D-Swoosh: A Family of Algorithms for Generic, Distributed Entity Resolution

Omar Benjelloun Hector Garcia-Molina Heng Gong Hideki Kawai
Tait E. Larson David Menestrina Sutthipong Thavisomboon

Stanford University
{benjello,hector,henggong,hkawai,telarson,dmenest,sthavis}@cs.stanford.edu

Abstract

Entity Resolution (ER) matches and merges records that refer to the same real-world entities, and is typically a compute-intensive process due to complex matching functions and high data volumes. We present a family of algorithms, D-Swoosh, for distributing the ER workload across multiple processors. The algorithms use generic match and merge functions, and ensure that new merged records are distributed to processors that may have matching records. We perform a detailed performance evaluation on a testbed of 15 processors, for cases where application knowledge can eliminate some comparisons and where all records must be matched. Our experiments use actual comparison shopping data provided by Yahoo!.
(**keywords:** Entity Resolution, Information Integration, Data Cleaning)

1 Introduction

The process of *Entity Resolution (ER)* identifies records that refer to the same real-world entity and merges them. For example, in a comparison shopping application, records arrive from different stores. Two records r_1 and r_2 may refer to exactly the same product, e.g., an iPod, but may not have a unique identifier that links them. (For instance, each store may use a different identification scheme, and in addition, product names, descriptions and other fields may contain typos or may be missing.) If r_1 and r_2 *match*, i.e., are deemed to be "similar enough" that they represent the same product, then it may be useful to *merge* them into, say, r_{12}, a "composite" of the original records. Because r_{12} contains more information that r_1 or r_2 alone, it may now match other records that neither r_1 nor r_2 match.

ER is a critical component of any information integration task, whether we are looking for terrorist activities, combining customer records after two companies merge, or consolidating student records for students that change schools. Unfortunately, ER is usually a computationally expensive process. First, there are often many records. In a comparison shopping application, for instance, tens of millions of records can be received daily from different merchants. Second, the record comparisons that are performed are typically expensive, as compared to, say, checking an equality predicate in a database join. For example, to compare records, we may perform maximal common sub-sequence computations on text fields. We may also look up values in large dictionaries of canonical terms (e.g., to map "Bobby" to "Robert"). We may also need to do language or character set translations on some fields.

There are generally two ways to deal with the extremely high computational load of ER: One is to "partition" the problem using semantic knowledge and the other is to exploit multiple processors. As an example of partitioning, we can divide our product records (in the comparison shopping application) by category (CDs, MP3 players, cameras, etc.), and then only try to match records in the same category (possibly with some overlap between partitions, e.g., if products have multiple categories). Here we are using the knowledge that records in different categories will never match. Of course, in some applications we may not have such knowledge, or we may not be willing to assume that the categorization is perfect. But even with semantic knowledge, there may still be many comparisons to perform because the resulting categories or buckets are still relatively large. Thus, we believe that the second option, parallelism, will be essential in many applications. In this paper we focus on techniques for distributing the ER process over multiple processors, both for the case where we have no semantic knowledge, and the case where we have ways of partitioning the problem.

A majority of the work on ER to date has been done in the context of a specific application, e.g., resolving author or customer records. Thus, it is often hard to generalize the results, i.e., it is difficult to separate the application-specific techniques from the ideas that could be used in other contexts. To clearly separate the application details from the generic ER processing, in this paper we encapsulate the record matching and merging into two black-box functions. The *match* function takes as input two records, and decides if they represent the same real-world entity. The match function may compute similarities between the two records (and perhaps other records), but eventually it must decide if the two records should be merged. If records are to be combined, the *merge* function is called to generate a new composite record.

Note that by encapsulating match and merge, we are separating accuracy issues from performance issues. From our perspective, accuracy (i.e., if the composite records are "correct") is a function of the match and merge functions, which we are given. Our job is only to invoke these functions as few times as necessary, and to distribute the work of evaluating the functions across processors.

1.1 Overview of Our Approach

To motivate our approach, and to illustrate some of the challenges faced, consider the following simple example. Say we have 6 input records, r_0 through r_5. Figure 1 shows one possible ER outcome for this example, assuming we run on a single processor. After comparing the input records, we discover that r_0 and r_3 match, so they are merged into r_6. The new record then matches with r_4; the resulting merged record is r_8. In this example, r_2 and r_5 also merge, into record r_7. The final answer is $\{r_1, r_7, r_8\}$, where none of these records match any further.

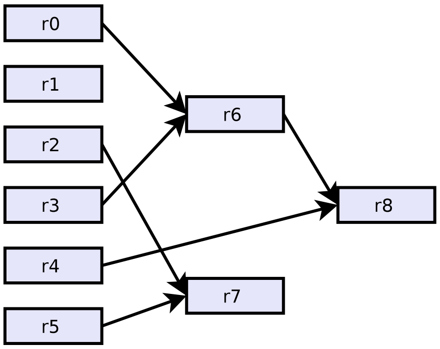

Figure 1: A sample run of Entity Resolution

The matches in our example could have been found in a different order, e.g., perhaps r_0 and r_4 are merged first, and then with r_3. In this paper we will assume that the order of merging does not impact the final outcome, and that the records that went into a composite record can be deleted from the final answer. (The precise properties are given in Section 2.) However, it is important to note that if these properties do not hold, our distributed algorithms can be extended in a straightforward way (for example, we would not delete records after they merge).

