Adaptive Set Intersections, Unions, and Differences

Erik D. Demaine* Alejandro López-Ortiz† J. Ian Munro*

Abstract

Motivated by boolean queries in text database systems, we consider the problems of finding the intersection, union, or difference of a collection of sorted sets. While the worst-case complexity of these problems is straightforward, we consider a notion of complexity that depends on the particular instance. We develop the idea of a *proof* that a given set is indeed the correct answer. Proofs, and in particular shortest proofs, are characterized. We present adaptive algorithms that make no a priori assumptions about the problem instance, and show that their running times are within a constant factor of optimal with respect to a natural measure of the difficulty of an instance. In the process, we develop a framework for designing and evaluating adaptive algorithms in the comparison model.

1 Introduction and Overview

Our work can be seen in the general context of performing searches quickly in a database or data warehousing environment. The broad issue is that of characterizing what type of join operations can be performed without scanning the relations involved or actually materializing intermediate relations. The specific problem addressed here can be seen in that context or in the context of performing a web search, or a search in another large text database, for documents containing some or all of a set of keywords. For each keyword we are given the set of references to documents in which it occurs [2, 6, 9]. These sets are stored in some natural order, such as document date.

In practice, the sets are large. For example, the average word from user query logs matches approximately a million documents on the AltaVista web search engine. Of course, one would hope that the answer to the query is small, particularly if the query is an intersection. It may also be expected that the elements of such an intersection are not spread uniformly through the initial sets. In dealing with news articles in particular, one will find a large number of references to one term over a few relatively short periods, and little outside these periods.

We would like algorithms to take advantage of such features of the data, and indeed develop a model of complexity for classes of instances that take it into account. An extreme example that makes this notion more precise is that of computing the intersection of two sorted sets of size n. On the one hand, if the sets interleave perfectly $\Omega(n)$ comparisons are required. On the other hand, if all elements of one set are known to fall between a pair of consecutive elements in the other set, the problem simply reduces to a search in a sorted array and so $\log_2 n+O(1)$ upper and lower bounds apply.

Similar motivation and examples apply to a more general class of queries, including set union and set difference. While the answer to a union query is at least as large as the largest input set, one may be able to construct the answer using the input sets without even examining much of the input, let alone copying it over. For example, if we are to produce the union of two sets (each represented by a B-tree) and all elements in one set fall between a pair in the other set, the answer is a B-tree consisting of $O(\log n)$ new nodes with references to portions of the two input trees.[1]

This leads to the idea of an *adaptive* algorithm [4, 5, 11]. Such an algorithm should make no a priori assumptions, but determine the kind of instance it faces as the computation proceeds. The running time should be reasonable for the particular instance—not the overall worst-case.

Based largely on the development of a proof in a canonical form and its specification, we develop algorithms whose running times are within constant factors of our worst-case lower bounds. Our methods, while phrased in terms of a pure comparison model, are immediately applicable to any balanced tree (e.g., B-tree) model.

The general outline of our approach, which we apply to each of the three problems, is as follows. First, in Section 2, we characterize "proofs" that an algorithm has obtained the correct answer. Then, in Section 3, we see how to best encode proofs in binary, the idea being

*Department of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada, email: {eddemaine, imunro}@uwaterloo.ca

†Faculty of Computer Science, University of New Brunswick, P. O. Box 4400, Fredericton, N. B. E3B 5A3, Canada, email: alopez-o@unb.ca

[1] Aspects of this idea are explored for the case of two sets in [10].

that "easy" instances have succinctly encodable proofs. In Section 4, we extend lower bounds beyond the most basic information theoretic argument. In Section 5, we develop an algorithm to find a proof in time matching our lower bound. Finally, in Section 6, we extend these algorithms to produce the desired answer by reusing portions of the input in the output.

2 Proof Structure

We are interested in three problems concerning sets. In all cases, an *instance* of the problem is a collection of k sets $A_1, \ldots, A_k$, each presented in sorted order. Hence

$$A_s = \{A_s[1], \ldots, A_s[n_s]\}$$

implies that $A_s[i] < A_s[j]$ for all s and $i < j$. Some basic terminology that we will use throughout this paper is as follows. An *element* is one of the $A_s[i]$'s; a *value* is a member of the universe, which may occur as an *element* in several sets. An element $A_s[i]$ *precedes* [*weakly precedes*] $A_s[j]$ if $i < j$ [$i \leq j$]. Successors and weak successors are defined similarly; note that they only involve elements in the same set.

Consider the following *set problems*:

1. *Intersection*: Compute $A_1 \cap \cdots \cap A_k$.
2. *Union*: Compute $A_1 \cup \cdots \cup A_k$.
3. *Difference*: Compute $A_1 - (A_2 \cap \cdots \cap A_k)$.[2]

Our work explores each of these three problems in the *comparison model*. That is, the only way in which an algorithm can use the elements of the sets is to test whether $A_s[i]$ is less than, equal to, or greater than $A_t[j]$, for given s, t, i, j. An algorithm also knows the *signature* of the instance, that is, the size n_s of each set A_s.

Any algorithm for a set problem must also be able to construct a proof that its answer is correct. Hence, we focus on algorithms for computing such proofs. This is particularly helpful in the context of unions and differences, where enumerating the elemnts of the answer can take more time than computing the proof. On the other hand, we will show that this explicit enumeration can be avoided. Concentrating initially on proofs allows us to ignore this problem until we have the tools to solve it.

Formally, an *argument* is a finite set of symbolic equalities and inequalities, or *comparisons*, of the form $(A_s[i] < A_t[j])$ or $(A_s[i] = A_t[j])$ for $s, t, i, j \geq 1$. An instance *satisfies* an argument if all the comparisons in the argument hold for that instance.

