[DGS94] DE CASTRO C., GRANDI F., SCALAS M.R.: "Interoperability of Heterogeneous Temporal Relational Databases", Proc. of *12th International Conference on Entity-Relationship Approach (ER '93)*, 460-473, also in Lecture Notes in Computer Science, Springer-Verlag 1994, Vol. 823, 463-474.

[GST91] GRANDI F., SCALAS M.R., TIBERIO P.: "A History Oriented Data View and Operation Semantics for Temporal Relational Databases", C.I.O.C.-C.N.R. Tech. Rep. No. 76, Bologna, Jan. 1991.

[JW93] JAJODIA S., WANG X.S.: "Temporal Mediators: Supporting Uniform Access to Heterogeneous Temporal Databases," Proc. of *Workshop on Interoperability of Database Systems and Database Applications*, Fribourg, 1993.

[TCG+93] TANSEL A., CLIFFORD J., GADIA V. et al.: *Temporal Databases: Theory, Design and Implementation*, The Benjamin/Cummings Publishing Company, Redwood city, California, 1993.

# An Output Sensitive Solution to the Set Union and Intersection Problem

Carlo Gaibisso[1], Enrico Nardelli[1,2], Guido Proietti[2]

1: Istituto di Analisi dei Sistemi ed Informatica, C.N.R., 00185 Roma, Italy
2: Dipartimento di Matematica Pura ed Applicata, Università di L'Aquila, 67100 Italy

**Abstract**

In this paper we propose an alternative approaches for the efficient solution of the set union and intersection problem, a variant of the classical disjoint set union problem, in which two distinct sequences of unions are simultaneously performed on two distinct collections of $n$ singletons. In this context, it makes sense to introduce a new operation, named *findint*($x$), that, given an element $x$, returns the intersection of the two sets containing $x$ in both collections. We use a new data structures called *intersection lists* that reaches the optimal $\Theta(p)$ worst case time for executing a *findint* operation, where $p$ is the size of the output, and spends $O(n)$ worst case time for a *union* and $O(1)$ worst case time for a *find* operation.

## 1 Introduction

Given a collection of $n$ disjoint singletons $S_1, S_2, ..., S_n$, the *disjoint set union problem* [4] consists of efficiently performing an intermixed sequence of the following operations:

*union*($S_i$,$S_j$): combine the disjoint sets named $S_i$ and $S_j$ into a new set named $S_i$;
*find*($x$): return the name of the unique set containing the element $x$.

The *set union and intersection problem* [2] is a natural extension of the disjoint set union problem. It can be formulated in the following way: given two collections of $n$ disjoint singletons:

$S=\{S_1, S_2, ..., S_n\}$
$T=\{T_1, T_2, ..., T_n\}$

such that initially $S_i$ contains the same element of $T_i$ for $i$=1, ...,$n$, perform an intermixed sequence of the following operations:

*union*($S$,$S_i$,$S_j$): combine disjoint sets of the collection $S$ named $S_i$ and $S_j$ into a new set named $S_i$;
*union*($T$,$T_i$,$T_j$): combine disjoint sets of the collection $T$ named $T_i$ and $T_j$ into a new set named $T_i$;
*find*($S$,$x$): return the name of the set in collection $S$ containing the element $x$;
*find*($T$,$x$): return the name of the set in collection $T$ containing the element $x$;
*findint*($x$): return the intersection of the two sets containing $x$ in $S$ and $T$.

As an example of application of the set union and intersection problem consider the incremental maintenance of constraints in a concurrent environment. Here, concurrent agents working on different sets of constraints define different partitions into equivalence classes over the universe of feasible values. When it needs to know which is the equivalence class of a given value with respect to two different agents, a set union and intersection problem has to be solved.

The known algorithms for the set union and intersection problem make use of classical data structures employed to efficiently solve the set union problem.