Say we want to distribute this ER processing over three processors, P_0, P_1 and P_2. The problem is analogous to performing a distributed self-join in a database system: we wish to distribute the records to the processors so that the comparisons are done in parallel. However, there are two important differences with joins:

- With a join, the match function is simple and known, so we can exploit this knowledge. For example, if the common join attribute is X, we can distribute records with $X < 10$ to one processor, those with $X \geq 10$ to another, and comparisons will be localized to one processor. With ER, we may not know anything about the match function, so we need to develop mechanisms that work in general.
- With ER, unlike joins, there is a "feedback" loop where merged records need to be re-distributed and compared to other records.

To address theses issues, we use a general distribution function we call *scope*. That is, scope(r_i) gives us the set of processors where r_i will be sent. For example, Figure 2 shows one possible scope function. Note that this scope function can be expressed by scope$(r_i) = \{P_j, P_k\}$ where $j = i \bmod 3$, and $k = (i+1) \bmod 3$.

P_0	P_1	P_2
r_0	r_0	
	r_1	r_1
r_2		r_2
r_3	r_3	
	r_4	r_4
r_5		r_5
r_6	r_6	
	r_7	r_7
r_8		r_8

Figure 2: One possible scope function

Notice that for every pair of records, there is at least one processor that has both records. For example, the pair r_0, r_2 is at P_0, and the pair r_0, r_3 is at P_0 and P_1. Thus, if each processor executes all match comparisons on records in its scope, we will not miss any comparisons. Our algorithm will proceed by distributing the input records to processors using the scope function. Then each processor will apply the match function to the pairs of records it has.

Of course, with this scheme we may perform redundant comparisons, e.g., in our case, the r_0, r_3 comparisons would be performed by both P_0 and P_1. To avoid redundant comparisons we also introduce a *responsible* function (denoted resp): processor P_k will apply the match function to r_i and r_j only if it is in the scope of both records, and resp(P_k, r_i, r_j) is true. Intuitively, the responsible function needs to compute the set of processors that will have both r_i and r_j, and then deterministically choose one of them, without overloading some processors. Since the responsible function typically performs simple arithmetic (or more basic) operations, it is much cheaper to compute than the match function, and hence the saving can be significant. In Sections 4.1 and 4.2, we provide several instances of scope and resp functions.

When a processor finds a match among its records, it computes the merged record and distributes it so that the new record is compared against all existing records. Thus, scope and responsible functions should work with merged records, and guarantee that no matches are missed. However, even with a good responsible function, it will be impossible to avoid all redundant work due to the concurrent execution of related comparisons. To illustrate this unavoidable redundancy, suppose that three records r_i, r_j and r_k all match pairwise. Say that processor P_0 is responsible for the r_i, r_j comparison, processor P_1 for the r_i, r_k pair, and P_2 for the r_j, r_k pair. Each processor in parallel will discover a match and will merge its pair of records. The three resulting records, r_{ij}, r_{ik} and r_{jk} are distributed, and compared again, and each pair of the new records could be the responsibility of a different processor. Each of these processors will then independently compute the same merged record r_{ijk} (recall that merge order is not important). Thus, r_{ijk} is generated three times. Our algorithm will eventually remove the duplicate copies, but we have performed more work than necessary to get r_{ijk}. Depending on the timing of events, some of the redundant work may be avoided, but there is always the potential for redundancy. As a matter of fact, adding processors beyond some limit may be counter productive, as it may lead to more redundant work and more communication overhead. One of our goals here is to empirically study these issues, and to evaluate how much redundant work may be done in practice.

If we have semantic knowledge, we may develop scope and responsible functions that reduce the number of comparisons, analogous to what is done with joins. For instance, if product records can only match if their "price" attribute is numerically close, we can design a scope function that assigns records to processors by price, so a processor only gets records whose price is close. We discuss scope and resp functions that exploit semantic knowledge in Section 4.2.
In summary, the contributions we make are:

- We define the problem of distributed entity resolution, for the case where black-box match and merge functions are used (Section 3).
- We present a generic, distributed algorithm, D-Swoosh, that runs ER on multiple processors using scope and responsible functions, and computes the correct answer (Section 3.2).
- We suggest a variety of scope and responsible functions, some for scenarios with no semantic knowledge, others that exploit knowledge that often arises in ER applications (Section 4).
- We present metrics to evaluate distributed ER, and conduct a detailed evaluation of D-Swoosh and the various scope and responsibility functions on a 15 processor testbed, using comparison shopping data provided by Yahoo! (Section 5. Our results show that the ER process can take advantage of parallelism with great benefit.

Background material is given in Section 2, and related work is presented in Section 7. Section 8 is our conclusion.

2 Preliminaries

We first define our generic setting for ER; additional details can be found in [2]. We are given a set of records $R = \{r_1, \ldots, r_n\}$ to be resolved. The *match function* $M(r_i, r_j)$ evaluates to true if records r_i and r_j are deemed to represent the same real-world entity. As shorthand we write $r_i \approx r_j$. If $r_i \approx r_j$, a *merge function* generates $\langle r_i, r_j \rangle$, the composite of r_i and r_j.

