Optimized Union of Non-disjoint Distributed Data Sets *

Itay Dar Tova Milo Elad Verbin
School of Computer Science, Tel Aviv University
{daritay,milo,eladv}@post.tau.ac.il

ABSTRACT

In a variety of applications, ranging from data integration to distributed query evaluation, there is a need to obtain sets of data items from several sources (peers) and compute their union. As these sets often contain common data items, avoiding the transmission of redundant information is essential for effective union computation. In this paper we define the notion of *optimal union plans* for non-disjoint data sets residing on distinct peers, and present efficient algorithms for computing and executing such optimal plans.

Our algorithms avoid redundant data transmission and optimally exploit the network bandwidth capabilities. A challenge in the design of optimal plans is the lack of a complete map of the distribution of the data items among peers. We analyze the information required for optimal planning and propose novel techniques to obtain compact, cheap to communicate, description of the data sources. We then exploit it for efficient union computation with reasonable accuracy. We demonstrate experimentally the superiority of our approach over the common naive union computation, showing it improves the performance by an order of magnitude.

1. INTRODUCTION

In a variety of applications, ranging from data integration to distributed query evaluation, there is a need to obtain sets of data items from several sources (peers) and compute their union. As these sets often contain common data items, a naive algorithm where each peer sends to the target peer its full set of items (or even just the ids of all items) is wasteful communication-wise, having common items (ids) sent multiple times. The goal of this paper is to develop efficient algorithms that avoid such redundant data transmission and minimize the time required to compute the union of non-disjoint data sets residing on distinct peers.

To situate and motivate the problem that we study here, we next present a simple example that illustrates its practical origin and explains how a good solution to the problem addressed here can contribute to better query evaluation in today's Web-oriented distributed environment.

*The research has been partially supported by the European Projects EDOS and MANCOOSI and by the Israel Science Foundation.

Example. Consider a distributed full-text index of a given set of documents, based, for instance, on a Distributed Hash Table (DHT) [16, 28]. For every word w appearing in the documents, the index contains some peer p_w holding the urls of the documents that include the word. One can consider here *conjunctive* and *disjunctive* queries (or some combinations). Given a set of words $w_1, \ldots, w_n$, a *conjunctive* (resp. *disjunctive*) query returns the documents where *all* (*some of*) the words appear. Much research has been devoted to the efficient computation of conjunctive queries [2, 17]. In contrast, this paper focuses on disjunctive queries and is the first, to our knowledge, to propose efficient algorithms for their computation.

As a simple example for a disjunctive query, consider a user interested in finding documents where some word w_1 or any of its synonyms $w_2, w_3, \ldots, w_n$ appear. The answer for the query is the union of the url sets held by the peers $p_{w_1}, \ldots, p_{w_n}$. Observe that the sets are likely to have significant overlap – a document containing a word w_i often also includes several of its synonyms. For example, assume that some subset D of the documents contains the three words w_1, w_2 and w_3. The urls of the documents in D thus all appear in p_{w_1}, p_{w2} and p_{w_3}. It is naturally wasteful to transmit the same urls several times over the network, in particular if the set D is large. A better performance would be obtained if we could split D into three disjoint subsets D_1, D_2, D_3 and have each of the peers p_{w_i}, $i = 1, 2, 3$, send only D_i. Before explaining how such a split can be accomplished, it is important to note that the optimal way to split the sets depends on how many additional (disjoint) data items each peer needs to send, as well as on network constraints such as the download and upload rates of the involved peers. For instance if p_{w_1}, p_{w_2}, p_{w_3} hold no urls other than those in D and have equal bandwidth, the best performance would be obtained by having D_1, D_2, D_3 be of equal sizes, each containing a third of D. But if p_{w_1} has half as much bandwidth as the other two peers, it is best for D_1 to have size half of D_2 and D_3 (i.e. to contain $1/5$th of D while D_2, D_3 each hold $2/5$th). This split is also optimal if p_{w_1} has bandwidth equal to the others but contains an additional set D' of urls of size $1/5$ of D.

In the sequel, given a peer that is interested in the union of some non-disjoint data sets residing on distinct peers, we use the term *union plan* to denote a set of instructions that determines which items should be sent by which peer to which peer and when, so that each data item is eventually transmitted (at least once) to the target peer and all network constraints are respected. (A formal definition is given in the following section). An *optimal* such plan is one where the time it takes for the target peer to obtain the full

union set is minimal. We will see below that for every union request there exists an optimal plan which identifies peers holding common item sets, splits those sets into disjoint subsets, and has each peer transmitting its (disjoint set of) items to the target peer in a manner that best exploits the network capabilities.

A key challenge in the design of such an optimal union plan is the lack of a complete map of the distribution of the data items among peers. A peer typically knows what items it holds but rarely knows what is held on each of the other peers. To analyze what information is needed for optimal planning we first consider an "ideal" scenario where each peer knows precisely what is held by other peers, and provide an efficient algorithm for computing an optimal union plan in this setting. Gathering precise such information is costly. However, we will see that if one is willing to slightly compromise the optimality of the plan or the accuracy of the union result, it is possible to get a compact, cheap to communicate, description of the data sources that allows for efficient union computation with reasonable accuracy. Indeed, in many real life scenarios, some imprecision in the union result is tolerable. This is the case ,for instance, when the result is used to compute some statistics/approximated aggregate function on the queries data. Similarly, if the data sets to be unioned originate from a Web search, hence are typically incomplete to start with, and important information is likely to be repeated in several distinct items, (e.g. important news are likely to be reported in several news articles, important web sites may have several of their main pages occurring in the result), the risk of missing some occurrences may be acceptable.

Before presenting our algorithms, let us briefly consider a related problem. More related work appears in Section 6.

Sets reconciliation. A restricted version of the problem that we study here, called *sets reconciliation*, was introduced in [24]. It focuses on computing the union of a *pair* of item sets. Given a pair of hosts A and B, each with a set of length b bit strings, these works propose an array of algorithms to allow both hosts to determine the union of the two sets with a minimal amount of communication. Both exact and approximated computations were considered, with applications to a variety of domains including PDA synchronization, routing table maintenance, and gossip-based content delivery protocols[23]. A key difference from the present work is that we consider here the union of an *arbitrary, possibly large, number of sets* held on distinct peers, and the result is required by a different *target peer*. While some of the techniques used for pairwise sets reconciliation are naturally still useful here, we will see that the need to handle multiple sets introduces new challenges, requiring a tighter, more global coordination of the peers. For instance, while network constraints do not play a central role in existing algorithms for sets reconciliation (data is simply sent from one peer to another at the maximal rate allowed by the upload and download rates of the peers), when several peers wish to send data to a single target peer, a more careful bandwidth distribution is required to guarantee optimal performance.

Results. The contributions of this paper are the following.

- We introduce a simple generic model for describing union plans and formally define the notion of an optimal plan.
- To understand what information (on the content of the peers and the correlations among them) is needed to derive an optimal union plan, we first consider a scenario where full knowledge about the distribution of data items among peers is given and design an efficient algorithm for computing optimal union plans in such setting.
- As gathering such precise data is costly, we propose two heuristics, one for reducing the number of correlations among peers to be considered (at the expense of compromising the optimality of the plan) and the second for gathering compact, cheap to communicate, description of the data sources (at the expense of compromising the accuracy of the union result).
- To evaluate the effectiveness of our algorithms and heuristics, we have implemented them and conducted an experimental comparative study of their performance. Our experiments show that, compared to a naive union, our techniques can improve the performance by an order of magnitude, yielding only very marginal error.

Problem setting. We consider here multi-set union with the number of sets in the union ranging from two to hundreds. We assume that the query (target) peer knows the set peers and that can be direct communication between any pair of peers (set or query). This is a common assumption in the implementation of DHT-based applications where an overlay network (e.g. ring- or tree-shaped) is used to locate peers, but once a required peer is identified, the data transfer to the target is performed in a direct manner [20]. We focus here on ad hoc query answering, where the response time is expected to be short. Consequently, it is likely for the bandwidth constraints not to change much throughout this short time interval, and this is what we assume here. We also assume that failures are handled by the underlying data communication layer (e.g. the DHT layer in the above example).

The paper is organized as follows. In Section 2 we define optimal union plans and highlight a central property that is used in Section 3 to develop an efficient algorithm for deriving optimal plans. In Section 4 we present two heuristics for gathering compact information that allows for the derivation of (approximated variants of) such plans. Section 5 describes the algorithms implementation and experiments. Finally, Section 6 concludes with related work.

