $\hat{T}$ in a queue $\mathcal{Q}_{\neg R}$. Note that in the algorithm PDelayFD, all executions of TupExtFD use the same $\mathcal{Q}_{\neg R}$ and, furthermore, $\mathcal{Q}_{\neg R}$ is given by reference. Finally, $\mathcal{Q}_{\neg R}$ (that contains all the collected tuple sets) is used

[3]The tuple set T_s can be obtained from $T \cup \{s\}$ as follows. First, remove all the tuples t' such that $\{s, t'\}$ is not join consistent (i.e., for some attribute A, either $s[A] \neq t'[A]$ or $s[A] = t'[A] = \perp$). In particular, t' is removed if it is from the same relation as s. Then, remove tuples that are no longer in the same connected component as s.

RelExcFD($\mathcal{R}, R, \mathcal{Q}_{\neg R}$)

1: $\mathcal{C}_{\neg R} := \emptyset$
2: $S := \bigcup_{R' \in \mathcal{R}} \mathsf{sc}(R')$
3: **while** $\mathcal{Q}_{\neg R}$ is not empty **do**
4: remove the top element T from $\mathcal{Q}_{\neg R}$
5: **output** ($embed_S(T)$)
6: $\mathcal{C}_{\neg R}$.**insert**(T)
7: **for all** $s \in \mathcal{R} \setminus \{R\}$ **do**
8: let T_s be the maximal tuple set, s.t. $s \in T_s$, $T_s \subseteq T \cup \{s\}$ and $JCC(T_s)$
9: $\hat{T}$:=ExtendToMax($\mathcal{R}, T_s$)
10: **if** $\hat{T} \cap R = \emptyset$ and $\hat{T} \notin \mathcal{Q}_{\neg R} \cup \mathcal{C}_{\neg R}$ **then**
11: $\mathcal{Q}_{\neg R}$.**insert**($\hat{T}$)

Figure 8: Computing $FD_{|\neg R}(\mathcal{R})$

for enumerating $FD_{|\neg R}(\mathcal{R})$ with polynomial delay by applying the algorithm RelExcFD (shown in Figure 8).

The algorithm RelExcFD is very similar to TupExtFD, except for the following small differences. RelExcFD extends tuples of $\mathcal{Q}_{\neg R}$ (instead of tuples of $\mathcal{Q}_t$). Note that RelExcFD does not use $\mathcal{Q}_t$ at all, since the tuples of all the $FD_{|t}(\mathcal{R})$ have already been printed. RelExcFD uses $\mathcal{C}_{\neg R}$ to store tuple sets that have already been removed from $\mathcal{Q}_{\neg R}$ and printed. In comparison, TupExtFD does not remove any element from $\mathcal{Q}_{\neg R}$ and, so, it does not need $\mathcal{C}_{\neg R}$. Tuple sets $\hat{T}$, which are created by extending T_s in Line 9, are inserted into $\mathcal{Q}_{\neg R}$ only if they do not contain any tuple of R and are in neither $\mathcal{Q}_{\neg R}$ nor $\mathcal{C}_{\neg R}$.

Next, we prove the correctness of the algorithm PDelayFD. For that, we use the following lemma.

Lemma 4.2. *Consider a set of relations $\mathcal{R}$ and a relation $R \in \mathcal{R}$.*

1. *For a tuple $t \in R$ and a queue $\mathcal{Q}_{\neg R} \subseteq MaxJCC_{|\neg R}(\mathcal{R})$,* TupExtFD($\mathcal{R}, R, t, \mathcal{Q}_{\neg R}$) *enumerates $FD_{|t}(\mathcal{R})$.*

2. *Suppose that $\mathcal{Q}_{\neg R}$ is initially an empty queue. After executing* TupExtFD($\mathcal{R}, R, t, \mathcal{Q}_{\neg R}$) *for all tuples $t \in R$,* RelExcFD($\mathcal{R}, R, \mathcal{Q}_{\neg R}$) *enumerates $FD_{|\neg R}(\mathcal{R})$.*

Proof. We will first prove Claim 1. Consider an execution of the algorithm TupExtFD($\mathcal{R}, R, t, \mathcal{Q}_{\neg R}$). Lines 10–12 of the algorithm guarantee that only tuples of $FD_{|t}(\mathcal{R})$ are enumerated and that each tuple is printed at most once. It remains to prove that every tuple of $FD_{|t}(\mathcal{R})$ is printed.

Suppose, by way of contradiction, that there is some $T' \in MaxJCC_{|t}(\mathcal{R})$, such that $embed_S(T')$ is not printed by the algorithm. Clearly, the algorithm prints at least one tuple set of $MaxJCC_{|t}(\mathcal{R})$. Let $T \in MaxJCC_{|t}(\mathcal{R})$ be a tuple set, among all the printed ones, that yields a maximal T_m, such that *(1)* $t \in T_\mathsf{m}$ *(2)* $T_\mathsf{m} \subseteq T' \cap T$, *and (3)* $JCC(T_\mathsf{m})$. Note that T_m could be the set $\{t\}$. Since T' is connected, there exists a tuple $s \in T' \setminus T_\mathsf{m}$, such that $T_\mathsf{m} \cup \{s\}$ is connected. Consider the iteration of Lines 5–14 in which T is printed. Let T_s be the tuple set that is generated in Line 9 when the tuple s is chosen in Line 8. Since T_s is maximal and $JCC(T_\mathsf{m} \cup \{s\})$, it follows that $T_\mathsf{m} \cup \{s\} \subseteq T_s$. Thus, $T_\mathsf{m} \cup \{s\}$ is contained in the tuple set $\hat{T} \in MaxJCC(\mathcal{R})$ that is generated in Line 10. Since $t \in \hat{T}$, one of the following three options must hold: (1) $\hat{T}$ is inserted into $\mathcal{Q}_t$, (2) $\hat{T}$ is in $\mathcal{Q}_t$, or (3) $\hat{T}$ was in $\mathcal{Q}_t$ in the past (and is now in $\mathcal{C}_t$). Since the algorithm prints every tuple set that is inserted into $\mathcal{Q}_t$, the existence of $\hat{T}$ contradicts the choice of T and, hence, the existence of T'.