Match and merge functions are usually not arbitrary, and may satisfy the following four simple properties:

- *Commutativity:* $\forall r_1, r_2$, $r_1 \approx r_2$ iff $r_2 \approx r_1$, and if $r_1 \approx r_2$, then $\langle r_1, r_2 \rangle = \langle r_2, r_1 \rangle$.
- *Reflexivity/Idempotence:* $\forall r,\ r \approx r$ and $\langle r, r \rangle = r$.
- *Merge representativity:* If $r_3 = \langle r_1, r_2 \rangle$ then for any r_4 s.t. $r_1 \approx r_4$, we also have $r_3 \approx r_4$.

- *Merge associativity:* $\forall r_1, r_2, r_3$ such that both $\langle r_1, \langle r_2, r_3\rangle\rangle$ and $\langle\langle r_1, r_2\rangle, r_3\rangle$ exist, $\langle r_1, \langle r_2, r_3\rangle\rangle = \langle\langle r_1, r_2\rangle, r_3\rangle$.

As discussed in [2], if these properties do not hold, the entity resolution problem becomes much more expensive. For instance, without associativity we must consider all possible orders in which records may match and merge. In a sense the solution will not be "consistent" because there can be multiple composite records that can be derived from the same matching records. Because of these problems, there is a strong incentive for programmers to design match and merge functions that satisfy the above properties. Thus, in this paper we focus on applications where these properties hold. However, as noted earlier, the algorithms we present can be extended to perform the additional comparisons required if the properties do not hold.

During entity resolution, we may generate merged records that do not carry additional information, relative to other records. For instance, say records r_i, r_j, r_k have merged into record r_{ijk} (via two merges). In this case we no longer need records r_i, r_{ij}, or any other records derived from a subset of $\{r_i, r_j, r_k\}$. We call the unnecessary records *dominated*.

Definition 2.1 *Given two records, r_1 and r_2, we say that* r_1 is dominated by r_2*, denoted $r_1 \leq r_2$ if $\langle r_1, r_2\rangle = r_2$.*

One can easily verify that domination is a partial order on records. We are now ready to define the entity resolution problem:

Definition 2.2 *Given a set of input records R, an* Entity Resolution *of R, denoted $ER(R)$ is a set of records such that:*

- *Any record in $ER(R)$ is derived (through merges) from records in R;*
- *Any record that can be derived from R is either in $ER(R)$, or is dominated by a record in $ER(R)$;*
- *No two records in $ER(R)$ match, and no record in $ER(R)$ is dominated by any other.*

In [2], we show that the entity resolution of R is unique, and provide R-Swoosh, an optimal algorithm for performing entity resolution on a single processor. R-Swoosh incrementally processes the input records, while maintaining a set R' of (so far) non-dominated, non-matching records. R-Swoosh performs a merge as soon as a pair of matching records is found, puts the obtained record back into the input set R, and deletes the pair of matching records immediately.

To illustrate the operation of R-Swoosh, consider a run on the instance of our initial example, supposing the records are processed in the order of their identifiers. We take r_0 and compare it against records in R'. Since R' is initially empty, there are no matches with r_0, so r_0 is moved to R'. Next, r_1 is compared against records in R', i.e., against r_0. In our example (Figure 1), there are no matches, so r_1 is also moved to R'. Record r_2 is moved to R' in a similar fashion. When r_3 is compared against R', we discover that r_0 and r_3 match. Therefore, $r_6 = \langle r_0, r_3\rangle$ is added to R, and r_0 and r_3 are removed from R' and R, respectively. Next, r_4 is processed and added to R', r_5 is processed generating $r_7 = \langle r_2, r_5\rangle$, which is added into R; r_6 is processed generating $r_8 = \langle r_4, r_6\rangle$ into R. The remaining R records are processed and moved to R'. At the end, R' holds $ER(R) = \{r_1, r_7, r_8\}$.

In [2], we also give a variant of the algorithm, F-Swoosh, that efficiently caches the results of value comparisons.

3 Distributed ER

In this section, we extend the R-Swoosh sequential algorithm for generic ER to be distributed, in order to run in parallel on multiple processors. We start by defining the primitives needed to capture distribution, then present the algorithm.

3.1 Distribution Primitives

Our p processors are $\mathcal{P} = \{P_0, \ldots, P_{p-1}\}$. As mentioned in the introduction, we introduce a "scope" function to distribute records across processors and a "responsible" predicate to reduce redundant work, by deciding which processors are responsible for which comparisons. These are defined formally as follows.

Definition 3.1 *A* scope function *is a function from $\mathcal{R}$, the domain of records to $2^{\mathcal{P}}$ that assigns to each record r a subset $scope(r)$ of the processors. A* responsible predicate *is a Boolean predicate over $\mathcal{P} \times \mathcal{R} \times \mathcal{R}$. When $resp(P_i, r, r') =$ `true`, we say that processor P_i is responsible for the pair of records (r, r'). Scope and resp satisfy the following "coverage" property.*

Coverage property: *For any pair of matching records r, r', there exists at least one processor P_i such that $P_i \in scope(r) \cap scope(r')$ and $resp(P_i, r, r') = true$.*

Interestingly, the coverage property is related to the distributed mutual exclusion problem [8]. We can think of record r as a process that "locks" the processors in its scope. Since r' also locks its scope, and since $scope(r) \cap scope(r')$ is not empty, then r and r' are mutually excluded. Thus, requiring scopes to intersect (so every pair of matching records is compared) is equivalent to requiring the locks to conflict at least one processor. Any coterie [11] used for mutual exclusion can hence be used for guaranteeing coverage in our context. However, we only want coteries that distribute the workload evenly, since we are doing expensive record comparisons and not locking.