2. PRELIMINARIES

We start by presenting the basic formalisms used throughout the paper and introduce the notion of (optimal) union plan. We then present in the next section an algorithm for computing such plans.

To simplify we assume that all data items are of approximately the same size. Similar development can be done for the case where items of different types have different sizes. Time is viewed in a discrete manner and each time unit represents a communication round. A point in time is represented by a positive integer describing the number of time units that had passed since the beginning of the communication. Our measurement of cost for a union computation is in terms of the time it takes for the target peer to receive all data items. The bandwidth constraints of a given peer are modeled by the number of items it can send/receive in one time unit.

To make this more precise we use the following notation. Let $\mathcal{D}$ be a domain of data items and let P be a set of peers. The number of items that a peer $p \in P$ can receive (resp. send) in one time unit is denoted $download(p)$ (resp. $upload(p)$). For simplicity we assume that $download(p)$ and $upload(p)$ are positive natural numbers. The set of items from $\mathcal{D}$ that a peer p holds is denoted $items(p)$. The union of the item sets held by peers in P is denoted $items(P)$, namely $items(P) = \bigcup_{p \in P} items(p)$. For a set s (of items or peers), we use below $|s|$ to denote the set's cardinality.

A *union plan* is a set of instructions that describes which items are sent by which peer to which peer and when, so that all network constraints are respected and each data item in $items(P)$ is transmitted eventually, (at least once), to the target peer. An *optimal*

such plan is one where the time it takes for the target peer to obtain the full union set is minimal.

DEFINITION 2.1. *Given a set P of peers and a distinct target peer p_0, a* union plan *U (for P and p_0) is a set of quadruples of the form $(t, p, \hat{p}, d)$, where t is a time point, $p, \hat{p} \in P$ are two peers, called the* sending *and the* receiving *peers, resp., and $d \in \mathcal{D}$ is a data item sent from p to $\hat{p}$ at time t. We assume that the following conditions hold.*

- *The upload (resp. download) bandwidth constraints of all peers are respected. Namely for all t and p, U contains at most $upload(p)$ (resp. $download(p)$) tuples with time t and having p as the sending (resp. receiving) peer.*
- *A peer can send only data items that it originally held or had previously received. Namely $\forall (t, p, \hat{p}, d) \in U \quad d \in (items(p) \cup \{d' \mid \exists t' p',\ (t', p', p, d') \in U, t' < t\})$*
- *All data arrives at the target peer, i.e. $\{d \mid \exists t\, p\ (t, p, p_0, d) \in U\} = items(P)$.*

We use $time(U)$ to denote the maximal time point in the plan U. U is called an optimal *union plan if there is no other U' with $time(U') < time(U)$.*

In general, a union plan allows items to be sent indirectly to the target peer via other peers, and the target may receive the same item several times. A *direct* union plan is one where all items are sent directly to the target, namely for all $(t, p, \hat{p}, d) \in U$, $\hat{p} = p_0$. A *non redundant* plan is one where each item is received by the target exactly once. The following Proposition shows that when searching for an optimal union plans it suffices to consider *direct*, *non redundant* plans.

PROPOSITION 2.2. *For every set P of peers and a target peer p_0, there exists an optimal union plan that is direct and non redundant.*

PROOF. (Sketch) Clearly, any optimal plan U can be transformed into a non redundant plan U' by simply removing from U the transmission, to the target, of all items that it had previously received. To show that each non redundant plan can be turned into a non redundant direct one, we first observe that non redundant union plans can be "minimized" by (1) removing unnecessary transmissions of data items d from a peer p to peer p' if p' already holds d, and then (2) removing (recursively) all the transmissions of items to peers that do not send the items further. It is easy to see that in such a minimized optimal plan U", the subset $I_p \subseteq items(p)$ of items that each peer p sends are all disjoint. (In addition to this subset I_p, p may send additional items that it received from other peers). So the plan U" must also be an optimal union plan for this set of disjoint I_p's. (Or else we could have used their optimal union plan to obtain a better plan for our data, in contradiction to the optimality of U"). To conclude the proof, we use the fact (proved in Theorem 3.5 in the following section) that for *disjoint item sets* there always exist a *direct* optimal plan. As this plan is equivalent time-wise to the plan U" (both being optimal) we can replace U" by this direct plan to obtain a direct, non redundant optimal plan for our original P and p_0. □

The above proposition is important since it implies that all we need to do, in order to find an optimal plan for P and p_0, is to find for each peer $p \in P$ a subset $I_p \subseteq items(p)$ of its items such that (1) the subsets chosen for the different peers are pairwise disjoint, (2) together they include all the items in P, and (3) the optimal union plan for these chosen disjoint I_p's is of minimal time.

The algorithms presented in the following sections follow precisely this route.

3. OPTIMAL UNION PLANS

Before presenting our solution, note that the problem that we study here can be viewed as some sort of *scheduling problem* [26] where each data is a task which can be performed only by the peers holding it, with the network constraints acting as scheduling constraints. A common technique for solving scheduling problems is by reduction to a network flow problem [26]. Indeed, the first ingredient of our solution, presented below, uses network flow. A key challenge however that needs to be addressed in our setting is the large number of data items. An algorithm polynomial in the number of tasks (data items), which would be considered reasonable for standard scheduling problems, is impractical here. An additional difficulty here comes from the lack of a global map of which peers hold each data item (i.e. can perform the task).

To understand what information (on the content of the peers and the correlations among them) is needed for deriving an optimal union plan, we first consider a scenario where full knowledge about the distribution of data items among peers is given, and design an efficient algorithm for computing optimal union plans in such setting. Our algorithm consists of two steps. The first, called `AssignData`, considers data items that reside in several peers, and chooses for each item a unique peer that will send it to the target peer. For efficiency, `AssignData` does not treat each item individually but rather considers each subset of items residing on the same set of peers (and on no other peers) as an "equivalence class" and splits each class into disjoint subsets to be transmitted by the individual peers. Once the "best" assignment of items to peers has been determined, the second step, called `SendData`, determines in what order the peers should send the data they are responsible for so that the overall transmission time is minimized. We detail below each step, then prove that the resulting union plan is indeed optimal.

3.1 `AssignData`

`AssignData` uses a `CheckTime` subroutine that, given a set P of peers, a target peer p_0 and a time t, checks whether it is possible to find for each peer $p \in P$ a subset $I_p \subset items(p)$ of its items such that (1) the subsets chosen for the different peers are pairwise disjoint, (2) together they include all the items of P, and (3) the optimal union plan for the subsets is of time t or less. We call a set of I_p's satisfying the above requirements a *split* (of time t). To find an optimal union plan, `AssignData` will find the smallest t for which `CheckTime` returns a positive answer.

Let us first explain how `CheckTime` works. As mentioned above, the algorithm views each subset of items residing on the same set of peers (and on no other peers) as an "equivalence class". It reduces the problem of checking if a split of time t exists, to a Max-Flow problem [3] on a network (flow graph) whose nodes represent the equivalence classes and whose edges (and their flow capacity) represent the amount of data that can be transmitted by the peers in t time units.

Flow graph. An example flow graph is depicted in Figure 1. We first describe how such graphs are constructed, in general, and then explain this specific example. Given a set P of peers, a target peer p_0 and a time t, we construct the following flow graph. The graph has five layers. The first layer consists of a single *source* node. The second layer represents the equivalence classes of data

items, and has one node per each (non empty) set of items which all reside on the same set of peers and on no other peers. Note that the number of such nodes (equivalence classes) is bounded by $min(|items(P)|, 2^{|P|})$. The third layer represents the peers and has one node per each peer $p \in P$. The fourth layer consists of a single node representing the target peer p_0. Finally, the last layer consists of a single *sink* node.

The *source* node has outgoing edges to all the equivalence class nodes. The capacity of each such edge describes the number of items in the pointed equivalence class. Since each distinct item participates in a single such class, the total capacity of the edges is equal to the number of distinct items in P, i.e. $= |items(P)|$. Every equivalence class node has outgoing edges to each of the peers containing the class's elements. (The edge models the fact that each such peer may send elements of the class). The capacity of each edge, again, describes the number of items in the class. Every peer node has an outgoing edge to the target node. For a peer p, the capacity of the edge describes the maximal number of items that can be uploaded, from p to the target peer p_0, in t communication rounds, namely $= t \cdot (\texttt{min}(upload(p), download(p_0))$. Finally, an edge from the target to the *sink* node describes the number of items that can be received by the target peer p_0 in t communication rounds, and has capacity $t \cdot download(p_0)$.