The most interesting classes of arguments are those that prove that the answer to one of the three problems is a particular set. Formally, an argument P is called a *proof* for a particular set problem if all of the instances satisfying P have the same solution to that problem. If the answer is always the set A of elements, we call P an A-proof.

We cannot say much about the structure of proofs until we fix the problem to solve. This is done in each of the following subsections, in which we analyze what arguments are proofs. We also study the structure of *ordered* arguments and proofs, i.e., arguments and proofs with an associated order on the comparisons.

2.1 Intersection Proofs.

An intersection proof must show precisely which elements are contained in all sets:

LEMMA 2.1. *An argument P is a B-proof for the intersection problem precisely if there are elements $b_1, \ldots, b_k$ for each $b \in B$, where b_i is an element of A_i and has the same value as b, such that*

1. *for each $b \in B$, there is a tree on k vertices, every edge (i, j) of which satisfies $(b_i = b_j) \in P$; and*
2. *for consecutive values $b, c \in B \cup \{+\infty, -\infty\}$, the subargument involving the following elements is a $\emptyset$-proof for that subinstance: from each A_i, take the elements strictly between b_i and c_i.*

Thus, we turn our attention to $\emptyset$-proofs, that is, how to prove that a collection of sets is disjoint. The basic structure in a $\emptyset$-proof is to "eliminate" elements. Intuitively, if we make the comparison $(a < b)$ where b is the first element in its set (called "minimal"), then a and all its predecessors cannot be in the intersection of all of the sets, because in particular they are not in b's set. Thus, we say that a and its predecessors are "eliminated." Furthermore, the element immediately succeeding a is a new "minimal" element that can be used for further elimination.

More formally, recursively define an element e to be *eliminated* (in an argument P) if either

1. $(a < b) \in P$ where e is a weak predecessor of a, and b has no uneliminated predecessors;
2. $(a < b) \in P$ where e is a weak successor of b, and a has no uneliminated successors.

If b has no uneliminated predecessors [successors], we call it *minimal* [*maximal*]. Note that a minimal or maximal element may be eliminated.

[2] The difference operation is somewhat unnatural for more than two sets. We choose this generalization because difference is most like intersection.

Lemma 2.2. *An argument is a $\emptyset$-proof precisely if an entire set is eliminated.*

An important concept about ordered $\emptyset$-proofs is the notion of "low-to-high orderings." First, we need to introduce some additional terminology. In an ordered argument, we say that the ith comparison *eliminates* an element if the subargument with the first i comparisons has this element eliminated; the ith comparison *newly eliminates* an element if in addition just the first $i-1$ comparisons do not have this element eliminated. A *low-to-high ordering* of an argument is an ordering with the property that each comparison $(A_s[i] < A_t[j])$ newly eliminates elements just in A_s, unless it entirely eliminates A_s (in which case it may newly eliminate elements in all sets).

Theorem 2.1. *Every $\emptyset$-proof has a low-to-high ordering.*

Although not directly related to our study of finding proofs in the minimum amount of time, we mention a simple greedy method[3] to exhibit proofs with the fewest possible number of comparisons. Note this does not mean that the algorithm makes the fewest comparisons possible to actually discover a $\emptyset$-proof, and indeed our algorithms in Section 5 will search for easier-to-find proofs.

Define the *immediate successor* of an element $A_s[i]$ to be $A_s[i+1]$ if it exists, and infinity otherwise.

Method Fewest-Comparisons

1. Initialize the eliminator e to the maximum element $A_s[1]$ over all $1 \leq s \leq k$.
2. Until e becomes infinity:
   (a) Add the comparison $(a < e)$ to the proof, where the element a is chosen so that its immediate successor e' is maximized, subject to the constraint that $a < e$.
   (b) If $e \neq e'$, set e to e'.
   (c) Otherwise, e is present in all sets:
      i. Remove the just-added comparison $(a < e)$ from the proof.
      ii. Add the comparisons $(e_s = e_{s+1})$ to the proof, where e_s is the occurrence of e in A_s, for each $1 \leq s < k$.
      iii. Reinitialize e to the maximum immediate successor of e_i over all $1 \leq s \leq k$.

Theorem 2.2. *For any given instance, Method Fewest-Comparisons generates a proof for the intersection problem with the fewest comparisons possible.*

[3]We sidestep the technical details of an "algorithm" and nondeterministic choices with the term "method."

2.2 Difference Proofs. The difference problem $A_1 - (A_2 \cap \cdots \cap A_n)$ is much like the corresponding intersection problem $A_1 \cap A_2 \cap \cdots A_n$ with a twist in how the answer is reported. Specifically, whenever we find an element common to all the sets, this element is withheld from the answer; and all other elements of A_1 are reported in the answer. This is essentially the opposite of the intersection problem, though in the context of proofs the situation is basically the same:

Lemma 2.3. *An argument P is a B-proof for the difference problem precisely if it is an (A_1-B)-proof for the intersection problem.*

2.3 Union Proofs. Proofs for the union problem have a much simpler structure than intersections and differences.

Lemma 2.4. *An argument P is a proof for an instance I for the union problem precisely if*

1. *for any value v, if $A_{s_1}[k_1], \ldots, A_{s_m}[k_m]$ are exactly the occurrences of v, then there is a tree on m vertices, every edge (i,j) of which satisfies $(A_{s_i}[k_i] = A_{s_j}[k_j]) \in P$; and*
2. *for any value v occurring in I, if v and its immediate predecessor [successor] v' in the union of I do not occupy a common set in I, then for some occurrences e and e' of v and v' respectively, $(e' < e) \in P$ [$(e < e') \in P$].*

Note that we can discard any comparisons that follow transitively from others, i.e., comparisons that do not come from Lemma 2.4. We call such comparisons *useless*, and call others *useful*.