3.1.1 Distributed Computing Model

We make the standard assumptions about our distributed computing model: First, we assume that no messages exchanged between processors are lost. Second, that messages sent from a processor to another are received in the same order as they are sent, and processed in that same order (this can be achieved easily by numbering the messages). Finally, that processors are able to use some existing distributed termination detection technique to detect that they are all in an idle state (see, e.g., [6]).

3.2 The D-Swoosh Algorithm

We are now ready to give our general algorithm for distributed ER. The algorithm of Figure 1 asynchronously runs a variant of the R-Swoosh algorithm at each processor P_i. Initially, each P_i receives the records in its scope. Each P_i maintains a set R'_i of non-dominated, non-matching records. The processor also keeps a set D_i of the records it knows have been deleted.

When a new (added) record r is received by P_i, it is successively compared to every record r' in R'_i, provided P_i is responsible for that comparison. If a match is found, the matching records are merged immediately, and messages are sent to the relevant processors (identified through the scope function) instructing them to add or delete records. If no match is found, then r is added to R'_i.

Note that if the new merged record $\langle r, r'\rangle$ is identical to either r or r', then not all the messages are sent out. Furthermore, if $\langle r, r'\rangle$ is equal to r, the record we are processing, we continue processing r. If no match is found with any of the R'_i records, r is added to R'_i. The algorithm terminates when all processors have resolved all the records in their scope, and the messages they sent have been received and processed. The final answer is the union of the R'_i sets at the processors.

Consider running D-Swoosh on our 6 record example dataset. Recall that our example scope function (Figure 2) assigns to P_0 records $\{r_0, r_2, r_3, r_5\}$, while P_1 gets $\{r_0, r_1, r_3, r_4\}$ and P_2 gets $\{r_1, r_2, r_4, r_5\}$. Let us focus on the actions at P_0. Say P_0 is responsible for the r_0, r_3 comparison, but not the r_2, r_5 one. When P_0 merges r_0, r_3 into r_6, it sends $-(r_0)$, $-(r_3)$ messages to itself and P_1 (the processors that have these records in their scope). It also sends $+(r_6)$ to the processors in scope(r_6), i.e., P_0 and P_1.

In the meantime, P_0 will be getting messages from other processors. For example, the processor responsible for the pair r_2, r_5 will send to P_0 messages $-(r_2)$, $-(r_5)$ when those records merge. As the messages arrive at P_0, they are processed. For instance, when the $-(r_2)$ message arrives at P_0, record r_2 is removed from R'_0, if it is there. Note that r_2 may not be in R'_0, e.g., the initial $+(r_2)$ message may not have arrived. This is why the $-(r_2)$ message is "remembered" in the set D_0: when r_2 finally arrives, it will be ignored because r_2 is in D_0. Records r_6 and r_4 will be merged at processor P_1 to form r_8, $+(r_8)$ will be sent to processors r_0 and r_2.

Theorem 3.2 *Given a set of records R, the D-Swoosh algorithm terminates and computes $ER(R)$.*

Proof In [2], we show that given a set of records R, $ER(R)$ can be computed by any maximal derivation sequence of R. A derivation sequence is a sequence of merge steps (addition of a merged record) and purge steps (deletion of a dominated record). A derivation sequence is maximal if no additional merge or purge steps can be performed.

To prove the correctness of D-Swoosh, we show that it computes a maximal derivation sequence of the set $I = W \cup \bigcup_{P_i} R'_i$, where W is the set of all records pending processing at any of the processors. Since I is initially equal to R and $W = \emptyset$ when the algorithm terminates, this will effectively establish the correctness of D-Swoosh.

We first prove that each event handled by a processor corresponds either to a merge or a purge step on I, or keeps I unchanged. We then prove that the derivation sequence is maximal, and that the algorithm terminates.

When processor P_i handles a $+(r)$ message, r is either added to R'_i or a $-(r)$ message is sent to one or more processors. In both cases, r remains in I. If r matches a record r' in R'_i, the two records are merged into r'', and $+(r'')$ is sent to one or more processors (hence added to W). In case r'' was not present in I before (e.g., at some other processor), the algorithm has performed a merge step: a record obtained by merging two records previously present in I is added to I.

When processor P_i handles a $-(r)$ message, r is removed from R'_i, hence r is removed from I if no other processor still has r or is waiting to process r. It is easy to see that $-(r)$ messages are only sent when r is known to be dominated by another record in I. Therefore, if r is effectively removed from I, the algorithm has performed a purge step.

D-Swoosh performs a sequence of merge and purge steps on I, and therefore computes a correct derivation sequence of I. We must now show that the derivation sequence is maximal, i.e., once I stops evolving, no further merge or purge step is possible. We will then show that the algorithm terminates. Observe that for any record r, all messages relevant to r ($+(r)$ or $-(r)$) are sent exactly to the set of processors in $scope(r)$. Therefore the only processors to ever manipulate r are precisely those in $scope(r)$.