We next run a standard Max-Flow algorithm (e.g. the push-relabel algorithm [3]) on the network, to find the maximal flow that can arrive to the sink. If it equals $|items(P)|$ (or in other words, all items sent by the source can arrive to the sink), we conclude that a split (of time t) indeed exists.

EXAMPLE 3.1. *As a simple example consider three peers, p_1, p_2, p_3, each holding a set of data items, and a peer p_0 interested in their union. The upload rate of p_1, p_2 and p_3 is 2 data items per time unit. The download rate of p_0 is 3 items per time unit. Assume that there are 150 items that reside only on p_1, 100 items that reside only on p_2 and 60 items residing only on p_3. Additionally, there are 100 items that reside on both p_1 and p_2 (but not on p_3) and 10 items that reside on all three peers. There are no items that reside only on p_1 and p_3 and also none that reside only on p_2 and p_3. The overall number of distinct items held by the three peers is thus 420.*

The network graph constructed for the peers is depicted in Figure 1. The number attached to each edge describes its flow capacity. (Ignore for now the numbers in parenthesis). Clearly, for every time $t < 140$ the maximal flow that the network can pass is smaller than 420. (This is evident by the fact that the capacity of the edge from p_0 to the sink, for $t < 140$, is strictly less than 420. For $t = 140$ a maximal flow of 420 does exist. Indeed there are several possible such maximal flows, and one of them is depicted in the Figure - the numbers in parenthesis attached to each edge in the figure describe the flow passed by the edge in one such possible maximal flow. We thus conclude that a split of time 140 exists.

Deriving a split. The split itself is easily derived from the flow assigned to the edges connecting the equivalence class nodes to the peer nodes: for every equivalence class, the flow f_p on the edge pointing to a peer p indicate how many class members are assigned to p. W.l.o.g. assume that the items in each equivalence class are ordered (otherwise we can simply apply some hash function to their ids/content and order them according to the hash value). Also assume that there is some order on peer ids. Now, for each equivalence class, the first peer in this order is made responsible for the first f_{p_1} items of the equivalence class, the second peer for the following f_{p_2} items, and so on. Consequently, the set I_p of elements that a peer p is assigned consists of the subsets, from the various equivalence classes, that were assigned to p.

EXAMPLE 3.2. *To continue with the above example, every possible maximal flow solution for the network corresponds to one possible split. For instance, for the flow assignment in Figure 1, a split is obtained by (1) assigning to p_1 (resp. p_2) half of the 100 elements that reside on both p_1 and p_2; (p_1 gets the first half of the elements and p_2 the second half, for some arbitrary order of the elements), and (2) assigning to p_3 the 10 elements that reside on all the three peers. Consequently, p_1 will be responsible for sending to p_0 200 elements, p_2 will send 150 elements, and p_3 70 elements, (all sets pairwise disjoint).*

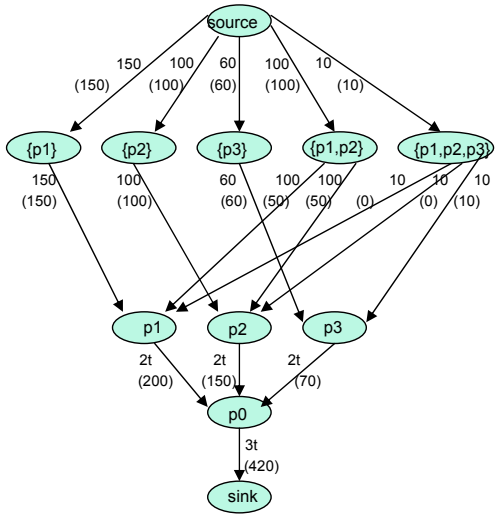

Figure 1: Max-Flow network for time t

The time complexity of the algorithm is polynomial in the size of the network (determined by the number of equivalence classes and the number of peers). Its correctness is stated below. We omit the proof for space constraints.

THEOREM 3.3. *Given a set P of peers, a target peer p_0 and a time t, `CheckTime` computes for P a split of time t iff one exists, and declares failure otherwise.*

Finding a Minimal t. To conclude we only need to find the minimal time t for which such a split exists. Observe that the value of this minimal t is bounded from below by the download rate of the target, and bounded from above by the minimum upload rate of the peers. Namely,

$$\frac{|item(P)|}{download(p_0)} \leq t \leq \frac{|item(P)|}{\texttt{min}(download(p_0), \texttt{min}_{p \in P} upload(p))}$$

Thus, to find the minimal t we can perform a binary search on this interval, running `CheckTime` for each tested t, (overall, a logarithmic number of times), and finding the minimal one for which a split exists.

3.2 `SendData`

The above algorithm tells us which (disjoint) subsets of items should be sent by each peer. Now, to derive an optimal union plan, we only need to determine in what order the peers should send this data. Or in other words, we need to derive an *optimal union plan* for the *disjoint item sets*. Before presenting our algorithm, let us first look at a simple example that demonstrates that the order in

which data is sent indeed matters. The example will also be useful to illustrate how our algorithm works.

EXAMPLE 3.4. *Continuing with the scenario of Example 3.2, the peers p_1, p_2, p_3 need to send to p_0 three disjoint sets of data items. p_1 has 200 items to send, p_2 has 150 items and p_3 70 items. Recall that the upload rate of each peer is 2 items per time unit and the download rate of the target peer p_0 is 3 items per time unit. Consider first a naive union plan U_1 where the peers transmit their data to p_0 in a simple round-robin fashion, where, in each round, the download capacity of the target is equally shared among them. The data transmission here will take 154 time units: At the first 70 rounds all peers send one item per time unit, in parallel. Then, only peers p_1 and p_2 send data, (as p_3 had finished sending all its data), sharing equally the bandwidth - in one round one peer sends 2 items and the other peer 1 item, then they alternate. This continues for 53 rounds, till p_2 also finishes sending its data. Finally, in the last 26 rounds p_1 sends its remaining data. A better planning, however, can yield better performance. Consider an alternative union plan U_2 where for the first 50 time units p_1 sends data at its full upload rate (i.e. 2 items per time unit) and p_2 at half speed (namely 1 items per time unit). At the end of this they are left with 100 items each. Now, for 20 time units they share the bandwidth equally - in one round one peer sends 2 items and the other peer 1 item, and then they alternate. At the end of this all three peers have each 70 units left, which they can send in parallel in 70 time units, yielding a total transmission time of just 140 units. In this example the time saving of U_2, relative to the previous naive plan U_1, is not too big because our example contains, for simplicity, only very few data items. In practice, as we shall see, results on real-life scenarios demonstrate significant time reduction.*

Given a set P of peers holding *pairwise disjoint* item sets, and a target peer p_0, the algorithm `SendData` depicted in Figure 2 computes an optimal union plan for P and p_0. The plan that it derives is a *direct* one, namely all peers send their data directly to the target. Its optimality (proved below) thus demonstrates that if the peers' data sets are disjoint, then there is a *direct* optimal union plan for them.

The Algorithm. To simplify the presentation we first present a simplified version of the algorithm, then explain how it can be optimized. The algorithm `SendData` works iteratively, in a greedy manner. At each iteration it identifies peers that are a bottleneck, namely where the time required to send their remaining data to the target is maximal, and allocates bandwidth for them. Since a peer p cannot upload data to the target at a rate higher than the target's download rate, we assume below, for simplicity, that for every $p \in P$, $upload(p) \leq download(p_0)$. We use the following notations. t denotes the time (initially 0) and U denotes the constructed union plan (initially empty). For a peer p, $\sharp items(p)$ denotes the number of items remaining for p to send. At the beginning of the algorithm, $\sharp items(p) = |items(p)|$ and it decreases when items are sent out. The algorithm continues as long as there are still some items left to send (line 6). The time required for a peer p to send all its remaining data to the target (assuming it is the only peer exploiting the target's bandwidth) is denoted $time\_to\_finish(p)$. Note that at each point in time $time\_to\_finish(p) = \sharp items(p)/upload(p)$.