The proof of Claim 2 is similar, except for the following differences. First, we assume that $T' \in FD_{\neg R}(\mathcal{R})$. Second, among all the tuple sets of $MaxJCC(\mathcal{R})$ that are printed either in one of the executions of TupExtFD or in the execution of RelExcFD, we choose a tuple set T that yields a maximal T_m, such that *(1)* $T_\mathsf{m} \subseteq T' \cap T$, *and (2)* $JCC(T_\mathsf{m})$. Third, the set $\hat{T}$ is generated in either Line 10 of TupExtFD or Line 9 of RelExcFD. Note that a contradiction is obtained immediately if $\hat{T} \in FD_{\neg R}(\mathcal{R})$. Otherwise, we use the correctness of Claim 1. □

Correctness of the algorithm PDelayFD follows immediately from Lemma 4.2 and is formally stated in the next theorem. This theorem also analyzes the running time of PDelayFD. We assume that $\mathcal{Q}_t$, $\mathcal{C}_t$, $\mathcal{Q}_{\neg R}$ and $\mathcal{C}_{\neg R}$ provide logarithmic access time; that is, insertion, removal and membership test require $\mathcal{O}(\log k)$ comparisons, where k is the number of elements in the repository. For simplicity, our analysis assumes that the relations of $\mathcal{R}$ have a bounded number of columns. Therefore, operations such as testing join consistency of two tuples can be done in constant time. We use n to denote the number of relations in $\mathcal{R}$ and N to denote the total number of tuples in the relations of $\mathcal{R}$.

Theorem 4.3. PDelayFD($\mathcal{R}$) *computes $FD(\mathcal{R})$ with an $\mathcal{O}(n^2N + N^2 + nN \log N)$ delay.*

Theorem 4.3 states that PDelayFD computes full disjunctions with polynomial delay and in total time that is linear in the size of the output, for any arbitrary set of relations. In both these aspects, PDelayFD is superior to the state of the art algorithm [2].

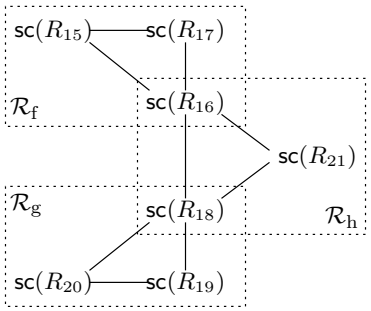
5. THE MAIN ALGORITHM

The algorithm NLOJ for computing left-deep outerjoins is more efficient than PDelayFD, since its delay is linear compared with the quadratic delay of PDelayFD. However, NLOJ only applies to acyclic sets of relations. In fact, in [13], it was shown that there are full disjunctions that cannot be formulated as any sequence of outerjoins (i.e., with arbitrary placement of parentheses). In this section, we show that the problem of computing full disjunctions can be decomposed into smaller subproblems, such that PDelayFD is needed only for solving subproblems and NLOJ can compute the full disjunction using the solutions of the subproblems. For now, we assume that there are no null values in the input relations. Toward the end of the section, we explain why this assumption is needed and we provide a simple solution for the case of input relations with null values.

Let $\mathcal{R}$ be a set of relations. A subset $\mathcal{R}_1$ of $\mathcal{R}$ is said to be *biconnected* if for each pair of relations $R_1, R_2 \in \mathcal{R}_1$, the scheme graph of $\mathcal{R}$ has (at least) two node-disjoint paths connecting $\mathsf{sc}(R_1)$ and $\mathsf{sc}(R_2)$. A *biconnected component* of $\mathcal{R}$ is a biconnected subset $\mathcal{R}_\mathrm{m}$ that is maximal w.r.t. containment (that is, no biconnected subset of $\mathcal{R}$ properly contains $\mathcal{R}_\mathrm{m}$). Note that a relation can belong to more than one biconnected component (such a relation is an *articulation point*). Also note that a biconnected component may contain only one relation. For instance, if $\mathcal{R}$ is acyclic, then all

rel.	attributes
R_{15}	A, B, C
R_{16}	A, D, E
R_{17}	A, B, F
R_{18}	E, G, H, I
R_{19}	H, J
R_{20}	I, J, K
R_{21}	D, G

(a) Schemes of $\mathcal{R}_4$



(b) Scheme graph $\mathcal{G}_{\mathsf{sc}}(\mathcal{R}_4)$

Figure 9: Schemes of a relation set $\mathcal{R}_4$

the biconnected components of $\mathcal{R}$ are singletons. It is known that biconnected components can be efficiently constructed by detecting articulation points [3].