Suppose I does not evolve anymore, and a merge step is still possible, say between records r_1 and r_2. W.l.o.g. we can assume that r_1 and r_2 are not dominated by any other records in I. Therefore a $+(r_1)$ (resp. $+(r_2)$) message is received at some point by all the processors in $scope(r_1)$ (resp. $scope(r_2)$). By the coverage property, one of the processors in the intersection of these scopes (say, P_i) is responsible for the pair (r_1, r_2). Upon receiving the first of these two records (say, r_1), P_i adds r_1 to R_i, because it is not dominated. When P_i receives r_2, it compares it with r_1 and performs the merge step, a contradiction.

Suppose now that I does not evolve, but a purge step is possible, say record r_1 is dominated by record r_2. Again, by the coverage property r_1 and r_2 are handled by at least one processor P_i. Processor P_i must send $-(r_1)$ to all processors in $scope(r_1)$. After all processors have handled this message, r_1 is not in W. Moreover, r_1 is not present in any of the R'_i sets, and cannot reappear because it was added to the D_i sets. Therefore, the purge step would effectively have been performed, again a contradiction.

No merge or purge step is possible on the final state of I, hence the algorithm computes a maximal derivation sequence. We now show that the algorithm terminates. Observe that after the algorithm has computed the maximal derivation sequence, no two records may match. Indeed, if two records match, then a merge or purge step would necessarily follow, a contradiction. As a consequence, no more "send" messages are exchanged by processors. Any record in W disappears from there once it has been processed by all the processors in its scope. W eventually becomes empty, and all processors become idle, hence the algorithm terminates. □

3.2.1 Lineage Optimization

The algorithm presented above can be improved by considering the *lineage* of records, i.e., their derivation history. The properties of the match and merge functions entail that some pairs of records can be determined to match without even comparing them, just by looking at the sets of records they were respectively derived from.

More precisely, let the *base* of r, be the set of original records r was derived from. $base(r) = \{r\}$ for input records, and if $r = \langle r_1, r_2 \rangle$, then $base(r) = base(r_1) \cup base(r_2)$. The following properties can be shown to hold:

- If $base(r_1) \subseteq base(r_2)$ then $r_1 \leq r_2$ (in particular, two records with the same base are identical).
- If $base(r_1) \cap base(r_2) \neq \emptyset$ then $r_1 \approx r_2$.

The first property means that if a record r_1 was derived from a subset of the records a record r_2 was derived from, then r_1 is dominated by r_2. The second property says that if r_1 and r_2 have some base records in common, then they are known to match and can be merged right away.

The D-Swoosh algorithm does not even need to be modified to leverage this lineage information. In fact, for each record r, $base(r)$ can be kept as an extra attribute. The match function can be replaced by a new match function that first checks for a non-empty intersection on the bases of records, and only if the intersection is empty evaluates the original match function. Similarly, the merge function can be extended to check for an inclusion relationship (and return the non-dominated record immediately) and to construct the base of merged records.

4 Choosing scope and resp

We now consider particular scope and resp functions to distribute the ER work among a set of processors. We start with strategies that are always applicable because they do not make any extra assumptions, then turn to strategies which exploit existing semantic knowledge.

4.1 Strategies Without Domain Knowledge

In this section, we consider strategies in which we do not have any a priori domain knowledge about records, and therefore must consider all possible matches between records. Our goal will be to distribute the work evenly across processors, while trying to reduce communications and redundant computations. We present three schemes: full replication, majority (which generalizes our initial example) and grid. We do not prove it here, but it is not difficult to see that the scope and resp functions given in Tables 1, 2, and 3 satisfy the coverage property of Definition 3.1.

4.1.1 Full Replication Scheme

In the full replications scheme, records are sent to all p processors. The scheme is defined is Table 1, using the same style that will be used for the other schemes. To select the processor that is responsible for comparing records r_i and r_j, we select the smallest of the two indexes, i or j, and use it to identify the processor. If records identifiers are evenly distributed, this scheme ensures that each processor has a similar number of pairs to compare.

The full replication scheme clearly has a high storage overhead, so our next two schemes will reduce storage costs. However, as shown by our experiments (Section 5), full replication has benefits under some of the other metrics we consider.

4.1.2 Majority Scheme

The majority scheme (Table 2) generalizes the scope and resp functions we used in the introduction, from 3 to an arbitrary number p of processors. The essential idea is that if any record has a scope that consists of more than half (i.e., a majority) of the processors, then any pair of records will share at least one common processor in their scopes. We also generalize the resp function such that any pair of record is the responsibility of exactly one processor. With the majority scheme, every processor receives and stores about half of the records in the dataset.

P_0	P_1	P_2	P_3	P_4	P_5	P_6
b_0	b_0	b_0	b_0			
	b_1	b_1	b_1	b_1		
		b_2	b_2	b_2	b_2	
			b_3	b_3	b_3	b_3
b_4				b_4	b_4	b_4
b_5	b_5				b_5	b_5
b_6	b_6	b_6				b_6

Figure 3: Scope for the majority scheme

Figure 3 illustrates the majority scheme with $p = 7$ processors. The records are partitioned into 7 buckets, b_0 through b_6, using hashing or modulo arithmetic. Each bucket b_x is distributed to 4 processors, starting with processor x. Additional records would follow the same pattern. For example, if r_{11} falls into bucket b_4, the record is sent to P_4, P_5, P_6, and P_0. Similarly, we see that processor P_2 holds buckets b_0, b_1, b_2 and b_6.