At each iteration the algorithm selects the peers with highest $time\_to\_finish$ to send data in the current communication round, and instructs them how much data to send. $\sharp send(p)$ denotes the number of items that a peer p is instructed to send. It is initialized to be zero and increases, if the peer is selected. Peers with maximal

	SendData(input: P, p_0; output: U)
1	$t := 0$; $U := \emptyset$;
2	**for each** $p \in P$
3	$\sharp items(p) := \|items(p)\|$;
4	$time\_to\_finish(p) := \sharp items(p)/upload(p)$;
5	**end for**
6	**while** there exists some peer p with $\sharp items(p) > 0$
7	$t := t + 1$;
8	$\sharp send(p) := 0$ for every $p \in P$;
9	$free := download(p_0)$;
10	**while** $free > 0$
11	choose a peer p, among those with $\sharp send(p) < upload(p)$,
12	where $time\_to\_finish(p)$ is maximal.
13	$\sharp send(p) := \sharp send(p) + 1$;
14	$\sharp items(p) := \sharp items(p) - 1$;
15	$time\_to\_finish(p) := \sharp items(p)/upload(p)$;
16	$\sharp free := free - 1$;
17	**end while**
18	**for each** $p \in P$, with $\sharp send(p) > 0$, add to U instructions
19	to send, at time t, $\sharp send(p)$ new items from p to p_0;
20	**end while**
21	return U;

Figure 2: The `SendData` Algorithm

$time\_to\_finish$ are selected as long as the number of items allocated to them does not exceed their upload capacity (lines 11,12). The variable $free$ records at each point how much available bandwidth the target peer still has for download. At the beginning of each iteration $free = download(p_0)$ (line 9). It decreases as more bandwidth is allocated for the sending peers (line 16), till reaching zero, when the full bandwidth for the given transmission round is used. When this happens, the allocation for this communication round ends (line 10), and a new one for the next round starts. We can prove (by induction on the number of rounds) the following.

THEOREM 3.5. *Given a set P of peers holding pairwise disjoint sets of data items, and a target peer p_0, the direct union plan generated by* `SendData` *is an optimal union plan.*

To continue with Example 3.4 above, plan U_2 is the optimal plan derived by `SendData` for the peers.

Optimized Bandwidth Allocation. The algorithm above allocates to the peers one item at a time (see lines 13,14). When the number of items held by the peers is large, this becomes prohibitory inefficient. To overcome this we use an optimized variant of the algorithm, that allocate bandwidth in bigger chunks, based on the following two key observation.

- Consider the relative order of the $time\_to\_finish$ of the peers, at the beginning and the end of a round. The first observation is that when, in a sequence of rounds, this relative order stays the same, the bandwidth allocation to the peers in these (consecutive) rounds does not change. So rather than distributing one item at a time, we can assign, at once, this round allocation for the maximal number of rounds (time interval) where the $time\_to\_finish$ order among peers stays unchanged.
- A second observation is that when all the peers that are currently allocated bandwidth have the same (maximal) $time\_to\_finish$, they equally share the target's bandwidth, till the $time\_to\_finish$ of one of them reaches zero or drops down to be equal or lower than that of some peer outside this group. Here again, we can allocate, at once, the bandwidth to the

peers in this group (for the maximal possible time interval where their $time\_to\_finish$ does not drop in the above way), splitting it equally among them. (If the target's bandwidth is not divisible by the number of peers is this set, the reminder is allocated to the different peers in alternation).

When assigning bandwidth chunks in this manner, the algorithm produces a *concise, compact description of the union plan*, and its worst case time complexity drops down to polynomial in the number of *peers*. This is because now the number of bandwidth allocation steps depends only on number of distinct *initial* $time\_to\_finish$ values of the peers, and these are bounded by the number of peers. (To see this note that each allocation step now makes the current maximal $time\_to\_finish$ drop one "step" down to the next lower $time\_to\_finish$ value. The number of such steps is as the number of distinct initial $time\_to\_finish$ values).

Going back to Example 3.4, it is easy to see that plan U_2 contains essentially three bandwidth allocation chunks. The first lasts 50 time units, allocating p_1 and p_2 two thirds and one third of the target's bandwidth, resp. The second lasts 20 time units where p_1 and p_2 share equally the bandwidth. Finally, the third lasts 70 time units, with all the three peers sharing the bandwidth equally.

Observe that the optimized algorithm not only has a lower time complexity but also allows to provide the peers with a very compact specification for their optimal operation plan: the peers only need to be told the start/end time of each chunk, and in what rate they should send data in this time interval.

4. COMPACT INFORMATION GATHERING

Let us now examine what information is used by the two-steps algorithm presented in the previous section and how such information can be gathered.

To simplify assume the existence of some *coordinating peer* (this could be the target peer or any other peer in the system) which is responsible for computing the optimal plan and then instructing the peers what data they should send and when. To enable this, three questions need to be addressed: (1) What kind of information the coordinating peer needs in order to compute the plan? (2) What kind of knowledge the peers need in order to execute the plan? and (3) Can we send all of this required data without too much communication cost? We answer these three questions below, in subsections 4.1, 4.2 and 4.3, respectively.

4.1 Deriving the Plan

To derive the plan, the coordinating peer needs to know the upload and download rates of all peers and the size of each equivalence class. The upload and download rates of the peers are easy to obtain by asking the peers. Computing the size of the equivalence classes is more tricky. For each subset of peers $\hat{P} \subseteq P$, we need to know how many elements reside on all the peers in $\hat{P}$ but on no other peers. Or, in other words, what is the size of $\bigcap_{p\in\hat{P}} items(p) - \bigcup_{p\in P-\hat{P}} items(p)$. There are several techniques proposed in the literature for estimating this size [12, 7]. One simple such method is based on a sampling technique called *bottom-k sketches*[12]. For a set s of items, a bottom-k sketch is a small sample of k elements from s. The sample is drawn in a particular way which guarantees, among others, that for any group of item sets $s_1 \dots s_j$, the value of $v = \frac{|s_1\cap\ldots\cap s_j|}{|s_1\cup\ldots\cup s_j|}$ (and resp. of $v_i = \frac{|s_i|}{|s_1\cup\ldots\cup s_j|}$, $i = 1 \dots j$) can be estimated, with high accuracy, using an analogous computation on the samples $k_1 \dots k_j$ of the sets, i.e. by $v' = \frac{|k_1\cap\ldots\cap k_j|}{|k_1\cup\ldots\cup k_j|}$ (and resp. $v'_i = \frac{|k_i|}{|k_1\cup\ldots\cup k_j|}$). The accuracy of the computation can be adjusted by tuning the size k of the sample. (For details see [12]).

To conclude, note that for all sets, $|s_1 \cap \ldots \cap s_j| = |s_i|\frac{v}{v_i}$, and thus can be estimated by $|s_i|\frac{v'}{v'_i}$. Considering again the equivalence classes whose size we want to compute, observe that for each equivalent class, its size, i.e. $\bigcap_{p\in\hat{P}} items(p) - \bigcup_{p\in P-\hat{P}} items(p)$ can be computed by the following inclusion-exclusion formula [10]:

$$
\begin{array}{lll}
|\bigcap_{p\in\hat{P}}| & - & \Sigma_{p\in P-\hat{P}} \mid items(p) \mid \\
& + & \Sigma_{p,p'\in P-\hat{P}} \mid items(p)\cap items(p') \mid \\
& - & \Sigma_{p,p',p"\in P-\hat{P}} \mid items(p)\cap items(p')\cap items(p") \mid \\
& + & \cdots \\
& \cdots & \\
& -(+) & \mid \bigcap_{p\in P-\hat{P}} items(p) \mid
\end{array}
$$

We have seen above that the value of each of the summands in this inclusion-exclusion formula can be estimated using the bottom-k sketches of the item sets and the size of each set. Consequently, all that the coordinating peer needs to have in order to estimate the the equivalence classes size is (1) bottom-k sketches from all peers and (2) the sizes of their item sets.

However, if we try to use the equation shown above, as is, we will encounter two problems. First, the number of summands in the formula is *exponential* in the number of peers, and so is the number of the equivalence classes whose size needs to computed. This may be fine if we have only a few peers to handle, but when dealing with tens or hundreds of peers this may become too heavy computation-wise. Also, in such a case the large number of summands in the formula, and the large number of bottom-k sketches that are intersected in their evaluation, will entail a large accumulated error. This hurdle is inherent, and seems to persist in whichever communication-efficient way we choose to estimate the sizes of the equivalence classes. To alleviate this problem we present below, in Subsection 4.3, an additional heuristic – we show there how to use iterative clustering of the peers such that we are only required to apply the algorithm `AssignData` for a small number of peers at a time. In such case, bottom-k sketches can be used and can be communicated efficiently. See more details in Section 4.3.

4.2 Executing the Plan

To execute the plan, each peer should know (1) which are the elements it is responsible for, and (2) when, and in what rate, to send them. We have seen at the end of the previous section that (2) can be efficiently communicated to each peer. We thus only need to consider issue (1).