Furthermore, in [3], a new quick-find data structure, called *binary list*, having the same performances of the well-known quick-find trees, has been introduced and analysed. The model of computation considered is the *pointer machine* [5], and the known upper bounds for the set union and intersection problem apply to the class of algorithms (called *separable algorithms*) which satisfy the following rules:

(i) The operations are presented on-line, i.e., each operation must be completed before the next one is known.
(ii) Each set element of each collection is separate node of the data structure.
(iii) (*Separability*) After any operation, the data structure can be partitioned into subgraphs such that each subgraph corresponds exactly to a current set. There exists no edge from a node in such a subgraph to a node outside the subgraph.
(iv) For executing a *find*$(S,x)$ (*find*$(T,x)$) operation the algorithm obtains the node $v$ containing the requested element in the collection $S$ ($T$). The algorithm follows paths with start node $v$ until it reaches the node which contains the name of the corresponding set.
(v) For executing a *find*, a *union* or a *findint* operation the algorithm may insert or delete any edge as long as rule (iii) is satisfied.

All of these algorithms perform a *findint* operation, in the best cases, in $O(n)$ time. Unfortunately, this time is far from the optimal, corresponding to $O(p)$ time, where $p$ is the size of the output. In this paper we propose an output sensitive approach that allows to decrease the cost for a *findint* operation to the optimal, without using any additional storage. Thus, in a highly dynamic context, when the number of *findint* operations becomes higher and higher, the saving in terms of global charged time will be larger.

# 2 A gallery of known algorithms

In this section we recall how to solve the set union and intersection problem using classical data structures, that is quick-find trees, quick-union trees [5], k-UF trees [1] and binary lists [3]. All of them are linked structures. The way subgraphs representing sets are built during *union* operations determines the complexity of various approaches. Figure 1 shows the general schema we use to satisfy a *findint* operation. Procedure *FIND*$(X,x)$ returns the pointer to the representative element of the set containing $x$ in the collection $X$, while procedure *OUTPUT*$(x)$ outputs element $x$:

```
procedure FINDINT(x); /*return the intersection of sets find(S,x) and
find(T,x);
begin
if ( |FIND(S,x)| ≥ |FIND(T,x)| ) then INTERSECT(FIND(T,x),FIND(S,x),S)
   else INTERSECT(FIND(S,x), FIND(T,x), T)
end;

procedure INTERSECT(A,B,Z);
   /*return the intersection of elements pointed by A and B, where elements
   /*pointed by B belong to collection Z;
begin
for (each element x contained in the set whose representative element is A) do
   if (B=FIND(Z,x)) then OUTPUT(x)
end;
```

**Fig. 1**: Procedure *FINDINT*$(x)$



Note that the efficient execution of the loop in the procedure INTERSECT(), requires elements in each set be circularly linked together. In this way, from anyone of them, all the elements in the same set can be reached in time proportional to the number of elements in the set itself. Details on how to manage these circular lists for the various classical data structures we are going to study are straightforward and left to the reader.

## 2.1 Weighted quick-find trees

In quick-find trees, a *union* is performed by making all the element of one set children of the tree root of the other. Thus a *union* costs $O(n)$ and, since every element points to the root, a *find* costs $O(1)$. Using the freedom implicit in each *union* operation, we make the children of the smaller tree point to the root of the larger. This improves to $O(\log n)$ the amortized complexity of a *union*.

**Proposition 1:** Using weighted quick-find trees, a single *findint* operation can be executed in $O(n)$ worst-case time.

**Proof:** Trivial. □

## 2.2 Weighted quick-union trees

In quick-union trees a *union* is performed by making the tree root of one set children of the tree root of the other. Thus a *union* costs $O(1)$. A *find* is performed by starting from the node representing the requested element and following the pointer to the parent until the tree root is reached, and thus a *find* costs $O(n)$. This time, making weighted unions (on the basis of the size or the rank of the involved trees), we can improve to $O(\log n)$ the worst case complexity of a *find*. The best possible solution for the worst case time of a sequence of operations can be achieved by applying one of the known *compaction rules* [6]. This leads to $O(n+m\alpha(m+n,n))$ amortized time complexity on a sequence of $n$ *union* and $m$ *find* operations, where $\alpha$ is the functional inverse of the Ackermann's function.

**Proposition 2:** Using weighted quick-union trees, a single *findint* operation can be executed in $O(n\log n)$ worst-case time.

**Proof:** Trivial. □

## 2.3 k-UF trees

k-UF trees [1] support each *union* and each *find* in $O(\log n/\log\log n)$ time in the worst case, so balancing the cost of the two operations. No better bound is possible for any separable pointer algorithm.