As shown in Table 2, there are three cases to consider in determining where the scopes of two records intersect. For instance, consider two records in buckets b_1 and b_3 respectively. This situation corresponds to the first case since $x - y$ (i.e., the difference in bucket indexes) is $3 - 1$, which is less than half the processors. Here, the scopes overlap at $m = 2$ processors: P_3 and P_4. The resp function then uses the smallest record identifier to select one of these two processors.

4.1.3 Grid Scheme

With the grid scheme (Table 3), we again divide records into B disjoint buckets. For this scheme, we require that the number of processors be $B(B+1)/2$, which corresponds to the number of elements in half a matrix of size B, including the diagonal. For $B = 7$ buckets, we need 28 processors, which are arranged as illustrated in Figure 4.

	b_0	b_1	b_2	b_3	b_4	b_5	b_6
b_0	P_0						
b_1	P_1	P_7					
b_2	P_2	P_8	P_{13}				
b_3	P_3	P_9	P_{14}	P_{18}			
b_4	P_4	P_{10}	P_{15}	P_{19}	P_{22}		
b_5	P_5	P_{11}	P_{16}	P_{20}	P_{23}	P_{25}	
b_6	P_6	P_{12}	P_{17}	P_{21}	P_{24}	P_{26}	P_{27}

Figure 4: Processor arrangement in the grid scheme

For the scope, every record in bucket j is sent to the processors that appear on the j^{th} line and column of the (half) matrix. For instance, in our example, records in bucket 3 are sent to processors $\{P_3, P_9, P_{14}, P_{18}, P_{19}, P_{20}, P_{21}\}$. Observe that processors on the diagonal get records from exactly one bucket, while processors in other positions receive records from two buckets. For the resp function, the processor at the intersection of the i^{th} and j^{th} column is responsible for the pairs of records r, r' such that one of them is in bucket i and the other in bucket j. Hence, processors on the diagonal are responsible for comparing all the records in their scope, while other processors are only responsible for the pairs of records in their scope which belong to different buckets.

To illustrate, in our example, processor P_{18} has all the records of bucket 3 in its scope, and is responsible for comparing all of them pairwise. Processor P_{16} is at the intersection of line 5 and column 2 in the matrix, and therefore has all records of both buckets 2 and 5, but is only responsible for comparing the pairs belonging to different buckets.

Note incidentally that with the grid scheme, the D-Swoosh answer can be computed by taking the union of the records held by the processors on the diagonal only. This operation does not even require looking for duplicate records, as these sets are disjoint.

We can show that the grid scheme is within a factor of at most $\sqrt{2}$ of the initial storage costs of an optimal scheme, when the number of buckets B is large. For large B, the number of processors $p = B(B+1)/2$ approaches $B^2/2$, so $B = \sqrt{2p}$. With the grid, each of n input records is copied to B processors, so the total initial storage cost is $n\sqrt{2p}$ (compared to np for the full replication scheme, and $n(\lfloor p/2 \rfloor + 1)$ for the majority scheme).

In a space optimal scheme, a processor that has x records would perform at most all $x^2/2$ comparisons if no records match. For load balancing, each processor should have the same number of records, so the total number of comparisons across processors is at most $px^2/2$. Since we cannot miss any of the $n^2/2$ comparisons which happen among the initial records if no records match, we see that x^2 must be greater than or equal to n^2/p, or $x \geq n/\sqrt{p}$. The total storage cost for the optimal scheme must be at least this amount multiplied by p, or $n\sqrt{p}$. Thus, we see that the grid is within a factor of $\sqrt{2}$ of the optimal scheme for the initial storage cost.

4.1.4 Cost of the resp Function

In the introduction, we argued that calling the responsible function is inexpensive. The functions we have presented here are relatively cheap since they only involve simple arithmetic. Furthermore, for the grid scheme, the resp function can be checked simply by keeping the records at a processor in two areas. In particular, for non-diagonal processors (say, at position (i, j)), the set R' can be kept as two disjoint sets $R'|_i$ and $R'|_j$ holding the records that belong to each of the buckets respectively. When a new record arrives, it need only be compared to the records in the opposite bucket. For processors on the diagonal, resp is true for all pairs of records in the scope of the processor (which are the only records the processor gets to process), and hence can be skipped.

4.2 Strategies With Domain Knowledge

We now turn to strategies that use domain knowledge to distribute ER computations across multiple processors. Essentially, domain knowledge provides *necessary conditions* for pairs of records to match. In our example comparison shopping application, we may know that products may match only if they are in related categories, or if their prices are not too far apart. The key assumption is that checking necessary conditions (e.g., whether two products are in the same category) is much cheaper than invoking the full match function. With a single processor, the cost of ER can already be greatly reduced by only comparing pairs of records that satisfy the necessary conditions. This technique is widely used in a sequential setting, and commonly referred to as "blocking" in the ER literature [1].