3 Encoding Proofs

The next few sections are concerned with how to find proofs by using as few comparisons as possible. First we must be precise about the phrase "as few as possible" for an adaptive algorithm. It is too much to hope for an adaptive algorithm to use the smallest amount of time possible for each particular instance. The class of instances mentioned in the Introduction, in which all elements of one set fall between two consecutive elements in another, is a clear illustration. Only two comparisons are needed for a proof, yet $\log_2 n$ has been noted as a lower bound for any algorithm required to run on all of those instances.

Thus we require a notion of the worst-case performance of an adaptive algorithm. Of course, we cannot use the worst-case running time as our metric, because that will only reflect the case in which the instance is difficult to solve. While apparently unstated in the literature, a natural metric is the worst-case value of the ratio of running time to difficulty, where "difficulty" is an information theoretic measure of the difficulty of the instance. We can think of this ratio as a *scaled running time*, which allows the running time to be large for difficult instances, but enforces it to be small for easy instances. An algorithm that minimizes the worst-case scaled running time is a natural definition of an *optimal* adaptive algorithm. Scaled running time is similar to a "competitive ratio," which measures the effectiveness of the *output* of an algorithm (instead of its running time) relative to the optimal.

Next we require a definition of the difficulty of an instance. A natural definition is the information theoretic lower bound on the running time of any comparison-based algorithm. In the context of this paper, this lower bound is the length of the shortest binary encoding of some proof. As the name suggests, this is a lower bound on the running time of any correct algorithm in the comparison model, for the following reason. An algorithm can only be sure that it knows the correct answer if it knows a proof, or equivalently an encoding of a proof. Because each comparison (over the operators $<$, $=$, and $>$) only reveals a bit of information, the number of comparisons must be at least the length of the shortest binary encoding of a proof. (In fact, the situation is somewhat more complex than this; see the proof of Corollary 3.1.)

The rest of this section analyzes the information theoretic lower bound (that is, optimal encodings of proofs) for each of the three problems. Sections 4 and 5 will use this analysis to prove results about scaled running time.

3.1 Encoding Intersection and Difference Proofs. Let us begin with the basic idea for encoding proofs for the intersection and difference problems (which are identical by Lemma 2.3). We will concentrate on the most important case of $\emptyset$-proofs, which prove that the intersection is empty. Call an element *compared* if it occurs in one of the proof's comparisons. Because compared elements can be arbitrarily spaced out in each set, it is natural to encode the size of the gaps (i.e., the differences in index) between compared elements, which costs roughly $\lg g$ bits for each gap of size g, where $\lg g = \log_2(1+g)$.

We must also handle two further details. First, by appropriately switching between specifying gaps from the low and high sides, we can avoid encoding the largest gap in each set. Second, we must specify the pairing between compared elements that forms the proof.

The following encoding fills in both of these details. Take a low-to-high ordering of the proof P by Theorem 2.1. Let $c = (A_s[i] < A_t[j])$ be the first comparison. First encode s and t using roughly $\log_2 k$ bits and $\log_2(k-1)$ bits, respectively (because $s \neq t$). Encode i $[j]$ by specifying the smallest gap g $[h]$ to an already compared element in A_s $[A_t]$, using essentially $\lg g$ $[\lg h]$ bits. The total cost of encoding c in this way is $\log_2 k + \log_2(k-1) + \lg g + \lg h$.

Encoding all comparisons in this way, we obtain a formula for the length of encoding an entire proof P. We call this length the *cost* of P and denote it by $c(P)$. We break $c(P)$ into two components: set cost $s(P)$ and gap cost $g(P)$.

Let $|P|$ denote the number of comparisons in P. The *set cost* is the cost of encoding the sets A_s and A_t involved in each comparison:

$$s(P) = |P| \left(\log_2 k + \log_2(k-1)\right). \tag{3.1}$$

The *gap cost* is the cost of encoding all the gaps except for the largest gap in each set. More formally, let $g_s[0], \ldots, g_s[p_s]$ denote the gaps in A_s for P, including the "end gaps" before the first compared element and after the last compared element in A_s. Then

$$g(P) = \sum_{s=1}^{k} \left(\sum_{i=0}^{p_s} \lg g_s[i] - \max_{0 \le i \le p_s} \lg g_s[i] \right). \tag{3.2}$$

Finally, the cost of P is $c(P) = s(P) + g(P)$.[4]

Now we claim that the described encoding is optimal: if we fix a language for encoding all $\emptyset$-proofs, then on average, a $\emptyset$-proof P requires at least $c(P)$ bits to be encoded in this language.

THEOREM 3.1. *Given any $\emptyset$-proof P for an instance I, there are $2^{c(P)}$ $\emptyset$-proofs (one of which is P) for instances with the same signature (i.e., set sizes) as I. Furthermore, each of the $\emptyset$-proofs has $|P|$ comparisons and cost at most $c(P)$, and no two of the $\emptyset$-proofs apply to a common instance.*

Proof. First let us give a construction of $\Omega\left(2^{g(P)}\right)$ different $\emptyset$-proofs based on a proof P. Let $g_s[i]$ be defined as in the definition of $g(P)$. We decrease every gap, except the largest gap in each set, to any amount less than or equal to the original gap in P. The pairing between elements stays the same; we simply move the compared elements. To compensate for these shrinking gaps and to keep the signature the same, the

[4] We ignore the cost of specifying relative to which side each gap is taken, that is, the location of the largest gap in each set. This can be encoded in $\sum_{s=1}^{k} \lg p_s$ bits, which is a negligible lower-order term: more bits are necessary just to encode the instance signature.

largest gap in each set grows, and hence remains largest. These modified gap sizes induce moved positions of the compared elements.