EXAMPLE 5.1. The biconnected components of $\mathcal{R}_1$ (Figure 1(a)), surrounded by dotted polygons in the scheme graph of Figure 1(b), are the following: $\mathcal{R}_{\rm a} = \{R_1, R_2, R_3\}$, $\mathcal{R}_{\rm b} = \{R_3, R_4, R_5\}$, $\mathcal{R}_{\rm c} = \{R_6\}$, $\mathcal{R}_{\rm d} = \{R_7, R_8, R_9\}$ and $\mathcal{R}_{\rm e} = \{R_{10}\}$. □

Consider a connected set $\mathcal{R}$ of relations with the biconnected components $\mathcal{R}_1, \ldots, \mathcal{R}_k$. We say that the sequence $\mathcal{R}_1, \ldots, \mathcal{R}_k$ is a *strong connected-prefix order* (abbr. *SCP order*) if for all $2 \leq i \leq k$, the following two conditions hold. First, $\mathcal{R}_i$ has an attribute that appears in $\mathcal{R}_1 \cup \cdots \cup \mathcal{R}_{i-1}$. Second, if $\mathcal{R}_j \cap (\mathcal{R}_1 \cup \cdots \cup \mathcal{R}_{i-1}) \neq \emptyset$ for some $i \leq j \leq n$, then $\mathcal{R}_i \cap (\mathcal{R}_1 \cup \cdots \cup \mathcal{R}_{i-1}) \neq \emptyset$. It is straightforward to show that an SCP order of the biconnected components exists (provided that $\mathcal{R}$ is connected) and can be obtained efficiently by the following procedure. First, choose an arbitrary connected component $\mathcal{R}_1$. Having chosen $\mathcal{R}_1, \ldots, \mathcal{R}_{i-1}$ (and $i \leq k$), check whether one of the remaining components contains a relation R that is in $\mathcal{R}_1 \cup \cdots \cup \mathcal{R}_{i-1}$. If so, choose $\mathcal{R}_i$ to be such a component. Otherwise, choose $\mathcal{R}_i$ to be a component that has an attribute A that appears in $\mathcal{R}_1 \cup \cdots \cup \mathcal{R}_{i-1}$. In Example 5.1, $\mathcal{R}_{\rm a}, \mathcal{R}_{\rm b}, \mathcal{R}_{\rm c}, \mathcal{R}_{\rm d}, \mathcal{R}_{\rm e}$ is an SCP order for the biconnected components of $\mathcal{R}_1$. $\mathcal{R}_{\rm a}, \mathcal{R}_{\rm c}, \mathcal{R}_{\rm b}, \mathcal{R}_{\rm d}, \mathcal{R}_{\rm e}$ is not an SCP order, since among $\mathcal{R}_{\rm b}$ and $\mathcal{R}_{\rm c}$, only $\mathcal{R}_{\rm b}$ contains a relation of $\mathcal{R}_{\rm a}$.

The next theorem shows that the full disjunction of $\mathcal{R}$ can be obtained by first computing the full disjunction of each biconnected component $\mathcal{R}_i$ and then applying a left-deep sequence of outerjoins in an SCP order. Note that if $\mathcal{R}_i$ is a singleton $\{R\}$, then $FD(\mathcal{R}_i)$ is actually R itself.

THEOREM 5.2. *Suppose that $\mathcal{R}$ is a connected set of relations and $\mathcal{R}_1, \ldots, \mathcal{R}_k$ are the biconnected components of $\mathcal{R}$ in an SCP order. Then, $FD(\mathcal{R}) = \overset{\circ}{\bowtie}(FD(\mathcal{R}_1), \ldots, FD(\mathcal{R}_k))$.*

Note that in Theorem 5.2, the scheme graph of the relations $\{FD(\mathcal{R}_1,), \ldots, FD(\mathcal{R}_k)\}$ is not necessarily acyclic. Thus, the proof is different from that of Proposition 3.1 and more intricate.

EXAMPLE 5.3. Figure 9 depicts the scheme of the relation set $\mathcal{R}_4 = \{R_{15}, \ldots, R_{21}\}$. The biconnected components of $\mathcal{R}_4$, surrounded by dotted rectangles in Figure 9(b), are the sets $\mathcal{R}_{\rm f}$, $\mathcal{R}_{\rm g}$, and $\mathcal{R}_{\rm h}$. It is easy to see that the sequence $\mathcal{R}_{\rm f}, \mathcal{R}_{\rm h}, \mathcal{R}_{\rm g}$ is an SCP order of the three biconnected components. It follows from Theorem 5.2 that $FD(\mathcal{R}_4) = \overset{\circ}{\bowtie}(FD(\mathcal{R}_{\rm f}), FD(\mathcal{R}_{\rm h}), FD(\mathcal{R}_{\rm g}))$. In comparison, $\mathcal{R}_{\rm f}, \mathcal{R}_{\rm g}, \mathcal{R}_{\rm h}$ is not in an SCP order, since $\mathcal{R}_{\rm h}$ shares a relation with $\mathcal{R}_{\rm f}$ (the relation R_{16}) while $\mathcal{R}_{\rm g}$ does not share a relation with $\mathcal{R}_{\rm f}$. In this specific case, it can also be shown that $FD(\mathcal{R}_4)$ is not equivalent to $\overset{\circ}{\bowtie}(FD(\mathcal{R}_{\rm f}), FD(\mathcal{R}_{\rm g}), FD(\mathcal{R}_{\rm h}))$. That is, there are relations $R_{15}, \ldots, R_{21}$ for which $FD(\mathcal{R}_4) \neq \overset{\circ}{\bowtie}(FD(\mathcal{R}_{\rm f}), FD(\mathcal{R}_{\rm g}), FD(\mathcal{R}_{\rm h}))$. □

It is important to emphasize that an SCP order is not a necessary condition for equivalence between $FD(\mathcal{R})$ and $\overset{\circ}{\bowtie}(FD(\mathcal{R}_1), \ldots, FD(\mathcal{R}_k))$. It is only a sufficient condition for having an equality between the two expressions.