With multiple processors, domain knowledge can be even more beneficial because groups of records that are known in advance not to match can be processed independently (and in parallel) by different processors. For instance, one processor can match the DVD records, another the cameras, and so on, assuming that merged records remain in their same category.

4.2.1 Groups

To exploit domain knowledge, we divide the records into G semantically meaningful groups, $g_0, g_1, ...g_G$. Each record r (in the original dataset, or derived through merges) belongs to one or more groups. (Note that the buckets used in Section 4.1 had no semantic meaning, and records could only be in one bucket.)

Figure 4 summarizes the scope and resp functions we use with groups. As we can see, the scheme is very simple: each group is assigned to one processor, and one processor gets only one group. With buckets (Section 4.1), the scope function ensured that any pair of records r_i, r_j would show up at some processor for comparison. With groups, there is no such guarantee: r_i and r_j will be at a common processor only if they happen to be in a common group. Hence, now it is the burden of the semantic group assignment function to ensure that if there is any chance that r_i and r_j may match, then they should be assigned to some common group. This property can be expressed as follows.

Group property: *If two records (initial or merged) r_i and r_j may match, they must be assigned to at least one common group.*

The above group property ensures that the coverage property of Definition 3.1 holds (in spite of the trivial scope function), so D-Swoosh correctly computes $ER(R)$. We now show how groups that satisfy this property can be derived for common forms of domain knowledge: value equality, hierarchies and linear ordering.

4.2.2 Value Equality

A common situation in ER is when pairs of matching records share a common value for some attribute, which we call the "equality" attribute. For instance, product records may all have a "category" associated with them, and two records may only match if they are in the same category. For flexibility, records can have multiple values for the equality attribute (e.g., a product may be a sporting good and a clothing item). In this case, pairs of records match if they share at least one value for the equality attribute, in the spirit of canopies [15]. A merged record inherits the equality values of its source records (to ensure representativity holds).

One possible strategy for value equality is to have one group per value, e.g., one group for cameras, one for DVDs, and so on. However, in some applications we may have many category values, and hence we may need too many groups/processors. Instead, a *partition* of the values can be used to determine the groups. For instance, cameras, MP3 players, etc. can go into one partition, while shoes, shirts, jackets, etc. can go into another. The group property continues to hold, as products in the same category continue to be in the same group. The partition can either be arbitrary, or semantically meaningful as in our example. A semantic partition is advantageous if it increases the chances that a multi-attribute record falls within one group. For instance, with a semantic partition that includes cameras and telephones, a camera phone will be in one group; with an arbitrary partition, the camera phone record may have to be sent to two processors.

4.2.3 Hierarchies

Another common form of domain knowledge is when the values of some attribute, called the "hierarchy" attribute, are related through a hierarchy. For example, vehicles may be cars or trucks; Cars may be sedans, hatchbacks, station-wagons, etc.; Trucks may be pick-up, vans, etc. A pair of records may only match if one hierarchy attribute is a descendant of the other. For instance, a sedan may match a car, but not a truck. When two records match, the merged record inherits the most general of the hierarchy values (again, to ensure representativity). (If a record has no value for the hierarchy attribute, we can assign it the root value.)

To handle hierarchies, we create one group g_t for each term t of the hierarchy. A record with value t gets assigned to group t, as well as to all groups g_s where s is a descendant of t in the hierarchy. To illustrate, a car record will belong to the car and sedan groups; a sedan record will only belong to the sedan group.

4.2.4 Linear Ordering

A last form of domain knowledge we consider here is a linear ordering of what we call a "window" attribute. In this scenario, two records may match only if their values are within some window δ. For instance, product records may have a price attribute, and records whose prices differ by more than \$50 cannot represent the same product. The window size may also be relative to the values, e.g., one price must be within some percentage of the other, or may be determined based on the density of the records on the window attribute, e.g., a product record may only match with the n cheaper or more expensive records. Records may also have multiple values for the window attribute, in which case they may only match if at least one of the values for each of the records satisfy the window condition.

We form groups by dividing the range of window values into p partitions (recall that p is the number of processors and of groups). Say the i^{th} partition has lower bound L_i (inclusive) and upper bound L_{i+1}. Also, let δ be the window size, i.e., two matching records must have window attribute values distant by less than or equal to δ. A record r belongs to group g_i if one of the r values for its window attribute is in the interval $[L_i, L_{i+1} + \delta)$.

Notice that if δ is small relative to the partition sizes and records values for the window attribute are evenly distributed, only a "few" records that happen to have values close to the boundaries are replicated. For instance, if the boundaries are at prices 0, 1000, 2000, … dollars and δ is \$50, a record with price \$1049 will be in two groups, but a record with price \$1050 (to \$1999) will not be replicated. As δ grows, more records are replicated, into possibly more than two groups. For instance, if δ is 2000, a record with price \$2500 is in three groups. Thus, there is an interesting interplay between the application parameters (range of values, window size) and the number of processors we may have available or we may need to scale-up performance.