Because we only decrease gap sizes, except for the largest gap in each set which does not affect $g(P)$, the gap cost of any constructed proof is at most the gap cost of P. Furthermore, the number of comparisons in the proof and the number of sets does not change, so $s(P)$ does not change. Hence, the total cost of any constructed proof is at most $c(P)$.

Now for each $g_s[i]$, except the largest in each set, we have $g_s[i]$ choices for a new gap size. Therefore, the number of proofs constructed using this technique is

$$\prod_{s=1}^{k} \prod_{i=0}^{p_s} (1 + g_s[i]) \ / \max_{0 \le i \le p_s} (1 + g_s[i]) \, ,$$

which is $2^{g(P)}$, i.e., the exponentiation of Equation (3.2).

Next, we make some independent choices to improve the bound by a factor of $2^{s(P)}$, for a total of $2^{c(P)}$. Fixing the gap structure, that is, the collection of compared elements, the comparisons of a $\emptyset$-proof can be chosen as follows. Pick a set A_s such that the smallest so-far uneliminated element $A_s[i]$ is compared in P (in particular, A_s cannot be entirely eliminated yet). Pick another set A_t that is not yet entirely eliminated, and let $A_t[j]$ be the smallest so-far uncompared element in A_t that is compared in P. Then choose the next comparison to be $(A_t[j] < A_s[i])$.

The number of choices for the comparisons is somewhat less than $2^{s(P)}$, because of the constraints on the sets A_s and A_t. However, this reflects the sloppiness in our definition of $s(P)$: indeed, as indicated above, not all sets can be involved in a comparison at a given point, given the gap structure. Hence, the encoding and definition of $s(P)$ can be optimized so that the number of proofs generated is precisely $2^{s(P)}$. For simplicity of exposition, we leave $s(P)$ as the overapproximation in Equation (3.1). □

It now makes sense to talk about *optimal* proofs, that is, proofs with minimum cost. This minimum cost is called the *difficulty*, $\mathcal{D}$, of the instance. This terminology is motivated by the following superficially trivial result:

COROLLARY 3.1. *Any algorithm for the intersection or difference problem requires at least $\mathcal{D}$ comparisons in the average case.*

Proof. Certainly any correct algorithm must understand what the intersection is, and hence the comparisons it makes must form a proof P for the intersection problem. This only proves a lower bound of $|P|$, or the smallest possible number of comparisons in a proof for the intersection problem. It is not necessary that the algorithm discover an encoding of P, one bit per comparison. Instead, an algorithm may discover a collection of proofs for the instance. Potentially, this collection could be encoded in fewer bits than any individual proof (such as P). So the only lower bound we could prove from encoding optimality is $c(P)$ minus the logarithm of the number of proofs for the instance.

However, the last part of Theorem 3.1 gives us what we need: any two of the $2^{c(P)}$ $\emptyset$-proofs do not apply to a common instance. So only one of these proofs can be in the discovered collection. Hence, the algorithm must truly distinguish between the $2^{c(P)}$ proofs. In the best case, each comparison halves the search space. Therefore, at least $c(P)$ comparisons are needed. □

It turns out that another important measure on proofs is the gap cost $g(P)$. The minimum gap cost, denoted by $\mathcal{G}$, will show up in the scaled running time of an optimal adaptive algorithm for the intersection and difference problems. Note that it is possible for the optimal gap cost $\mathcal{G}$ to only be realized by nonoptimal proofs, that is, proofs with total cost higher than $\mathcal{D}$.

3.2 Encoding Union Proofs. Recall from Lemma 2.4 that all proofs for the union problem are roughly the same: they can only differ in what trees are formed by the equality comparisons, and by adding extra (useless) comparisons. Thus, instead of encoding a particular proof, we can consider encoding all proofs for the given instance. These two encoding problems are equivalent for the union problem, because given all proofs for an instance we can certainly find a canonical one, and furthermore from a single proof we could construct all other proofs for the instance. Hence, in this context we can define the cost of an *instance* instead of a proof.

The basic idea of encoding the gaps between compared elements is the same, although now the compared elements (from useful comparisons) are in fixed locations as defined by Lemma 2.4. Indeed, we will use exactly the same method to encode inequality comparisons. What differs is the way in which we encode equality comparisons. For each value v that occurs in multiple sets $A_{s_1}, \ldots, A_{s_m}$, we must encode the set numbers $s_1, \ldots, s_m$. The locations of v within the sets is already specified by the gap lengths. The only cost unique to the union problem is that of specifying the m out of k sets in which v occurs, namely $\log_2 \binom{k}{m}$ bits.

Encoding all comparisons as described, we obtain a formula for the length of encoding an entire proof P, or equivalently an instance I. We call this length the *cost* of I and denote it by $c(I)$. Again, we break $c(I)$ into two components, set cost $s(I)$ and gap cost $g(I)$, the latter of which is defined exactly as for intersections.

Let $\#(I)$ denote the number of distinct values in the union of I, and let $\#_I(i)$ denote the number of occurrences of the ith smallest value in the union of I. The *set cost* is the cost of encoding the sets A_s and A_t involved in each inequality comparison, and the collection of sets involved in each equality comparison.

$$s(I) = \sum_{i=1}^{\#(I)} \log_2 \binom{k}{\#_I(i)}. \tag{3.3}$$

Again we claim that the described encoding is optimal: if we fix a language for encoding all instances, then on average, an instance I requires at least $c(I)$ bits to be encoded in this language.