Recall that the `AssignData` algorithm splits every equivalence class among the peers, with each peer being assigned a range of items for which it is made responsible. (Recall that we assumed, w.l.o.g, that some order relation on the class items exists, and each peer was assigned by the algorithm a range of items, based on this order). Naturally, this range can be easily communicated to the peer. But to identify the specific items in the range, the peer should figure out which of its items belong to the equivalence class.

One simple way to do this, without actually sending all the data items (or their ids) around, is to use *Bloom filters* [5]. A Bloom filter is a compact data structure used to support set membership queries. The space efficiency is achieved at the cost of a small probability of error. A Bloom filter is a bit array of some size n, where all bits are initially set to zero. It uses l independent hash functions $h_1, h_2..h_l$ with output in the range of 1 to n. When an item is inserted into the Bloom filter, the l bits corresponding to

the result of the l hash functions are set to 1. To test if an item belongs to the set, the l hash functions are evaluated for it. If the corresponding bits in the array are all set to 1 we reply positively. Note that Bloom filters allow for false-positive answers but not for false-negatives. The size of the Bloom filter and the number of hash functions can be tuned to achieve compactness with minimal error probability. Some refined Bloom filter variants that further reduce the probability for error also exist. (For a survey see [6])

Given the Bloom filters of all peers, a peer p can determine, for each data item $d \in items(p)$, which are the other peers $p' \in P$ s.t. $d \in items(p')$, with small error probability. It is important to note, however, that false-positives here may lead a peer p to wrongly conclude that one of its items resides, and is sent by, some other peer, and not send the item even though it should have. Consequently there is some (very small) probability that some items are omitted from the union result. But as mentioned earlier, in many real life scenarios, some small imprecision in the union result is tolerable if allowing for faster processing. There is still, however, some additional difficulty here. For each peer to have the Bloom filters of all the other $|P|-1$ peers, $|P|(|P|-1)$ Bloom filters need to be sent around. As in Section 4.1, if we have hundreds of peers this may entail a significant communication overhead. We explain next how this is avoided.

4.3 The c-Cluster Algorithm

The discussion above pointed out two difficulties. One is the potentially high number of equivalence classes that need to be considered for generating the union plan, and the large inclusion-exclusion formulas required for computing their size. The second is the number of Bloom filters that need to be exchanged for executing the plan. The `c-Cluster` algorithm, described below, alleviate these two problems (at the expense of somewhat compromising the optimality of the plan). The algorithm applies `AssignData` only to small subsets of the peers at a time, and iterates this until most of the data redundancies are eliminated. Then it applies `SendData` to derive the union plan. The algorithm is depicted in Figure 3. We next describe it in detail.

The `c-Cluster` algorithm divides the peers into smaller groups, called clusters, each consisting of c peers, for some $c > 1$ (line 2). If c does not divide $|P|$, the last cluster is the remainder. It then runs `AssignData` individually for each cluster (line 3). This eliminates redundant data items (within each cluster) and assigns to the cluster peers pairwise disjoint item sets (line 4). Note however that peers in different clusters may still hold redundant items. To further reduce this redundancy, the process is reiterated - new clusters are formed and redundant items within them are eliminated. At each iteration, the clusters are chosen (using the peers' bottom-k samples) so that the elements redundancy among the cluster peers is maximal (to maximize the number of redundant elements eliminated in the iteration). [1] The process stops when the overall number of redundant items is considered negligible (line 1). At this point `SendData` is used to determine the best way for the peers to send their assigned items (line 6).

Note that since each cluster contains not more than c peers, the number of equivalence classes that need to be considered by `AssignData` in each cluster (as well as the length of their size formulas) is bounded by 2^c. So the overall number of equivalence classes considered at each iteration is bounded by $2^{c\lceil \frac{|P|}{c} \rceil}$. Also note that to identify the members of each equivalence class, a peer only needs to obtain the Bloom filter of the cluster members (i.e. c - 1 filters).

[1]We employ a standard greedy clustering algorithm [32] that, in a bottom up manner, joins pairs of peers (then cluster), that are estimated to have largest intersections.

	c-Cluster(input: P, p_0, $c > 1$, r; output: U)
1	**while** the number of redundant items in P is above the threshold r
2	divide P into pairwise disjoint clusters (subsets of peers) of size c
3	call `AssignData` for each cluster;
4	**for each** $p \in P$, remove from $items(p)$ all the elements that where not assigned to p by the `AssignData`;
5	**end while**
6	call `SendData` to obtain a union plan U for the peers;
7	return U;

Figure 3: The `c-Cluster` Algorithm

So the overall number of Bloom filters transmissions in each iteration is $|P|(c-1)$. Finally observe that the number of elements held by the peers is reduced at each iteration. Thus smaller Bloom filters can be used to represent their data (without losing accuracy). Consequently, the overall size of the $|P|(c-1)$ Bloom filters exchanged at each iteration shrinks as the algorithm advances.

We present in the following section experimental results that demonstrate that the algorithm scales well to handle hundreds of peers - our stated target scale. To be applied to larger scales, one may want to use a hierarchy of clusters rather than a flat structure. This is a challenging future research.

Observe that a small cluster size c entails low computation and communication cost at each iteration, but may require more iterations to sufficiently reduce the overall number of redundant items. A larger c, makes each iteration more expensive, but decreases the number of iterations. To get some intuition on how an optimal c may be chosen for a given distribution of items to peers, let us look at a simple example.

EXAMPLE 4.1. *Consider a simple scenario where all peers hold the same set of data items and have the same bandwidth. Assuming a "perfect" clustering, at each iteration each cluster identifies c copies of each item and eliminates them. So the redundant $|P|-1$ copies of all items are eliminated after approximately $\log_c |P|$ iterations, and the algorithm can stop. If we assume that the replicas are eliminated from the peers in an "even" manner, the number of items that each peer holds is also reduced, at each iteration, by a factor of c, and so is the size of the Bloom filter that represents them. To summarize, if $bloom(|items(P)|)$ denotes the size of the Bloom filter required to record the initial $|items(P)|$ peer items, then the overall communication cost for the Bloom filters exchanged by the algorithm is given by the following formula.(To simplify, assume that $|P|$ is a power of c.)*

$$|P|(c-1)\Sigma_{i=0\ldots \log_c |P|-1} \frac{bloom(|items(P)|)}{c^i}$$

$$= |P|(c-1)bloom(|items(P)|)\Sigma_{i=0\ldots \log_c |P|-1} \frac{1}{c^i}$$

$$= |P|(c-1)bloom(|items(P)|)\frac{|P|-1}{|P|}\frac{c}{c-1}$$

whose value is minimized when $c = 2$. Our experiments, reported in the following section, indicate that such small cluster sizes are optimal also for other common item distributions.

5. EXPERIMENTAL EVALUATION

To evaluate our algorithms we implemented them and conducted an experimental comparative study of their performance. Since we could not find any previous algorithms aiming at efficient multi-

set union, we compared the performance of our algorithm to (1) the classical sets union where peers send their full data sets to the target (with the peers sending data at each round being chosen in a round-robin fashion), and (2) an adaptation of the pair-wise set reconciliation algorithm [23] where we performed multi-set union recursively by first unioning pairs, then unioning the results, again in pairs, etc. This use of pair-wise sets union turned to be worse than the classical union, even when the pairs where chosen manually in an optimal manner so as to eliminate redundancy as early as possible in the union, since many data items ended up transmitted many times (first in the initial pairwise union, then again in the pairwise union of the results, etc.). We thus compare below our results only to the classical union.

Experimental environment. All the algorithms presented in the previous sections were implemented and ran on a simulator, written in C++, which can be instantiated with different configurations as explained below. We compared unions performed (1) in a classical manner, (2) using our optimal union plans, and (3) using the c-Cluster algorithm of the previous section. Each of the experiments reported below was run 20 times and the graphs show the average of the results. At each case we also discuss the observed deviation from the average. To measure performance we measured the computation time it took to derive the optimal plan and the time (number of bandwidth rounds) it then took for the target to obtain the data. For the c-Cluster algorithm we also added the number of bandwidth rounds it took the coordinating peer to compute the plan (including obtaining the necessary information from the peers, and disseminating the plan to the peers once computed). Since in all the experiments the computation time for deriving the plan took less than 0.5% of the overall time, we ignore it below and focus on data transmission time. To view the performance gain of our algorithms, relative to classical union, we computed the ratio between the number of bandwidth rounds taken by our algorithms and that of the classical union. We term this number the *performance ratio* (PR) of the experiment. A PR below 1 means that our algorithm performed better than the classical union, and the lower is the PR the better is the performance improvement.