Theorem 5.2 implies the following three-step algorithm for computing the full disjunction of a connected set of relations $\mathcal{R}$. First, find the biconnected components of $\mathcal{R}$ and obtain an SCP order $\mathcal{R}_1, \ldots, \mathcal{R}_k$. Second, compute the full disjunctions $FD(\mathcal{R}_1), \ldots, FD(\mathcal{R}_k)$. Third, compute $\overset{\circ}{\bowtie}(FD(\mathcal{R}_1), \ldots, FD(\mathcal{R}_k))$ and return it as the answer. This algorithm enumerates, in polynomial total time, the full disjunction of a connected set of relations.

In the above algorithm, the full disjunctions of the biconnected components are completely generated before applying the left-deep outerjoins. The size of any relation $FD(\mathcal{R}_i)$ can be exponential in the size of the original relations and, therefore, the delay of the whole algorithm is not polynomial—an exponential time is needed for computing intermediate full disjunctions that are the input to NLOJ, and the delay of NLOJ is linear in the size of its input. Next, we show how the algorithm presented above can be modified to enumerate full disjunctions with polynomial delay.

Consider a connected set $\mathcal{R}$ of relations and let $\mathcal{R}_1, \ldots, \mathcal{R}_k$ be the biconnected components of $\mathcal{R}$ in an SCP order. For $i = 2, \ldots, n$, the *connecting relation* of $\mathcal{R}_i$ is defined as follows. If $\mathcal{R}_i$ contains a relation R that appears in $\mathcal{R}_1 \cup \cdots \cup \mathcal{R}_{i-1}$, then the connecting relation is R (note that in this case, R is an articulation point); otherwise, the connecting relation is a relation that contains an attribute A that appears in $\mathcal{R}_1 \cup \cdots \cup \mathcal{R}_{i-1}$. It can be shown that in both cases, the connecting relation is unique.

We now describe how to enumerate the full disjunction of a set of relations $\mathcal{R}$, given that $\mathcal{R}_1, \ldots, \mathcal{R}_k$ is a sequence of the biconnected components of $\mathcal{R}$ in an SCP order. According to Theorem 5.2, it is sufficient to enumerate the expression $\overset{\circ}{\bowtie}(FD(\mathcal{R}_1), \ldots, FD(\mathcal{R}_k))$. This is done in the algorithm BICOMNLOJ of Figure 10. BICOMNLOJ can be seen as an advanced version of NLOJ. The main difference between BICOMNLOJ and NLOJ is that while NLOJ enumerates the left-deep outerjoin of the relations of the input, BICOMNLOJ enumerates a left-deep outerjoin of relations that are full disjunctions of biconnected components. Two major changes are necessary due to the difference between the algorithms. First, given a tuple $\hat{t}$, we cannot enumerate $FD(\mathcal{R}_k)$ with polynomial delay by simply looking for tuples that can be joined with $\hat{t}$, because of the following reason. The size of $FD(\mathcal{R}_k)$ could be exponential in the size of the input and, hence, the overall delay would not be polynomial. The second problem is that of efficiently enumerating the tuples of $FD(\mathcal{R}_k)$ that cannot be joined with any tuple of $\overset{\circ}{\bowtie}(FD(\mathcal{R}_1), \ldots, FD(\mathcal{R}_{k-1}))$.

The solution to the first problem stems from the following property. Let R be the connecting relation of $\mathcal{R}_k$. A tuple

$\hat{t} \in \overset{\circ}{\bowtie}(FD(\mathcal{R}_1), \ldots, FD(\mathcal{R}_{k-1}))$ can be joined with some tuple $t' \in FD(\mathcal{R}_k)$ if and only if there is a tuple $t \in R$, such that (1) t is join consistent with $\hat{t}$ and (2) $t' \in FD_{|t}(\mathcal{R}_k)$. Thus, we need to find all the tuples $t \in R$ that are join consistent with $\hat{t}$ and enumerate $FD_{|t}(\mathcal{R}_k)$. This is done using an iterator over TupExtFD (Lines 12–14).

We now address the second problem. There are two types of tuples of $FD(\mathcal{R}_k)$ that cannot be joined with any tuple $\hat{t} \in \overset{\circ}{\bowtie}(FD(\mathcal{R}_1), \ldots, FD(\mathcal{R}_{k-1}))$. One type consists of the tuples in the relations $FD_{|t}(\mathcal{R}_k)$, where t is a tuple of R that is not join consistent with any tuple $\hat{t}$. The second type includes all the tuples of $FD_{|\neg R}(\mathcal{R}_k)$. For enumerating tuples of the first type, we mark (in Line 11) the tuples of R that are chosen in Line 10 (thus, we are interested in the unmarked tuples). For the second type, we use an iterator over RelExcFD (Lines 21-23). Note that RelExcFD is executed only after TupExtFD is applied to all the tuples of R.

If $\mathcal{R}_k$ is a singleton biconnected component $\{R\}$, the computation can be done similarly to the way it was done in NLOJ, which is simpler and more efficient. However, I (in Line 7 of Figure 10) still has to be an iterator of BiComNLOJ rather than of NLOJ, since recursive calls may encounter biconnected components that are not necessarily singletons. Nonetheless, when $\mathcal{R}$ is acyclic, the computation becomes identical to that of NLOJ.

The complexity of BiComNLOJ is considered in the following theorem. We use n_i to denote the number of relations in the biconnected component $\mathcal{R}_i$ and N_i to denote the total number of tuples in $\mathcal{R}_i$.