THEOREM 3.2. *Given any instance I, there are $2^{c(I)}$ pairwise distinct instances (one of which is I) with the same signature as I. Furthermore, each of the instances has cost at most $c(I)$.*

The cost of an instance is also called the *difficulty*, $\mathcal{D}$, of the instance. Unlike intersections, there are no possible tricks with encoding a collection of proofs instead of a single one in order to save bits, so the information theoretic lower bound is immediate:

COROLLARY 3.2. *Any algorithm for the union problem requires at least $\mathcal{D}$ comparisons in the average case.*

4 Further Lower Bounds on Finding Proofs

Corollaries 3.1 and 3.2 give an information theoretic lower bound, denoted $\mathcal{D}$, on the running time of any adaptive algorithm. The scaled running time (running time divided by $\mathcal{D}$) must therefore be $\Omega(1)$. For the union problem, we will in fact be able to find an algorithm with scaled running time $\Theta(1)$. For the intersection and difference problems, however, we are not so fortunate. A stronger worst-case lower bound of $\Omega(k\mathcal{G}/\mathcal{D})$ holds for the scaled running time of any adaptive algorithm. (Recall that k denotes the number of sets in the instance.) In other words, if we optimize scaled running time, then the running time is $\Omega(k\mathcal{G})$ in the worst case.

Stated differently, this section proves

THEOREM 4.1. *Given positive integers k, p, and g ($p \leq g$), and given an algorithm for finding $\emptyset$-proofs for the intersection problem, there is a collection of k sets having a p-comparison $\emptyset$-proof with cost $O(p \log_2 k + g)$, such that every $\emptyset$-proof has gap cost $\Omega(g)$, and the algorithm takes $\Omega(kg)$ time on this input. In particular, $\mathcal{D} = O(p \log_2 k + g)$ and $\mathcal{G} = \Omega(g)$, so the scaled running time is $\Omega(k\mathcal{G}/\mathcal{D})$ for this instance.*

The basic idea is to construct a parameterized class of instances, and have an adversary pick a bad instance for the algorithm. Let $\ell_1, \ldots, \ell_p$ be positive integers summing to g, such that each is either $\lfloor g/p \rfloor$ or $\lceil g/p \rceil$. An ℓ_i will represent the ceiling of the lg of a gap in the $\emptyset$-proof.

First let us describe the parameters for an instance. Pick $p+1$ "magic" values $m[0] < \cdots < m[p]$. Pick a sequence $s_0, \ldots, s_p$ of set numbers such that $s_{i-1} \neq s_i$ for all $1 \leq i \leq p$. Furthermore, each $s \in \{1, \ldots, k\}$ must occur at least every $2k$ elements in the sequence. One way to do this is to concatenate several permutations of $\{1, \ldots, k\}$, chosen randomly such that the first value of one permutation is different from the last value of the previous permutation. Finally, pick integers $j_s[1], \ldots, j_s[p]$ such that $\lceil \lg j_s[i] \rceil = \ell_i$ for all i and s, except for $j_{s_{i-1}}[i]$ which is defined to be zero for all i.

Then we construct an instance as follows (see Figure 1). Each magic value $m[i]$ occurs in every set except A_{s_i}; we will denote the occurrence of $m[i]$ in A_s by $m_s[i]$. In every set A_s, there are precisely $j_s[i]$ elements strictly between $m[i-1]$ and $m[i]$. In particular, $A_{s_{i-1}}$ has no elements strictly between $m[i-1]$ and $m[i]$; indeed, it also has no elements equal to $m[i-1]$.

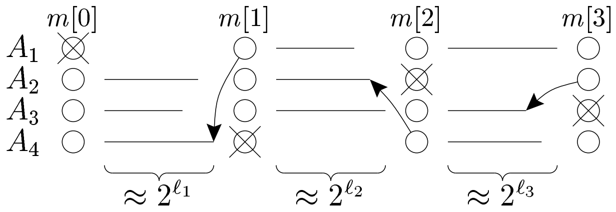

Figure 1: Illustration of lower-bound construction: Circles show magic elements, and crossed-out circles indicate missing magic values. Arrows indicate comparisons that form a $\emptyset$-proof.

Next let us describe the $\emptyset$-proof. Note that there are no elements before $m_{s_0}[1]$, and hence it is minimal. Suppose in general that $m_{s_{i-1}}[i]$ is minimal. Then we can use it to eliminate all elements less than $m[i]$ in A_{s_i}. But there are no elements between $m[i]$ (inclusive) and $m[i+1]$ (exclusive) in A_{s_i}, so this elimination makes $m_{s_i}[i+1]$ minimal. This continues by induction until we find that "$m_p[p+1]$" is minimal, that is, A_p is entirely eliminated.

LEMMA 4.1. *The described proof has cost $O(p \log_2 k + g)$.*

It turns out that this is the only $\emptyset$-proof for this instance, except for two types of possible modifications, each of which increases the gap cost. As a consequence, we have the following result:

LEMMA 4.2. *Every $\emptyset$-proof for the described instance has gap cost $\Omega(g)$.*

Finally, we can show a lower bound on the running time of the algorithm. The algorithm is allowed to know the magic values $m[1], \ldots, m[p]$, as well as $\ell_1, \ldots, \ell_p$, that is, the approximate gap sizes. The algorithm does not know the exact gap sizes (the $j_s[i]$'s), nor the numbers s_i of the sets missing the magic values.