We ran two series of experiments to validate our approach. First, we used synthetic data to vary the main parameters that may affect the performance of our algorithms. Second, we used real data, in the context of a distributed full-text index of Wikipedia documents.

5.1 Experiments on Synthetic Data

The parameters we varied in the experiments are the following. The first two parameters are relevant to all algorithms.

Item sets. We ranged the number of sets in the union from 2 to 1000. In each case the sets are assumed to reside on distinct peers with the union request being issued by another target peer. We used three million items and considered uniform as well as Zipfian-like distributions to determine the percentage of items that each peer holds, as well as the items themselves. The k-samples used for the sets consist of 1024 items each. Finally, another parameter that we varied in the experiment is the size of the data items. Small items (e.g. 256 bits) were used to model situations where only item ids are being sent; larger items were used to model scenarios where the items themselves are sent.

Bandwidth. We considered various download and upload rates. The results were fairly consistent, thus we show here only a representative sample whose characteristics are similar to that of an ADSL-based environment [1], having upload rate of 75KBps and a download rate of 750KBps.

The following three parameters are relevant only for the c-Cluster algorithm.

Clusters size. Recall from that Section 4.3 the size of c affects the amount of computation and communication needed at each iteration, as well as the number of iterations required to sufficiently reduce the overall number of redundant items. In our search for an optimal cluster size, we experimented with varying values of c. In accordance with the empirical analysis demonstrated in Section 4.3, the best performance was observed for $c = 2$, and this is what is used in the experiments reported below.

Replication-level Threshold. The c-Cluster algorithm works in an iterative manner. In each iteration, clusters are formed, and redundant items are eliminated *within each cluster*. The process stops when the *overall* number of redundant items is considered negligible, i.e. is below a given threshold. A low threshold implies less replicated items, hence less data to be sent, and consequently a more efficient union. But low threshold also means more iterations, hence more use of Bloom filters to identify common items, and consequently higher potential for errors. To understand the tradeoff between performance and accuracy, and choose an adequate threshold, we ran `c-Cluster` with a varying number of clustering iterations, and analyzed the results.

To see an example, consider the following experiment, where 25 sets are being unioned, with the sets sizes and item distribution to the sets drawn in a uniform manner. (The size of the Bloom filters being used will be discussed below). The performance gain, for varying number of iterations is depicted in Figure 5. Recall that the performance ratio (PR) is the ratio between the performance of the c-Cluster algorithm and that of the classical union. Lower values imply better performance improvement. It is evident that the PR decreases as the number of iterations increases, i.e. the performance gain grows. The improvement speed is drastic in the first bunch of iterations and then becomes marginal. This is because, after a while, most of the redundant items are eliminated. Further iterations bring only marginal benefit.

The corresponding error rate, for the varying number of iterations, is depicted in Figure 4. As expected, we can see that the error increases when the number of iterations increases. The error, and its growth, are marginal for the first bunch of iterations, and then increase.

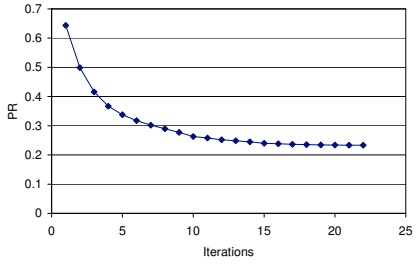

Figure 4: PR after each iteration

We see that in this experiment stopping after approximately 10 iterations is optimal for achieving both good performance and marginal error. Running a series of such experiments, and analyzing these "optimal" stopping points, we observed they have rather strong characteristics: In cases where the original average number of item replicas is not too high (up to 5) the optimal stopping point

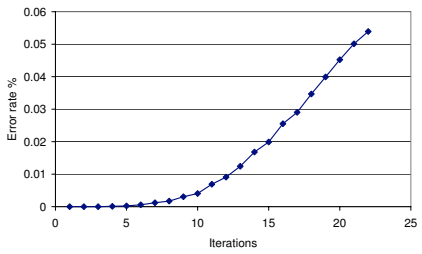

Figure 5: Error rate after each iteration

bloom filter	error rate %	PR
4	2.639	0.15
8	0.145	0.20
16	0.003	0.28

Figure 6: Bloom filters of varying sizes

was close to the ideal state of "one copy per item", with about 1.2 average number of replicas. For cases with higher number of original item copies, getting to such low number of replica requires too many iterations, (hence introduces too much error). The experiments showed that it suffices that the ratio between the original number of replicas and their number at the stopping point is high (about 5 to 1). Our threshold for the remaining experiments was thus chosen following this policy.

Bloom Filters. The c-Cluster algorithm uses Bloom filters to determine, at each peer, which elements in $items(p)$ belong to sets residing on other peers. Large Bloom filters have lower error probability. However, large filters also entail higher communication overhead, hence harm the performance. To better understand the tradeoff, we experimented with Bloom filters of various sizes. A sample such experiment is depicted in Figure 6. We see here the performance ratio (PR) and error rate. In this experiment 25 sets are being unioned, with uniform distribution for sets sizes and assignment of items to sets. (Similar results were observed when varying the number of sets and distributions, hence we omit them here.) In the first (resp. second, third) row, each peer p uses Bloom filter of size $4 * |items(p)|$ (resp. $8 * |items(p)|$, $16 * |items(p)|$). We can see that the smaller the bloom filters is, the greater is the performance gain (i.e. the PR value is lower), but the error is higher. Shooting for an average error rate lower than 0.01% we chose to use here the 16 bits per item.

Now that we explained how the c-Cluster parameters were tuned, let us examine the resulting performance gains and error rates. In all the experiments below, unless stated otherwise, we ranged the number of peers (sets in the union) from 2 to 1000. In all cases, from 50 peers and up the results basically did not change and the curves remained horizontal. We thus show in the figures the results up to 65 peers only. We first present the results for uniform distribution then discuss the Zipfian one.

Figure 8 shows the performance gain for the union of a growing number of sets. We see here the performance for small items of 256 bits (essentially the size of a url) and for larger items of size 512 and 1024 bits each. As expected, the larger the items are the higher is the performance gain (i.e. the lower is the PR). This is because more bandwidth is saved when avoiding the transmission of a big item. But even for small items the saving is significant. While the improvement, relative to the classical union, is moderate when just two sets are unioned (20%), we see a significant gain (over 50%) already at five sets, and over 70% improvement when

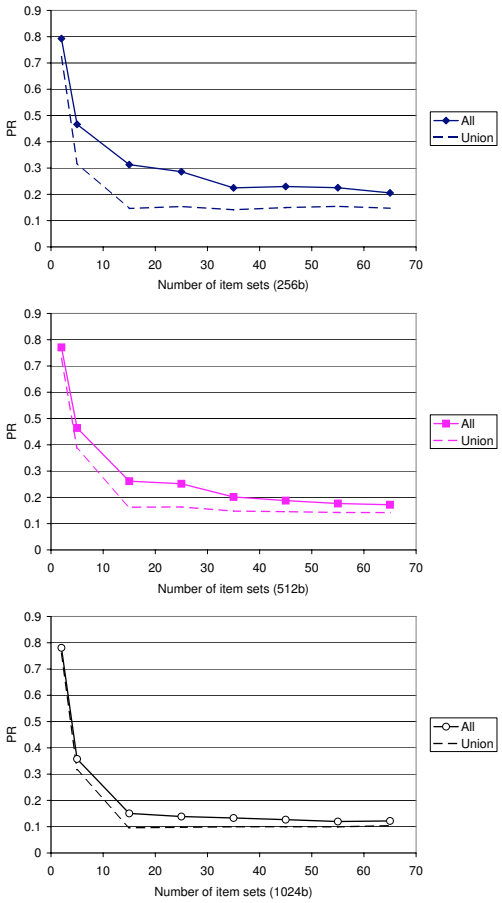

Figure 7: PR with(out) transfer of auxiliary data

10 or more sets are unioned. This growing improvement is due to the fact that when a larger number of sets is considered, items may have more replicas, whose elimination speeds up the union. It should be stressed that the variance in the experiments results was small, and in all cases below 10%.

Figure 7 demonstrates which part of the effort is spent on computing the union plan (namely obtaining the bottom-k samples and Bloom filters from the peers, and disseminating the plan to the peers once computed), and which part is spent on the actual union execution. The items here are of size 256, 512, and 1024 bits, resp. For small items the plan computation takes a relatively large portion of the overall execution time. (But still, as we saw in Figure 8 above, even with this overhead the overall execution time is still significantly lower than that of a classical union.) For bigger items, the plan computations time becomes relatively marginal. This is because the transmission time of larger items is higher, while the union plan computation time stays about the same.