Theorem 5.4. *Consider a connected set of relations $\mathcal{R}$. If $\mathcal{R}_1, \ldots, \mathcal{R}_k$ are the biconnected components of $\mathcal{R}$ in an SCP order, then* BiComNLOJ$(\mathcal{R}_1, \ldots, \mathcal{R}_k)$ *enumerates the full disjunction $FD(\mathcal{R})$. The delay of the enumeration is $\mathcal{O}(\sum_{i=1}^{k} M_i)$, where $M_i = N_i$ if $\mathcal{R}_i$ is a singleton component and $M_i = n_i^2 N_i + N_i^2 + n_i N_i \log N_i$ otherwise.*

Theorem 5.4 shows that BiComNLOJ runs with polynomial delay, thereby implying a total time that is linear in the size of the output. Observe that when $k > 1$, the delay of BiComNLOJ is shorter than that of PDelayFD, due to our strategy of decomposing the scheme graph into biconnected components.

In Theorem 5.4, we assume that the set $\mathcal{R}$ is connected. Recall that when $\mathcal{R}$ is not connected, the full disjunction of $\mathcal{R}$ is simply an outerunion of the full disjunctions of the connected components of $\mathcal{R}$.

We now return to the problem of null values in the input relations. First, we emphasize that all our algorithms, except for BiComNLOJ, work correctly for relations that contain null values. The problem with BiComNLOJ is illustrated by the following example. Consider two biconnected components $\mathcal{R}_1$ and $\mathcal{R}_2$. Suppose that the intersection of $\mathcal{R}_1$ and $\mathcal{R}_2$ is the relation (articulation point) R and, moreover, R has an attribute A that appears in no other relation of either $\mathcal{R}_1$ or $\mathcal{R}_2$. Also, we assume that R contains a single tuple t that has a null in A. Let $\hat{T}_1$ and $\hat{T}_2$ be tuple sets of $\mathcal{R}_1$ and $\mathcal{R}_2$, respectively, such that t is contained in both $\hat{T}_1$ and $\hat{T}_2$. Because $\hat{T}_1$ and $\hat{T}_2$ are join consistent with the tuple of the shared relation, they should be joined. However, since there is a null in A, the tuples $embed_{S_1}(\hat{T}_1)$ and $embed_{S_2}(\hat{T}_2)$ (where S_1 and S_2 are the schemes of $\mathcal{R}_1$ and $\mathcal{R}_2$, respectively) are not join consistent.

```
BiComNLOJ((R_1, ..., R_k))
 1: if k = 1 then
 2:   PDelayFD(R_1)
 3: else
 4:   S := ⋃_{R' ∈ (R_1 ∪ R_2 ∪ ... R_k)} sc(R')
 5:   let R be the connecting relation of R_k
 6:   Q_{¬R} := ∅
 7:   I := new Iterator(BiComNLOJ, (R_1, ..., R_{k-1}))
 8:   while I.hasNext() do
 9:     t̂ := I.next()
10:     for all tuples t ∈ R, s.t. JCC({t, t̂}) do
11:       mark t
12:       I_t := new Iterator
            (TupExtFD, (R_k, R, t, Q_{¬R}))
13:       while I_t.hasNext() do
14:         output (embed_S({I_t.next(), t̂}))
15:     if ∄t ∈ R, s.t. JC({t, t̂}) then
16:       output (embed_S(t̂))
17:   for all unmarked tuples t ∈ R do
18:     I_t := new Iterator(TupExtFD, (R_k, R, t, Q_{¬R}))
19:     while I_t.hasNext() do
20:       output (embed_S(I_t.next()))
21:   I_{¬R} := new Iterator(RelExcFD, (R_k, R, t, Q_{¬R}))
22:   while I_{¬R}.hasNext() do
23:     output (embed_S(I_{¬R}.next()))
```

Figure 10: Main algorithm for computing $FD(\mathcal{R})$

There are two possible solutions to the problem of null values in the the input relations. One solution is to change the algorithms so that they work with tuple sets—actual tuples of the full disjunction are generated just before printing them. A second solution is to slightly modify the functions $JC()$ and $JCC()$, used in lines 10 and 15 of BiComNLOJ, so that embeddings of tuple sets that have a tuple in common will be considered join consistent even when the common tuple contains null values. Note that this requires some bookkeeping to indicate whether embeddings were generated from tuple sets that included tuples from articulation points.

6. FULL DISJUNCTIONS AND SQL

In this section, we discuss in general lines the incorporation of full disjunctions into query plans for SQL. By using the algorithms presented in this paper, the full-disjunction operator can be incorporated into query plans as a fully pipelined component. Therefore, in many cases, full disjunctions need not be stored in temporary relations. Next, we discuss how to optimize queries that involve full disjunctions as well as additional operators. The query of Example 1.1, for instance, also involves a projection, a selection and sorting.

We start by considering the `ORDER BY` operator. Suppose that in a given query, this operator is applied to only one attribute A (e.g., as in the query of Example 1.1). The algorithm PDelayFD can be modified to enumerate the full disjunction in a sorted order, without increasing the delay. This requires starting with the relations whose schema contain A. Tuples from these relations are then taken ordered according to A and extended as being done in PDelayFD. This can be generalized to enumerating full disjunctions in ranked

order according to any ranking function that is "monotonically c-determined", as defined in [2]. (Intuitively, a ranking function is monotonically c-determined if for each tuple t in the full disjunction it is possible to determine the relative ranking of t by looking at a fixed number c of tuples among the tuples that are joined to generate t.)