LEMMA 4.3. *The algorithm must determine the s_i's and $j_{s_i}[i-1]$'s, independently of each other.*

Hence, the algorithm's job reduces to p independent subjobs, each of the following form: given k sorted sets, each of unknown size whose $\lceil \lg \rceil$ is ℓ_i, find the unique set whose last element is not magic. We need the following observation, which to our knowledge has not appeared before.

LEMMA 4.4. *Given k sorted sets, each of size n, and an element e, finding the unique set not containing e requires $\Omega(k \lg n)$ comparisons.*

Therefore, the algorithm takes $\Omega(kg)$ time, proving Theorem 4.1.

Our example relies on $(k-1)$-way repetition of elements. A similar argument shows that there exist *pairwise-disjoint* instances in which the scaled running time is $\Omega(k\mathcal{G}/\mathcal{D})$. This result holds provided $k = o(g_i/\log g_i)$ for each of the gaps g_i.

5 Finding Proofs

This section presents algorithms for finding proofs that match the lower bounds presented in previous sections. Specifically, for intersections and differences, we give an algorithm running in $O(k\mathcal{G})$ time; and for unions, we give an algorithm running in $O(\mathcal{D})$ time. For the intersection problem, this will solve the whole problem: it is easy to extract the intersection A from an A-proof. But for unions and differences, there is an additional problem of encoding the output. This will be addressed in Section 6.

5.1 Finding Intersection and Difference Proofs. We begin with an algorithm for finding $\emptyset$-proofs for the intersection problem, and then generalize it to A-proofs for both problems. Essentially, the algorithm "gallops" in parallel through all the sets, from both the low and high sides. Galloping consists of doubling the jump in position each iteration, until it "overshoots" the current eliminator (which will always be on the low side). Upon overshooting, the other parallel processes pause while the overshooter does a binary search to find the largest eliminatable element, and chooses the next higher element as the new eliminator.

In more detail, the algorithm works as follows.

Algorithm Empty-Intersect

- Initialize low-jump(s) and high-jump(s) to 1, and done(s) to 0, for each $s \in \{1, \ldots, k\}$.
- Initialize elim-set to 1 and eliminator to $A_1[1]$.
- For s ranging through $\{1, \ldots, k\}$ cyclicly:
  - Skip this step if $s =$ elim-set.
  - Low step:
    1. Let $p =$ done(s) + low-jump(s).
    2. If $A_s[p] \geq$ eliminator (we overshot),
       (a) Binary search in the interval [done(s) + $1, p$] to find the smallest p' with $A_s[p'] \geq$ eliminator.
       (b) If $p' - 1 >$ done(s), add ($A_s[p'-1] <$ eliminator) to the proof.
       (c) Set done(s) to $p'-1$, and low-jump(s) to 1.
       (d) Set elim-set to s, and eliminator to $A_s[p']$.
    3. Otherwise, double low-jump(s) and set done(s) to p.
  - High step:
    1. Let $p = n_s + 1 -$ high-jump(s).
    2. If $A_s[p] <$ eliminator (we overshot),
       (a) Binary search in the interval $[p, n_s]$ to find the largest p' with $A_s[p'] <$ eliminator.
       (b) If $p' >$ done(s), add ($A_s[p'] <$ eliminator) to the proof.
       (c) If $p' = n_s$, stop.
       (d) Set done(s) to p', and low-jump(s) to 1.
       (e) Set elim-set to s, and eliminator to $A_s[p'+1]$.
       (f) Reset high-jump(s) to one.
    3. Otherwise, double high-jump(s).

Note that at any point in time, $A_s[i]$ is eliminated exactly when $i \leq$ done(s).

Now we claim that the algorithm matches the lower bound from Section 4, that is, has scaled running time $O(k\mathcal{G}/\mathcal{D})$.

THEOREM 5.1. *Algorithm Empty-Intersect runs in $O(k\mathcal{G})$ time, and makes at most $8k\mathcal{G}$ comparisons.*

Proof. Suppose we binary search inclusively between done(s) + 1 and $p =$ done(s) + low-jump; the high case is similar. This takes at most $t = \lg$ low-jump comparisons. But low-jump $= 2^i$ where i is the number of iterations we have already executed on this side of A_s since the last overshooting. Hence, $t = i + 1$, so we can charge the binary-search time t to these $i > 0$ iterations, and no other overshooting iterations will charge to the same iterations (as we now reset the jump). This amortization is the source of one factor of two in the comparison bound.

Let P be a proof with gap cost $\mathcal{G}$, and let it be ordered low-to-high by Theorem 2.1. Let $c = (A_s[i] < A_t[j])$ be the first comparison in P. We want to evaluate the number of iterations the algorithm spends to eliminate $A_s[i]$. The low and high parts of the algorithm run effectively in parallel; this causes the second factor of two in the comparison bound.

There are two main cases. In the first, the gap below $A_s[i]$ (size g) is not the largest gap in A_s. Ignoring the binary-search cost as described above, galloping effectively occurs in lock-step parallel over the sets. Local to A_s, the number of comparisons for galloping to $A_s[i]$ or beyond is $\lg g$. The other sets have had at most the same number of iterations, thus adding a factor of k.

In the second case the gap below $A_s[i]$ is the largest in A_s. If the sum of the other gaps' sizes is at least g, then we can charge the cost ($\lg g$) of running through the largest gap to the other gaps in A_s. These gaps will not be charged to again, because there is only one largest gap in A_s that the gap cost does not count (i.e., for which we must avoid paying directly). If, on the other hand, the other gaps' sizes sum up to some value h less than g, then the high step finds $A_s[i]$ from the high side in $\lg h$ iterations. But $\lg h$ is at most the sum of the lgs of the other gaps in A_s, so again we can charge to these gaps. These amortizations add the last factor of two to the comparison bound.