**Proposition 3:** k-UF trees support a *findint* operation in $O(n\log n/\log\log n)$ time in the worst case.

**Proof:** Trivial. □

## 2.4 Binary lists

Binary lists [3] have the same performances of the well-known quick-find trees. In addition, they allow to sort each set in $O(n\log\log n)$ time, while quick-find trees need $\Omega(n\log n)$ time. This means, for example, that the new structure is able to produce sorted output to a *findint* request in a better time than the classical approaches. Such

a property can be useful every time that a priority is attached to an element of the universe.

We briefly describe the binary list data structure. Given an integer $p \geq 0$, let $L_p$ be a list with exactly $2^p$ nodes sorted in decreasing order: we say that $L_p$ is a *bit list* (*b-list* for short) of order $p$. In order to use b-lists for representing sets whose number $k$ of elements is not a power of two, we consider the binary representation of $k=\Sigma_{i\geq 0} b_i 2^i$, with $b_i \in \{0,1\}$, and we define a *binary list* $C_k$ of order $k$ as the concatenation (linking) of b-lists, one for each 1 in the binary representation of $k$. Therefore the $i$-th component of $C_k$ is $L_i$ if $b_i=1$ and empty otherwise. With respect to classical quick-find trees, an element of a binary lists makes use of an additive pointer field containing the pointer to the next element in the binary list (in the representative element, this pointer points to the first element of the binary list representing the set).

In [3] it has been proved that using binary lists to represent sets it is possible to execute a *union* operation in $O(n)$ time in the worst case and in $O(\log n)$ amortized time. Of course, the standard *find* operation in $S$ or $T$ can be executed in constant time, since each element points to the representative element, and then it follows:

**Proposition 4:** Using binary lists to represent sets it is possible to execute a *findint* operation in $O(n)$ worst case time.

**Proof:** Trivial. □

We resume in Table 1 below worst case performances for each operation of the proposed approaches:

| Data Structure | *find* | *union* | *findint* |
|---|---|---|---|
| *Quick-Find Trees* | $O(1)$ | $O(n)$ | $O(n)$ |
| *Quick-Union Trees* | $O(\log n)$ | $O(1)$ | $O(n \log n)$ |
| *k-UF Trees* | $O\left(\frac{\log n}{\log\log n}\right)$ | $O\left(\frac{\log n}{\log\log n}\right)$ | $O\left(\frac{n \log n}{\log\log n}\right)$ |
| *Binary Lists* | $O(1)$ | $O(n)$ | $O(n)$ |

**Table 1:** Worst case time bounds for each operation

The following Table 2 resumes the worst case bounds on a sequence of $2n$-2 *union*, $m$ *find* and $k$ *findint* operations for all the considered structures:

| Data Structure | Time |
|---|---|
| *Quick-Find Trees* | $O(n \log n+m+kn)$ |
| *Quick-Union Trees* | $O(n+m\alpha(m+n,n)+kn \log n)$ |
| *k-UF Trees* | $O\left((n+m)\left(\frac{\log n}{\log\log n}\right) + kn \frac{\log n}{\log\log n}\right)$ |
| *Binary Lists* | $O(n \log n+m+kn)$ |

**Table 2:** Worst case time bounds on a sequence of operations



# 3 An output sensitive approach

Solving the set union and intersection problem using classical approaches allows to perform a *findint* operation, in the best cases, in $O(n)$ time. Unfortunately, this time is far from the optimal, corresponding to $O(p)$ time, where $p$ is the size of the output. In this section we propose an approach that allows to decrease the cost for a *findint* operation to the optimal, without using any additional storage. Thus, in a highly dynamic context, when the number of *findint* operations becomes higher and higher, the saving in terms of global charged time will be larger.