It follows that if the `ORDER BY` operator is applied to a bounded number of attributes, such that the relations containing these attributes form a clique in the scheme graph, then the tuples of the full disjunction can be enumerated with polynomial delay in sorted order. The algorithm BiComNLOJ can also be modified to enumerate the tuples in sorted order (under the same conditions described above). For that, all the attributes of the `ORDER BY` clause should be in a single biconnected component; however, if this is not the case, then we can combine all the biconnected components that contain these attributes and treat them as one. We leave further investigation of these to future work.

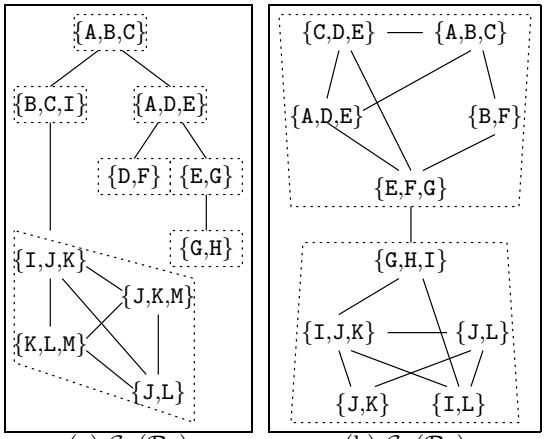
As for the *selection* operator, the situation is more complicated. It is NP-complete to test whether the result of selecting tuples from a full disjunction is not empty, even if the condition consists of only two equalities [12]. Intuitively, if we naively delete tuples that do not satisfy the condition of a selection prior to computing the full disjunction, we might end up with partial information that would not be there otherwise, i.e., would be joined with tuples that do not satisfy the condition of the selection. This means that, in general, selections cannot be pushed through a full disjunction.

In the special case that a selection with a single attribute is applied to a full disjunction, it is possible to compute the selection of a full disjunction, while retaining polynomial delay. Intuitively, this can be achieved by enumerating the result in a special order. We can push a selection though a full disjunction by applying a sorted enumeration that terminates after a certain point. Suppose, for example, that the selection consists of a condition $cond(A)$ that involves a single attribute A. We can enumerate the full disjunction in a sorted order according to A, where the order is defined so that values X satisfying $cond(X)$ precede all other values. Enumeration terminates when the first tuple that does not satisfy the condition is generated.

In general, it is not possible to compute the result of a *projection*[4] of a full disjunction in polynomial delay, or even polynomial total time, unless P≠NP [12]. However, it is possible to improve the naive method of first completely computing a full disjunction, and then applying a projection, by pushing the projection through the full disjunction, similarly to pushing it through a natural join. In other words, attributes that do not appear in the query and are not used for joining relations can be projected out early. Moreover, if all the projection and selection attributes belong to a single biconnected component, it is sufficient to compute the full disjunction of this component. In general, given a set of projection (and selection) attributes we only need to compute the full disjunction w.r.t. components that contain at least one relation that is on a simple path, in the scheme graph, between two relations that contain projection or selection attributes.

7. EXPERIMENTATION

[4]We consider only projection that discards duplications, i.e., when `distinct` is used. Dealing with projection that retains duplications is beyond the scope of this discussion.



(a) $\mathcal{G}_{sc}(\mathcal{R}_2)$ (b) $\mathcal{G}_{sc}(\mathcal{R}_3)$

Figure 11: Scheme graphs used in the experiments

We implemented our algorithms for computing full disjunctions in the open-source database system PostgreSQL, Version 8.0.3. The experiments were carried out on a Pentium 4 with a CPU of 1.6GHZ and 512MB of RAM, running the Linux Mandrake 2.6.3-7mdk operating system. We discuss implementation details and our experimentation.

7.1 Implementation Details

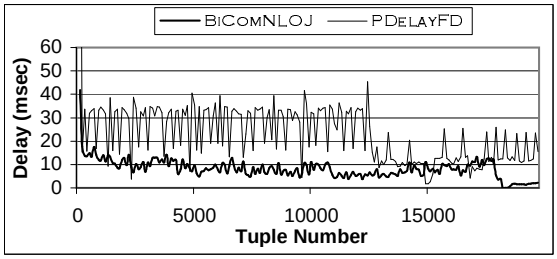
We implemented both PDelayFD and BiComNLOJ, and we added basic optimizations to this implementation. For example, we strategically discarded some intermediate results when it was possible to prove that these results would not yield new tuples. We also tried different strategies for choosing relations while extending tuple sets in ExtendToMax. The most important optimization in the implementation of BiComNLOJ is that in the execution of TupleExtFD, the generated tuples are cached on the disk. If TupleExtFD is called more than once with the same argument t, it retrieves the answer from the cache.

The presented algorithms are tuple based, however, our implementations are block-based versions of these algorithms. Intuitively, this means that whenever possible, they loop over blocks of tuples, instead of over individual tuples. This speeds up of the computation, in several places. For example, ExtendToMax potentially reads the entire contents of the database. In its block-based version, several tuples are extended at once, thereby reducing the number of reads from the disk. Our implementation of PDelayFD is fully block based. Our implementation of BiComNLOJ is only partially block based at present, due to the need for rather complex bookkeeping in a fully block-based version of this algorithm.

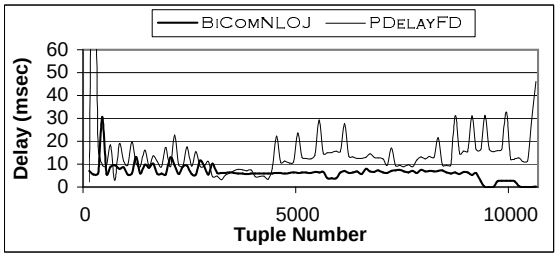
In addition to PDelayFD and BiComNLOJ, we have also implemented (within the PostgreSQL setting) a fully block-based version of the algorithm IncrementalFD [2]. This algorithm computes full disjunctions in incremental polynomial time and it is the most efficient algorithm in the literature.