The average error, for items of size 256 bits, is depicted in Figure 9. (The error for other item sizes is approximately the same, as it is not affected by the size of the items). In all cases the average error is below 0.008%. It is close to 0.008% when a small number of sets is being unioned (with the exception of the case of two sets, explained below) and decreases significantly as the number of sets grows. This decrease in error is because when a larger number of sets is being unioned, items can have more replicas. Consequently, even if one peer wrongly omits an item, there is a good chance that the item will nevertheless be sent to the target by some other peer. For union of two sets, the error is particularly low since the

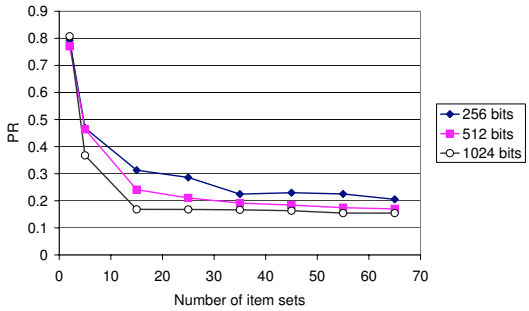

Figure 8: Performance Ratio for growing number of sets

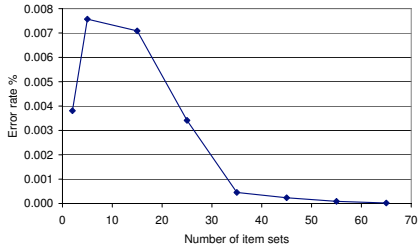

Figure 9: Error rate for growing number of sets

c-Cluster algorithm performs just one iteration (and a low number of iterations reduces the accumulated error). The figure shows the average error, but in all the experiments the maximal observed error was also very low and below 0.01%.

Optimal union plans. It is evident from the experiments above that the c-Cluster algorithm outperforms the classical union. But how far is its performance from the one we could obtain by using an optimal union plan? Figure 10 shows the ratio between the performance of the c-Cluster algorithm and the optimal union, for items of size 256 bits. (The results for other item sizes were similar). The number of unioned sets ranges here from 2 to 13. As expected, for optimal union plans, beyond 13 sets, the number of the equivalence classes became excessively large and computing their size and generating the corresponding plan was too slow. Hence we could not measure the performance. We can see that the c-Cluster algorithm approximates the optimal plan rather well (while being scalable, unlike the optimal union algorithm). The plan computed by the c-cluster algorithm takes at most 1.5 times the optimal one (and this, as we had already seen above, is already significantly more efficient than the classical union).

Our results for Zipfian distributions were consistent with those showed above for uniform distribution. In this set of experiments the peer sizes is determined in zipfian-like distribution, with varying number of small, medium and large peers. Figure 11 shows the performance gain, for the union of a growing number of sets, for three representative experiments. In experiment A the majority of peers are small (each holding 10-20% of the data items). In B the majority of peers are of medium size (20-40% of the data items). In C the majority of peers are large (40-80% of the data items). The items size here is 256b. We can see that the larger the peers the better the improvement (since large peers typically contain common items). But even with a majority of small peers we get 25% improvement already at 5 peers, and it grows significantly with the number of peers. The error rate here, as well as the performance relative to the optimal algorithm, were also consistent with what was observed for uniform distributions and are thus omitted.

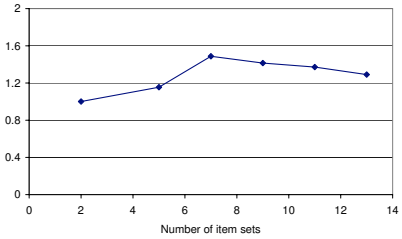

Figure 10: Time / optimal plan time

5.2 Experiments on Real Data

We tested our algorithms in the context of a real life application concerning distributed full text indexing of English Wikipedia documents. The scenario is similar to the one described in the Introduction. The index is based on a on a DHT [16, 28]. For every word w appearing Wikipedia, it contains some peer p_w holding the ids of the Wikipedia documents that include the word. The ids here are of 512 bits. The search is synonym-based. In particular, the user can give a word to the system and ask for the documents containing this word or any of its synonyms. To restrict the search, a subset of the synonyms, that best matches the user's particular interest/context, can be specified.

To test the performance of our algorithms, we emulated the above mentioned environment using our simulator. We run a series of experiments where in each experiment we randomly chose 10-15 queries (words). For each query (word) w_1, we considered its synonyms $w_2, \ldots, w_n$, and computed the union of the sets held by $p_{w_1} \ldots p_{w_n}$. The the size of the sets ranged from 20k-70k items (document ids), with the sum of the sizes of the sets involved in each query ranging from 200k-1000k items.

We show below the results obtained for a representative such experiment. For each query (word) in the experiment, we show how many synonyms it has (and thus how many sets were unioned) and what was the average replication level (i.e. in how many of the unioned sets each document id appears). Note that the replication level here essentially reflects the richness of the language used for writing the documents, i.e. how many synonyms are used, on the average, in each document.

Figure 12 shows the performance gain observed for each query (word), with the queries sorted by the observed replication level. We can see that, overall, the replication level is rather moderate, (implying that the language used in the documents is not very rich, with only few synonyms used in each document). Nevertheless, even with this low replication level we get significant performance improvements of more than 40% on the average. We can see that the PR decreases (i.e. the performance gain grows) with the increase in the replication level. Figures 13 shows the same experiment, only this time the queries are sorted by the number of words'

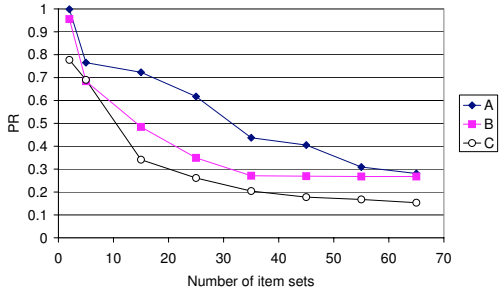

Figure 11: Performance Ratio for Zipfian distributions

synonyms (namely the number of sets being unioned). Here again we can see an increase in the performance gain (lower PR values) with the increase in the number of sets in the union. This is consistent with our results for synthetic data. Finally, we can see that the error rate, shown in Figure 14, is very low here too, again in accordance with the results on the synthetic data.

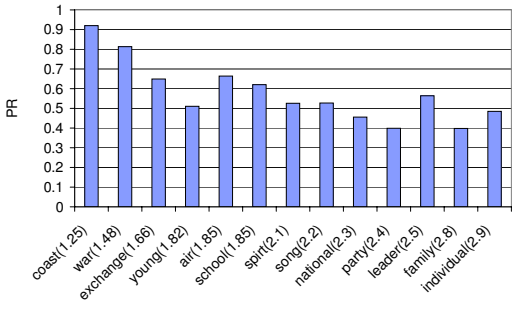

Figure 12: PR for varying replication level

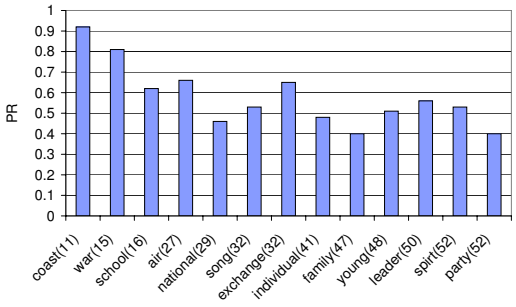

Figure 13: PR for varying number of synonyms (unioned sets)

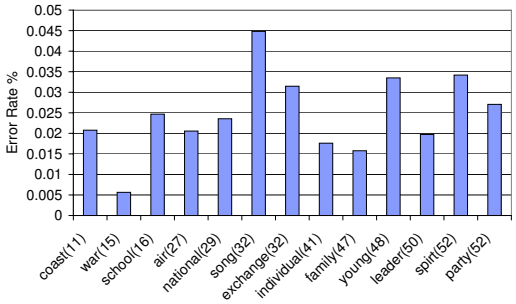

Figure 14: Error rate

6. RELATED WORK AND CONCLUSION

We studied in this paper the union of non-disjoint data sets residing on distinct peers. We defined the notion of optimal union plans and presented efficient algorithms for computing and executing such plans. They use a compact, cheap to communicate, description of the data sources, for efficient union computation with reasonable accuracy. We have implemented our algorithms, and showed experimentally that they are extremely effective.