Let assume for simplicity that it is initially $S_i=T_i=\{i\}$. The *universe* of elements over which the two collections are defined is then $U=\{1, 2, ...,n\}$. Let $Q=U/\mathcal{R}$ be the quotient set of $U$ by the equivalence relation $\mathcal{R}$:

$$x,y \in U,\ x\mathcal{R}y \Leftrightarrow x,y \in S_i \text{ and } x,y \in T_j$$

We improve the cost of a *findint* operation by dynamically maintaining up-to-date the set (of sets) $Q$. Remember that, for the assumptions we made on the model, no additional nodes can be used to maintain $Q$. Then we have to represent $Q$ implicitily, inside the sets of the two collections. To do that, we note that, on the basis of $\mathcal{R}$, every set in one of the collections is partitioned in subsets made up of those elements belonging at a same set into the other collection. If we update this subsets at each *union* operation executed on $S$ or $T$, then a *findint*($x$) can be executed in $O(find(S,x)+p)$ time, where $p$ corresponds to the output size; this implies that if we can execute a *find* operation in constant time, then a *findint* operation is performed in optimal time. To obtain this, we represent sets in the collections using trees in which each element points to the root, so that a *find* operation can be executed in $O(1)$ time and we add to the record structure the following field:

INT: this is a pointer field containing the pointer to the next element in the intersection list; initially, this pointer points to itself.

We call *intersection list* the structure resulting from this modification. To update $Q$ at each *union*, we use the procedure given in Figure 2.

The collection where most recently has occurred a *union* operation contains the updated status of $Q$. Thus, when a *findint*($x$) occurs, we focus on the most recently collection interested by a *union* operation (which can be determined in $O(1)$ time) and then we output the intersection list attached to $x$. Resuming, the following can be easily proved:

**Theorem 1:** Performing a *union* operation on $S$ or $T$ and maintaining the intersection lists costs $O(n)$ operations.

**Proof:** The analysis of the worst case is given in the comments of the procedure *UNION*() given above. □

This bound is tight, since there can be *union* operations on $S$ or $T$ producing $\Omega(n)$ merging in $Q$. Consider, for example, the following case: assume, without loss of generality, that $n$ is even; suppose that after $n$-2 *union* operations on $S$, this is made up by two sets (the set of odd numbers and the set of even numbers) and that after $n/2$ *union* operations on $T$, this is made up by $n/2$ sets, each containing two consecutive numbers. At the next *union* on $S$, the two residual sets are unified, and

```
procedure UNION(S,A,B);
    /*combine the sets named A and B of the collection S into a new set
    /*named A and update the intersection subsets of A;
  begin
  WEIGHTED_UNION(A,B);
    /*make the children of the root of the smaller tree point to root of the
    /*larger; this has a worst-case time complexity of
    /*O(min(|A|,|B|))=O(n) and an amortized cost of O(logn);
  for (each element x contained in A) do
    begin
      /*this loops costs Σ_{∀x∈A} O(|B_x|), where B_x is the generic set in T containing
      /*x; since each set in T is examined exactly once, summation is limited by n;
    B_x=FIND(T, x);     /*this costs O(1) time;
    if (B_x is unmarked) then
      begin
        MARK_SET(B_x); /* B_x is marked, to be examined only once;
        for (each element y other than x contained in B_x) do
          begin    /*this loop costs O(|B_x|)
            A_y=FIND(S, y)        /*this costs O(1) time;
            if (A_y=A) then
              begin    /*update the intersection lists in S and T;
                APPEND(S,x,y) ;
                APPEND(T,x,y);
              end
          end
      end
    end
  end;
```

**Fig. 2:** Procedure *UNION*(*S*,*A*,*B*)

there will be $n/2$ merging among the $n$ singletons in $Q$. This means that the lower bound for operation is in the worst-case $\Omega(n)$ and then we have in any case to pay $\Omega(n)$ to perform a *union* operation and simultaneously to update $Q$. On the other hand, what it happens on a sequence of *union* operations? The following proposition characterizes the problem of maintaining up-to-date the status of $Q$ at each *union* operation; for the sake of clarity, we will assume the existence of a third collection $C$ maintaining on-line the intersections among sets in $S$ and in $T$:

**Theorem 2:** Given a complete sequence of $2n$-2 unions on two collections $S$ and $T$, any algorithm based on comparisons requires $\Omega(n\log n)$ comparisons to maintain up-to-date the intersection collection $C$ at each *union* operation.