7.2 Experiments and Results

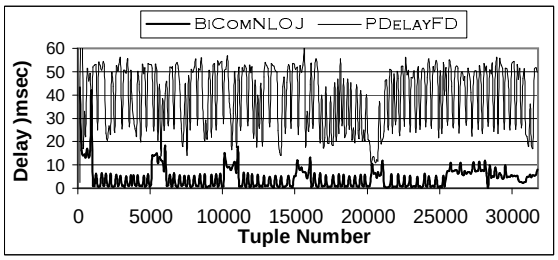
We conducted our experiments on synthetic, randomly generated databases with predefined schemes. We used data-



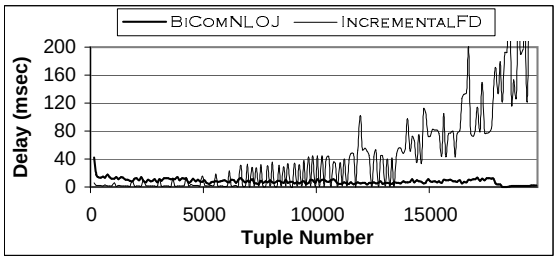
(a) Execution over $\mathcal{G}_{sc}(\mathcal{R}_1)$



(b) Execution over $\mathcal{G}_{sc}(\mathcal{R}_2)$

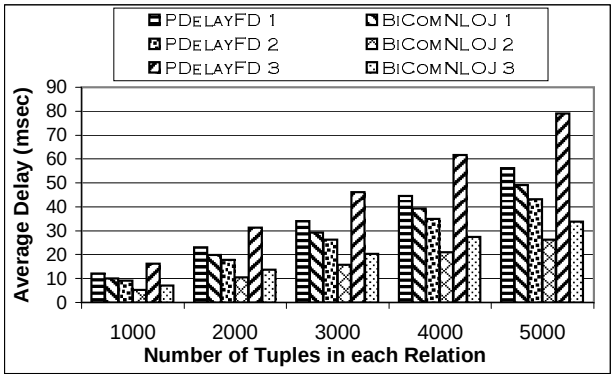


(c) Execution over $\mathcal{G}_{sc}(\mathcal{R}_3)$

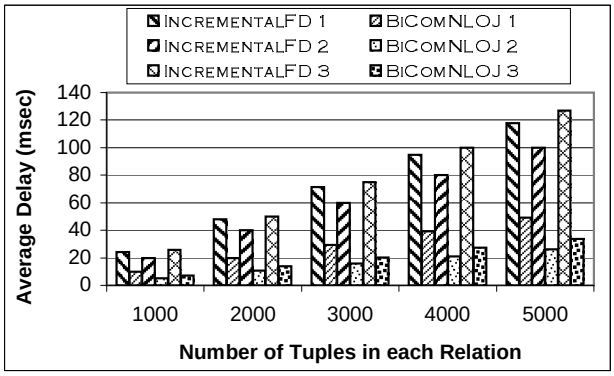


(d) Execution over $\mathcal{G}_{sc}(\mathcal{R}_1)$

Figure 12: Analysis of specific executions



(a) BiComNLOJ vs. PDelayFD



(b) BiComNLOJ vs. IncrementalFD

Figure 13: Delay vs. input size

bases with three different scheme graphs. The first graph is $\mathcal{G}_{sc}(\mathcal{R}_1)$ shown in Figure 1(b) (the relation schemes are given in Figure 1(a)). The other two graphs are $\mathcal{G}_{sc}(\mathcal{R}_2)$ and $\mathcal{G}_{sc}(\mathcal{R}_3)$ shown in Figures 11(a) and 11(b), respectively. Observe that these three scheme graphs contain many relations and are rather complex. The values in each tuple were chosen randomly and uniformly. For each specific database, all tables have the same size and all values were chosen from the same sample space. However, the size of the sample space varies in different databases, as explained later.

In the first experiment, the goal was to explore how the delay varies during the execution. We chose a specific instantiation for each of the three scheme graphs. We divided the output into chunks of 100 tuples each and recorded the average delay of the tuples in each chunk. Each of the tables contains 1000 tuples and the sample space is of size 1000. Figures 12(a) through 12(c) show the results of this experiment for the three scheme graphs and for the algorithms BiComNLOJ and PDelayFD. Figure 12(d) shows the results of this experiment for the scheme graph $\mathcal{G}_{sc}(\mathcal{R}_1)$ and for the algorithms BiComNLOJ and IncrementalFD. Note that the horizontal axis gives the number of the tuple in the output stream and the vertical axis gives the delay measured for this tuple. Also note that the graphs that compare BiComNLOJ and IncrementalFD have a larger scale than those comparing BiComNLOJ and PDelayFD, due to the slower runtime of IncrementalFD.

For the algorithms BiComNLOJ and PDelayFD, the delay is usually longest at the beginning of the execution, since at this point the data structures are being loaded with tuples from the database. The delay of BiComNLOJ diminishes towards the end, due to caching. In contrast, the delay of IncrementalFD is small at the beginning of the execution and becomes substantially long towards the end. Note that there are variations in the delay of all three algorithms due to a mismatch between the number of tuples in an output chunk (which was always 100) and the number of tuples in a block (which varied). So, sometimes a single block included more than 100 tuples of the full disjunction and sometimes several blocks were needed to generate 100 tuples. PDelayFD particularly suffered from this phenomenon, since its implementation is fully block based.