Therefore, eliminating $A_s[i]$ takes $\lg g$ amortized comparisons, unless g is the largest gap in its set, in which case it takes zero amortized comparisons. By induction, this holds for all future comparisons in P. $\square$

It may be possible to improve the factor 8 in the comparison bound down to 4 (plus lower order terms), using more sophisticated galloping techniques (see e.g. [3]) that find an integer x in roughly $\log_2 x + \log_2 \log_2 x + \log_2 \log_2 \log_2 x + \cdots + 1 + \log_2^* x$ comparisons, instead of the method presented which uses $2 \log_2 x + O(1)$ comparisons.

Finally, let us turn to the case where the intersection is not necessarily empty. Recall that a proof for either the intersection or difference problem must demonstrate the intersection elements (by making $k-1$ equality comparisons each), and forms a $\emptyset$-proof on each of the remaining subinstances between the partition points of the intersection elements.

Corollary 5.1. *A proof for the intersection or difference problem for k sorted sets can be computed in $O(k\mathcal{G})$ time and at most $8k\mathcal{G}$ comparisons.*

Proof. This follows from a simple modification to Algorithm Empty-Intersect above, namely whenever a comparison with the eliminator returns "equal," stop galloping in that set and increase the occurrence count of the eliminator. If the occurrence count reaches k, output the eliminator as part of the intersection, add $k-1$ appropriate comparisons to the proof, and take the eliminator's successor as the new eliminator. $\square$

Corollary 5.2. *The intersection of k sorted sets can be computed in $O(k\mathcal{G})$ time and at most $8k\mathcal{G}$ comparisons.*

Note that the described algorithms perform just as well on B-trees or related structures. We only need to start with the leftmost and rightmost leaves, and then gallop inwards from each side. This can be easily performed by traversing the parent and child pointers in a B-tree, with only a constant-factor overhead.

5.2 Finding Union Proofs. Essentially, the algorithm maintains a priority queue over the sets, where the priority of a set is the value of its smallest (unused) element. In the case of an inequality comparison, the algorithm takes the next-to-smallest element and finds where it fits in the set containing the smallest element, by galloping through the set. In the case of an equality comparison, Delete-All-Min matches and returns multiple elements. In both cases, the minimum elements are removed from consideration and the priority queues are updated.

In more detail, the algorithm works as follows.

Algorithm Union-Proof

- Initialize the priority queue Q with the smallest element of every set.
- Until all elements have been (conceptually) removed:
  1. Let M = Delete-All-Min (Q).
  2. If $|M| = 1$, in particular $M = \{A_s[i]\}$:
     (a) Let m' = Find-A-Min (Q), that is, one of the minima.
     (b) Gallop in A_s to find where m' fits.
     (c) Add the smallest element $A_s[j]$ greater than or equal to m' to Q.
     (d) If $m' < A_s[j]$, add $(m' < A_s[j])$ to the proof.
     (e) Remove all elements in A_s less than m'.
  3. Otherwise ($|M| > 1$):
     (a) For each $A_s[i] \in M$:
         i. Remove $A_s[i]$ from A_s.
         ii. Add $A_s[i+1]$ to Q.
     (b) Add equality comparisons to form a spanning tree of the elements of M.

Now we claim that the algorithm matches the lower bound from Section 3.2.

THEOREM 5.2. *Algorithm Union-Proof runs in $O(\mathcal{D})$ time.*

6 Computing the Answer

For both the union and difference problems, finding a proof is not the whole story. The problem asks for the actual answer, the union or difference of the sets, not just an understanding of the answer which is given by a proof. This understanding does, however, specify the *ranges* of elements in the answer. For example, a proof for the union problem encodes the total order of the answer. Thus, the output could be encoded in the following form:

take the first 12 elements from A_2
take the first 3 elements from A_5
take the next 11 elements from A_2
skip the first element in A_3
⋮

However, this kind of output encoding is unsatisfactory, because it is not in the same form as the input. In particular, if we want to use the result of this union operation as the input to another operation, e.g., an intersection, then it must be in a usable form for the latter operation. It is difficult to gallop in a set described as above.

Thus we need a better output encoding, one that matches the input encoding. We cannot simply use arrays for both input and output, because then writing down the answer beats the purpose of finding proofs adaptively (as unions and differences are typically very large). We turn instead to the most natural alternative: a balanced search tree structure. Specifically, we focus on *B-trees* as a commonly used representative of this class. As mentioned in Section 5.1, it is easy to gallop in such structures, paying only a constant factor of overhead in time. Furthermore, in text databases, input sets are often stored as B-trees to begin with.

The goal, then, is to build another B-tree representing the union or difference of a collection of sets. We assume that the sets cannot be modified; for example, in a database system, while the input sets for this operation may be stored in memory and thus there are copies stored on disk, the sets in memory often serve as a cache, whose modification would require expensive reloading from disk. The remaining freedom in encoding is subtle: we can use entire subtrees from existing B-trees for building new B-trees. In other words, constructed B-trees can have child pointers to nodes in other existing B-trees. This gives us a *persistent* mechanism for augmenting old trees.

This level of flexibility will be enough to allow us to construct B-trees representing the answer in the same time as for computing a proof representing the answer. In the next two subsections, we consider each of the difference and union problems in turn, and show how to build a B-tree assuming that we already know a proof.