**Proof:** Without loss of generality, suppose that $S$ and $T$ contain $n=m^2=2^k$ elements. At the beginning we have:

$S=\{S_1, S_2, ..., S_n\}$
$T=\{T_1, T_2, ..., T_n\}$
$C=\{C_1, C_2, ..., C_n\}$

where $S_i=T_i=C_i=\{i\}$. Consider now the following situation, where ($n$-$m$-1) unions on $S$ and ($n$-$m$-1) unions on $T$ have been done but $C$ is still made up by singletons:



$S=\{A_1, A_2, ..., A_m\}$  $T=\{B_1, B_2, ..., B_m\}$

with:

$A_1=\{a_{1,1}, a_{1,2}, ..., a_{1,m}\}, A_2=\{a_{2,1}, a_{2,2}, ..., a_{2,m}\}, .., A_m=\{a_{m,1}, a_{m,2}, ..., a_{m,m}\}$
with $a_{i,j} \in [1..n]$
$B_k=\{a_{i,k} \mid i=1,2, ...,m\}$ for $k=1,2, ..., m$.

To complete the union sequence, ($m$-1) unions on $S$ and ($m$-1) unions on $T$ are needed. We now show a particular sequence of unions on $S$ and $T$ that requires $\Omega(n\log n)$ comparisons to maintain up-to-date $C$. We divide such a sequence in *steps*; one step is a sequence of unions on the same collection which halves the number of contained sets. The first step manipulates $S$ executing $m/2$ unions. The $i$-th union in this step merges sets $A_{2i-1}$ and $A_{2i}$, $i=1, 2, ..., \frac{m}{2}$. Each merging of two sets in $S$ generates $m$ couples in $C$. This set of $m$ couples belongs to a universe of $m(m-1)(m-2)...1=m!$ possible sets of $m$ couples[1]. The size of the universe implies that we have to do at least $\log(m!)$ comparisons, that is (by the Stirling's approximation) at least $m\log m$ comparisons. Since we execute $\frac{m}{2}$ unions in the first step (generating $\frac{n}{2}$ couples in $C$), we spend a total of:

$$C_1 \geq \frac{m}{2} m \log m = \frac{m^2 \log m}{2} = \frac{n \log \sqrt{n}}{2} = \frac{n}{4} \log n$$

The second step manipulates $T$. The $i$-th union in this step merges sets $B_{2i-1}$ and $B_{2i}$, $i=1, 2, ..., \frac{m}{2}$. Each merging of two sets in $T$ at this step generates $\frac{n}{4}$ quadruples in $C$. This set of $\frac{m}{2}$ quadruples belongs to a universe of $\left(\frac{m}{2}\right)!$ possible sets of $\frac{m}{2}$ quadruples. In fact, now in $S$ there are sets of $2m$ elements each, so when unions are executed on $T$, quadruples result in $C$.

This means that we have to do at least $\log\left(\left(\frac{m}{2}\right)!\right)$ comparisons, that is (by the Stirling's approximation) at least $\frac{m}{2}\log\frac{m}{2}$ comparisons. Since we execute $\frac{m}{2}$ unions in the second step (generating $\frac{n}{4}$ quadruples in $C$), we spend a total of:

---

[1] In fact, given two disjoint sets $X$ and $Y$ of $m$ distinct elements each, there exist $m!$ different sets of exactly $m$ couples $\{x, y\}$ where $x \in X$ and $y \in Y$. To see why, consider that each element in the universe can be obtained in the following way: take the first element in $X$; this element can be coupled with any of the $m$ elements contained in $Y$; then, take the second element in $X$; this element can be coupled with any of the ($m$-1) remaining elements contained in $Y$; following this procedure, at the $m$-th coupling, we are forced to take the residual elements in the two merging sets.

$$C_2 \geq \frac{m}{2}\frac{m}{2}\log\frac{m}{2} = \frac{n}{8}\log\frac{n}{2}$$

By iterating the process, we have that at the last step on $T$ (i.e., the $(2\log m)$-th=$k$-th step) we merge two sets of cardinality $\frac{n}{2}$, producing $C=\{U\}$ without any comparison. The total cost $C_{TOT}$ is therefore:

$$C_{TOT} = \sum_{i=1}^{k-1} C_i \geq \sum_{i=1}^{k-1} \frac{n}{2^{i+1}} \log\frac{n}{2^{i-1}} = \sum_{i=1}^{k-1} \frac{n}{2^{i+1}} \left(\log n - \log 2^{i-1}\right) =$$

$$= \sum_{i=1}^{k-1} \frac{n \log n}{2^{i+1}} - \sum_{i=1}^{k-1} n \frac{i-1}{2^{i+1}} = n \log n \left(\frac{1}{2} - \frac{1}{2^k}\right) - n\left(\frac{1}{2} - \frac{k}{2^k}\right) = \frac{1}{2} n \log n - \frac{1}{2} n$$

from which the thesis. □

Finally, the following corollary immediately descends by the previous results:

**Corollary 1:** The set union and intersection problem can be solved dynamically by maintaining the intersection lists spending $O(n^2+m+P)$ time in the worst case on a sequence of $2n$-2 *union* operations, $m$ *find* operations and $k$ *findint* operations, where $P$ is the output size over all the $k$ *findint* operations. □

# 4 Conclusions

In this paper we have proposed an output sensitive approach to the set union and intersection problem that allows to perform a *findint* operation in optimal time.

Future work will be focused in the direction of determining lower bounds for a sequence of *find*, *union* and *findint* operations in the context of separable algorithms.

# References


[1] N. Blum, On the Single Operation Worst-case Time Complexity of Disjoint Set Union Problems, in *SIAM Journal on Computing*, 15, 1986, pp. 1021-1024.

[2] C.Gaibisso, E.Nardelli and G.Proietti, Algorithms for the Set Union and Intersection Problem, Technical Report nº 83 April 1995 of the Department of Pure and Applied Mathematics of the University of L'Aquila.

[3] C.Gaibisso, E.Nardelli and G.Proietti, Intersection Reporting on two Collection of Disjoint Sets, submitted to *Information Processing Letters*, April 1996.

[4] Z. Galil and G.F. Italiano, Data Structures and Algorithms for Disjoint Set Union Problems, in *ACM Computing Surveys*, 23 (3), 1991, pp. 319-344.

[5] R.E. Tarjan, A Class of Algorithms which Require Nonlinear Time to Maintain Disjoint Sets, in *Journal of Computer and System Sciences*, 18, 1979, pp. 110-127.

[6] R.E. Tarjan and J. Van Leeuwen, Worst-Case Analysis of Set Union Algorithms, in *Journal of the Association for Computing Machinery*, 31 (2), 1984, pp. 245-281.


# On the Semantics of Multistage Interconnection Networks*


Anna Gambin** and Sławomir Lasota

Fachbereich Informatik LS-2, Dortmund Universität, D-44221 Dortmund 50, Germany.
Instytut Informatyki, Uniwersytet Warszawski, Banacha 2, Warszawa 02-097, Poland.
{aniag,sl}@mimuw.edu.pl



**Abstract.** Multistage interconnection networks (MINs) have a number of applications in many areas, for example in communication networks or parallel computer systems. While several performance analyses have been done, as far as we know a formal description of the behaviour of such networks is lacking. In the paper we define Markov chains describing the behaviour of the buffered MIN with a butterfly interconnection structure and $2 \times 2$ switching elements. We study several models of packet flow, which appear in the literature and develop technical mathematical tools which allow us to compare them. We prove some equivalence results regarding the models.


## 1 Introduction

Multistage interconnection networks (MIN) have been widely used for connecting processors in parallel computing systems and in constructing communication networks [7]. It consists (in general) of $(k \times k)$-switching elements (*switches*), which are grouped into several stages. In this paper we are concerned with MINs having the butterfly interconnection structure containing $(2 \times 2)$-switches with input buffers of capacity one (see fig. 1). They belong to the broader class of *banyan networks* [1].

A number of papers studied the behaviour of MINs, aiming mainly at evaluation of their performance [1] [3, 5, 9]. In most of these papers some crucial assumptions (relaxations) are made, simplifying the analysis of the network. The reason is that the behaviour of MIN, arising from the dependencies between neighbour switches, is very complex.

In this paper we propose the formal definition of the dynamic behaviour of MIN without making any simplifying assumptions, i.e. in the *full blocking model*. As far as we know, no such definitions have been proposed by now. We describe the network by means of Markov chain, as its behaviour is history-independent. We develop some mathematical machinery which allows us to define


* This work was performed at the University of Dortmund.
** Research supported by DAAD (Deutscher Akademischer Austauschdienst).


[1] The most frequently evaluated performance measure is *throughput* (expected number of packets reaching their destinations per time step).