In the second experiment, the goal was to study how the delay is affected by the *size of the data*. In the tables we used, there is a constant ratio between the database size and

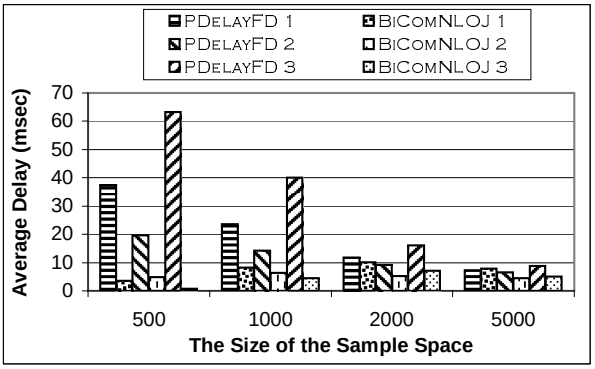


Figure 14: Delay vs. the size of the sample space

the size of the sample space. The results of this experiment, depicted in Figure 13, show the superiority of BiComNLOJ over both PDelayFD and IncrementalFD. (In fact, even PDelayFD is superior to IncrementalFD.) The difference in runtime becomes more significant as the size of the tables grows, i.e., BiComNLOJ scales gracefully with respect to the number of tuples in the input.

In the third experiment, the goal was to study how the delay is affected by the *density of values* in the data. Only BiComNLOJ and PDelayFD were tested in this experiment. All the tables had constant size of 1000 tuples while the size of the sample space varied. The results of this experiment, depicted in Figure 14, show that BiComNLOJ is significantly more efficient than PDelayFD, in particular when the sample space is small and caching can be used efficiently.

8. CONCLUSION

The algorithm BiComNLOJ for computing a full disjunction with polynomial delay was presented. This algorithm builds on two other algorithms, which also run in polynomial delay—NLOJ and PDelayFD. The NLOJ algorithm is very efficient but can be applied only when the full disjunction is equivalent to a left-deep outerjoin (e.g., when the scheme graph is a tree). PDelayFD can be applied to every set of relations but is less efficient (i.e., has a longer delay) than NLOJ. BiComNLOJ combines the two algorithms, thereby providing efficiency while being able to compute the full disjunction of every set of relations. We described some optimizations that can be applied when integrating full disjunctions into SQL query plans. Finally, our experiments prove that indeed BiComNLOJ is more efficient than PDelayFD, and both algorithms are more efficient than the state of the art [2].

The algorithm of [13] uses sequences of outerjoins to compute the full disjunction of a set of relations with a γ-acyclic scheme graph. This algorithm has efficient total-time computation, but does not run in polynomial delay. For the special case that the scheme graph is acyclic, our algorithm NLOJ computes a full disjunction in polynomial delay. As future work, we will study whether NLOJ can be adapted to deal with relations that have general γ-acyclic scheme graphs.

As additional future work, we intend to further explore how to enhance the efficiency of queries that involve full disjunctions. For example, we intend to study how the specific choice of the left-deep sequence of outerjoins affects the performance of the algorithm BiComNLOJ. Consequently, we will seek techniques for optimizing the choice of such sequences, e.g., by taking indices into consideration. We will also explore how existing techniques for outerjoin reordering [15, 8, 9, 14] can be used in our algorithms. Finally, we intend to explore how histograms and other statistical information can be exploited.

Acknowledgments

Sara Cohen was supported by the Israel Science Foundation (Grant 1032/05). Yaron Kanza was supported by the Natural Science and Engineering Research Council of Canada. Benny Kimelfeld and Yehoshua Sagiv were supported by the Israel Science Foundation (Grants 96/01 and 893/05).

9. REFERENCES

[1] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, 1977.

[2] S. Cohen and Y. Sagiv. An incremental algorithm for computing ranked full disjunctions. In *PODS*, 2005.

[3] T. Corman, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[4] C. J. Date. The outer join. In *ICOD*, 1983.

[5] C. J. Date. *Relational Database: Selected Writings*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[6] R. Fagin. Degrees of acyclicity for hypergraphs and relational database schemes. *J. ACM*, 30(3), 1983.

[7] C. Galindo-Legaria. Outerjoins as disjunctions. In *SIGMOD*, 1994.

[8] C. A. Galindo-Legaria and A. Rosenthal. How to extend a conventional optimizer to handle one- and two-sided outerjoin. In *ICDE*, 1992.

[9] C. A. Galindo-Legaria and A. Rosenthal. Outerjoin simplification and reordering for query optimization. *ACM Trans. Database Systems.*, 22(1), 1997.

[10] H. Garcia-Molina, J. D. Ullman, and J. Widom. *"Database Systems—The Complete Book"*. Prentice Hall, 2002.

[11] D. S. Johnson, M. Yannakakis, and C. H. Papadimitriou. On generating all maximal independent sets. *Information Processing Letters*, 27, March 1988.

[12] Y. Kanza and Y. Sagiv. Computing full disjunctions. In *PODS*, 2003.

[13] A. Rajaraman and J. D. Ullman. Integrating information by outerjoins and full disjunctions. In *PODS*, 1996.

[14] J. Rao, H. Pirahesh, and C. Zuzarte. Canonical abstraction for outerjoin optimization. In *SIGMOD*, 2004.

[15] A. Rosenthal and C. A. Galindo-Legaria. Query graphs, implementing trees, and freely-reorderable outerjoins. In *SIGMOD*, 1990.