Scheduling problems in query processing over parallel, distributed and P2P databases typically deal with inter-query, inter-operator, and intra-operator parallelism [17, 15]. Our work fits in the last category, offering an optimized parallel implementation of the union operator. Most works on query optimization in such distributed settings focus on SPJ (select,project,join) queries, employing e.g. semi-join [4] and bloom-join [18] techniques to reduce communication and speed up processing. Efficient aggregate computation has also been studied, e.g. [30]. Our work is complementary to these efforts, allowing for an efficient union/merge of (sub)query results. Previous work performs multi set union by recursively unioning pairs. In contrast we show here that a *direct* optimal union is possible and yields substantial performance gain.

The use of data synopsis, such as bottom-k samples and (variants of) Bloom filters, is common in (distributed) stream processing, e.g. for computing aggregates [19], distributed top-k queries[21], and eliminating redundant items from a single output stream [13, 29]. Developing optimal union plans in a streaming context is a challenging open problem.

As explained in the introduction, the problem of efficient multi set union computation is closely related to that of pairwise sets-reconciliation [24]. For both problems, computing the sets intersection is an important ingredient of the solution. Accurate computations of sets intersection in a distributed setting was shown to be hard [33], requiring exchange of data of size at least linear in the size of the input. To bypass this lower bound, sets-reconciliation algorithms use assumptions on the properties of the sets (e.g. bounds on the size of the intersection) or probabilistic solutions. For instance, Minsky et al. [23] use such assumptions to developed a sets-reconciliation algorithm with communication complexity logarithmic in the size of the symmetric difference between the sets. The algorithm relays on each peer computing a characteristic polynomial of the entire set, based on factorization and interpolation techniques. The use of factorization, interpolating and in general high order polynomials requires expensive computational tasks, which are not reasonable in our setting where peers serve an extensive number of user queries. Byers et all [8] developed an approximated sets-reconciliation algorithm based on a tree of bloom filters. While lighter computation-wise than [23], the resulting overall error rate is too high for our needs here, and we chose to communicate (wrapped and compressed) Bloom filters for lower error.

Parallel data download is used in many content dissemination and streaming systems [31, 9, 11, 22]. Such systems are often based on peers cooperation, where each peer provides (and obtains) to (from) other needing peers parts of the data - typically a large file - which she (they) had already acquired. Sets reconciliation is used by peers in content dissemination systems to determine which missing pieces of data they can acquire from their neighbors. In all the systems we are aware of, a peer that gets data from several neighbors performs pairwise sets reconciliation with each neighbor separately. This has been shown to be non efficient when neighbors contain overlapping information [9]. The union algorithms presented here can help to improve performance in such situations. It should also be noted that typical content dissemination platforms are designed to download large files and do not perform well when used to downloaded a large number of small items[27]. In contrast, we saw that our union algorithms perform well also for such data.

In information mediation systems data is often gathered from several independent data sources. Removing data redundancies and identifying and reconciling inconsistencies are essential for successful data integration [14]. The redundancies that our algorithms eliminate consist of multiple occurrences of *the same* data items. Complementary object fusion techniques [25, 14] may be used to fuse together distinct items that represent the same real life object.

Our work targets on-line query processing where the computation time is expected to be short and consequently assumes the network constraints to be fairly stable throughout the computation. Extending our algorithms to a dynamic setting is a challenging fu-

ture research. Another interesting problem left for future research is deriving a strong theoretical bound on the performance of our approximated union plan, relative to the optimal one.

7. REFERENCES

[1] ADSL - Asymmetric Digital Subscriber Line. http://en.wikipedia.org/wiki/Adsl.

[2] S. Abiteboul, I. Manolescu, N. Polyzotis, N. Preda, and C. Sun. Xml processing in dht networks. In *ICDE '08*, 2008.

[3] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network flows: theory, algorithms, and applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[4] P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve, and J. B. Rothnie. Query processing in a system for distributed databases (sdd-1). *ACM Trans. Database Systems*, 6(4), 1981.

[5] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Comm. of the ACM*, 13(7):422–426, 1970.

[6] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2004.

[7] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations (extended abstract). In *Proc. STOC'98*, pages 327–336, 1998.

[8] J. Byers, J. Considine, and M. Mitzenmacher. Fast approximate reconciliation of set differences. Technical report, 2002. Boston University Computer Science Tech. Rep. 2002-019.

[9] J. Byers, J. Considine, M. Mitzenmacher, and S. Rost. Informed content delivery across adaptive overlay networks. In *Proc. SIGCOMM'02*, pages 47–60, 2002.

[10] Zhiyuan Chen, Nick Koudas, Flip Korn, and S. Muthukrishnan. Selectively estimation for boolean queries. In *Proc. PODS'00*, pages 216–225, 2000.

[11] B. Cohen. Incentives build robustness in bittorrent. In *Proc. Workshop on the Economics of Peer-to-Peer Systems*, 2003.

[12] E. Cohen and H. Kaplan. Summarizing data using bottom-k sketches. In *Proc. PODC'07*, pages 225–234, 2007.

[13] F. Deng and D. Rafiei. Approximately detecting duplicates for streaming data using stable bloom filters. In *SIGMOD '06*, pages 25–36, 2006.

[14] A. Halevy, Z. Ives, J. Madhavan, P. Mork, D. Suciu, and I. Tatarinov. The piazza peer data management system. *IEEE Transactions on Knowledge and Data Engineering*, 16(7):787–798, 2004.

[15] R. Huebsch, M. Garofalakis, J. M. Hellerstein, and I. Stoica. Sharing aggregate computation for distributed queries. In *Proc. SIGMOD'07*, pages 485–496, 2007.

[16] Y. Joung, C. Fang, and L. Yang. Keyword search in dht-based peer-to-peer networks. In *Proc. ICDCS'05*, pages 339–348, 2005.

[17] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, 2000.

[18] L. F. Mackert and G. M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *Proc. VLDB'86*, pages 149–159, 1986.

[19] A. Manjhi, V. Shkapenyuk, K. Dhamdhere, and C. Olston. Finding (recently) frequent items in distributed data streams. In *ICDE '05*, pages 767–778, 2005.

[20] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Revised Papers from the First Int. Workshop on P2P Systems*, 2002.

[21] S. Michel, P. Triantafillou, and G. Weikum. Klee: A framework for distributed top-k query algorithms. In *Proc. VLDB*, 2005.

[22] T. Milo, T. Zur, and E. Verbin. Boosting topic-based publish-subscribe systems with dynamic clustering. In *Proc. SIGMOD*, pages 749–760, 2007.

[23] Y. Minsky and A. Trachtenberg. Practical set reconciliation. Technical report, 2002. Dept. Elec. Comput. Eng., Boston Univ., Boston, MA, Tech. Rep. BU-ECE-2002-01.

[24] Y. Minsky, A. Trachtenberg, and R. Zippel. Set reconciliation with nearly optimal communication complexity. In *Proc. International Symposium on Information Theory*, 2001.

[25] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object fusion in mediator systems. In *Proc. VLDB'96*, pages 413–424, 1996.

[26] A. Sedeno-Noda, D. Alcaide, and C. Gonzalez-Martin. Network flow approaches to pre-emptive open-shop scheduling problems with time-windows. *European Journal of Operational Research*, 174(3):1501–1518, 2006.

[27] G. Sivek, S. Sivek, J. Wolfe, and M. Zhivich. Webtorrent: a bittorrent extension for high availability servers. http://pdos.csail.mit.edu/6.824-2004/reports/jwolfe.pdf.

[28] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proc. SIGCOMM'01*, pages 149–160, 2001.

[29] N. Tatbul, U. Çetintemel, and S. B. Zdonik. Staying fit: Efficient load shedding techniques for distributed stream processing. In *VLDB'07*, pages 159–170, 2007.

[30] A. Tsois and T. K. Sellis. The generalized pre-grouping transformation: Aggregate-query optimization in the presence of dependencies. In *Proc. VLDB'03*, pages 644–655, 2003.

[31] C. Wu and B. Li. Optimal peer selection for minimum-delay peer-to-peer streaming with rateless codes. In *Proc. ACM workshop on Advances in P2P multimedia streaming*, pages 69–78, 2005.

[32] R. Xu and I. Wunsch. Survey of clustering algorithms. *IEEE Transactions on Neural Networks*, 16(3):645–678, 2005.

[33] A. C. Yao. Some complexity questions related to distributive computing. In *Proc. of STOC'79*, pages 209–213, 1979.