6.1 Computing Differences. The situation for the difference problem is fairly simple. A proof gives us the intersection of the sets, and it remains to remove those elements from A_1 to obtain the result. Thus, we want a persistent B-tree structure that supports deletions. In other words, given a B-tree T and elements $x_1, \ldots, x_m$, we would like to be able to construct a new B-tree with contents $T - \{x_1, \ldots, x_m\}$, without modifying T but by reusing nodes of T. This can be done using a standard persistence trick: perform the standard B-tree multidelete [10], but whenever a node is modified, first make a copy of the node and then modify the copy instead. This proves the following theorem:

THEOREM 6.1. *The difference of k sorted sets stored in read-only B-trees can be computed as another B-tree in $O(k\mathcal{G})$ time.*

6.2 Computing Unions. The situation for unions is more difficult. There are two steps. First, we carve each tree according to the partition defined by the proof. Second, we merge the pieces in the appropriate order to form the union. Both of these operations are done persistently. As we do not have room for details, we simply state the results.

LEMMA 6.1. *Given a read-only B-tree T and a collection of values $a_1, \ldots, a_m$ at which to cut it, the resulting B-trees $T_0, T_1, \ldots, T_m$ can be computed in $O(\sum_{i=0}^{m} \mathrm{height}(T_i))$ time.*

The algorithm for this lemma is a generalization of procedure DIVIDE of Aho et al. [1, p. 157] to support multiple cut points. The main difficulty is in proving the time bound.

LEMMA 6.2. *Given a sequence $T_1, \ldots, T_m$ of read-only B-trees, their concatenation can be computed (as a B-tree) in $O(\sum_{i=1}^{m} \mathrm{height}(T_i))$ time.*

Again, the algorithm is essentially a generalization of procedure IMPLANT of Aho et al. [1, p. 153] to support more than two trees. This cannot be done by repeatedly calling IMPLANT, because that may cause the heights of the trees being concatenated to grow significantly. Instead, we use a priority queue to order the trees appropriately. This takes only constant extra time per merge as the universe of heights is small.

THEOREM 6.2. *The union of k sets stored as read-only B-trees can be computed as another B-tree in $O(k\mathcal{G})$ time.*

We note that our adaptive algorithm for the union problem has been described for the special case of two sets [4, 10, 11]. Our new results are the generalization to multiple sets (which can offer a significant improvement in adaptive performance) and the matching lower bounds.

7 Conclusion

Perhaps the most interesting contribution of this work is our framework for designing and analyzing adaptive algorithms under the comparison model. The essential idea is to perform a worst-case analysis on the *scaled* running time instead of the usual running time. We defined the scaled running time to be the ratio of the running time to the difficulty of the instance. This difficulty of course depends on the problem, but a natural metric is the information theoretic lower bound.

Using this framework, we proved matching upper and lower bounds on finding intersections, unions, and differences of sorted sets. Specifically, for unions, the scaled running time is $\Theta(1)$. For intersections and differences, the scaled running time is $\Theta(k\mathcal{G}/\mathcal{D})$, where k is the number of sets, $\mathcal{G}$ is the so-called "gap cost," and $\mathcal{D}$ is the difficulty of the problem. In other words, if we take the worst-case performance relative to the scaled running time, then the best possible running time is $\Theta(k\mathcal{G})$.

For the union problem, or when the number of sets is constant, this is a truly ideal situation: the running time is proportional to the information theoretic lower bound. For asymptotically many sets, the running time for intersections and differences is away from this bound by a reasonable factor that is somewhat less than k; and furthermore it is impossible to achieve better than this factor in the worst case. In general, we expect our algorithms to be practical for evaluating boolean queries in text retrieval systems. Ongoing work on arbitrary query expressions, involving a mix of unions, intersections, and differences of sets, builds upon all the results outlined here.

The theme of this work is the exploitation of nonuniformity in data. A situation in which we might not expect an improvement is an instance with k sets each containing n elements chosen uniformly at random from $(0, 1)$. We can show that the expected number of comparisons in the smallest $\emptyset$-proof is about $n/\ln k$. As a consequence, our algorithm takes $O(nk \log\log k/ \log k)$ expected comparisons, which is asymptotically better (in terms of k) than previous algorithms which look at all nk of the elements.

Acknowledgments. We thank Ming Li for helpful discussions. This work was supported by NSERC.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] R. Baeza-Yates. *Efficient Text Searching.* PhD thesis, U. Waterloo, 1989.

[3] J. L. Bentley and A. C.-C. Yao. An almost optimal algorithm for unbounded searching. *IPL*, 5(3):82–87, Aug. 1976.

[4] S. Carlsson, C. Levcopoulos, and O. Petersson. Sublinear merging and natural mergesort. *Algorithmica*, 9:629–648, 1993.

[5] V. Estivill-Castro and D. Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys*, 24(4):441–476, Dec. 1992.

[6] W. Frakes and R. Baeza-Yates. *Information Retrieval.* Prentice Hall, 1992.

[7] F. K. Hwang and S. Lin. A simple algorithm for merging two linearly-ordered sets. *SICOMP*, 1(1):31–39, 1980.

[8] D. E. Knuth. *The Art of Computer Programming*, vol. 3. Addison-Wesley, 1968.

[9] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searchs. In *Proc. 1st Symp. Discrete Algorithms*, pp. 319–327, 1990.

[10] K. Mehlhorn. *Data Structures and Algorithms*, vol. 1, pp. 240–241. Springer-Verlag, 1984.

[11] A. Moffat, O. Petersson, and N. C. Wormald. A tree-based Mergesort. *Acta Informatica*, 35(9):775–793, Aug. 1998.