4.3 "Tuning" scope and resp

So far we have presented scope and responsible functions that try to distribute records and responsibility "uniformly" across processors. However, in some cases that may not be the best approach. For example, consider three records r_1, r_2 and r_3 that all match pairwise, and will eventually be merged into record r_{123}. In general, each pair of initial matches will be discovered by different processors, e.g., P_1 will generate $\langle r_1, r_2 \rangle = r_{12}$, P_2 will generate $\langle r_1, r_3 \rangle = r_{13}$, and P_3 will generate $\langle r_2, r_3 \rangle = r_{23}$. The new records will then be merged at other processors, e.g., P_4 will find $\langle r_{12}, r_{13} \rangle = r_{123}$, P_5

will find $\langle r_{12}, r_{23} \rangle = r_{123}$, P_6 will find $\langle r_{23}, r_{13} \rangle = r_{123}$. Of course, some of these processors may be the same, and such coincidences improve performance. For instance, if $P_4 = P_5$, one useless generation of r_{123} is avoided. Similarly, if $P_1 = P_2 = P_4 = P_5$, then r_{123} is found quickly by P_1 without any message delays. (The faster P_1 propagates $+(r_{123})$, the more unnecessary comparisons we can avoid.)

Our scope and resp functions differ in how they distribute records, and hence the number of "lucky coincidences" like the ones illustrated above varies. As a matter of fact, in our experiments we will see that a scheme like the full replication can do less work than Majority because merges are disseminated faster and there is less redundant work, even though the full replication is making more copies of records than Majority is.

If we change how records are assigned to buckets, or if we change the responsible function a bit, we may even be able to increase the chances that the lucky coincidences happen more often. For example, say a merged record is assigned to buckets using the smallest identifier in its lineage. In our example, r_{12} and r_{13} would fall in the same bucket as r_1. Since the scope of these three records would be identical, this mapping increases the chances that $P_4 = P_1$ and $P_5 = P_1$. We can increase the chances further by not selecting the responsible processor randomly. For instance, say processor P_k is responsible for r_i, r_j if it is the processor with the smallest identifier that is in both the scope of r_i and of r_j. With this additional change, it is certain that $P_4 = P_1$ (P_1 is the processor with the smallest identifier in the scope of r_{12}, and hence it is also the smallest processor in the intersection of the scopes of r_{12} and r_{13}) and that $P_5 = P_1$. Thus, we expect that merges will be discovered "faster," without as many message exchanges. On the other hand, the load may not be uniformly distributed (e.g., the processor with the smallest identifier will tend to get more work), so each alternative needs to be carefully evaluated.

When semantic knowledge is available, we can again modify the scope and responsible functions to try to discover merges with fewer message delays. For example, consider distance proximity, and some records that have close values for their window attribute and lie close to a partition boundary. In this case, it makes sense to have one processor out of the ones that share that overlap region be responsible for comparisons, instead of having all the processors share the load.

5 Experiments

We implemented the D-Swoosh algorithm, and the various choices of scope and resp functions discussed in the previous sections, and conducted extensive experiments on subsets of a comparison shopping dataset from Yahoo! In this section, we describe our implementation and experimental setting, and report the main findings of our experiments. Additional results from an implementation that simulates parallel operation can be found in Section 6.

5.1 Experimental Setting

We ran our experiments on subsets of a comparison shopping dataset provided by Yahoo!. The dataset consists of product records sent by merchants to appear on the Yahoo! shopping site. It is about 6Gb in size, and contains a very large array of products from many different vendors. Such a large dataset must be (and is in practice) divided into groups for processing. For our experiments we extracted a subset of about 138,000 records consisting of up to 50,000 records for each of four keywords: "iPod", "mp3", "book", and "dvd". From this subset, we randomly selected subsets varying in size, starting with a size of 5,000 records.

Each of these smaller datasets represents a possible set of records that must be resolved by one or more processors. If we are modeling a scenario with no semantic knowledge, then all records in the dataset must be compared. If we are considering a scenario where semantic knowledge is available, the database can be further divided by category.

Our match function considers two attributes of the product records, the title and the price. Titles are compared using the Jaro-Winkler similarity measure [23], to which a threshold t is applied to get a yes/no answer. Prices are compared based on their numerical distance, and match if one is within some percentage α of the other. Two records match if both their titles and price match. We experimented with different t and α values, and selected ones that worked reasonably well with this data ($t = 0.95$, $\alpha = 0.33$). In one of our experiments (see below) we vary t to model different applications where more or fewer records would match. To merge pairs of records, we simply took the union of distinct values for each attribute.

We implemented the D-Swoosh algorithm in Java 1.5, using TCP connections for peer-to-peer message passing. To avoid stalling due to TCP flow control, sending and receiving messages were separated into different threads. For our experiments we ran the code on up to 15 approximately identical machines, all connected via Ethernet. Each machine had a Pentium 4 2.8GHz CPU with hyperthreading enabled, and 2GB of RAM. We ran the code under the Sun Java VM 1.5.0_05-b05, on a GNU/Linux system using kernel 2.6.11 with SMP enabled.

To evaluate the performance of D-Swoosh, we use the following metrics, which can be precisely evaluated in our environment.

- Runtime: The total amount of time to distribute the records to the peers, compute the result, and detect the termination of the algorithm. This metric captures the raw performance of the algorithm under the specified conditions.

- Aggregated computation: The main cost of ER is the pairwise comparisons of records, so we count the overall number of comparisons performed by all processors. When multiple processors are used, this metric captures the overhead of parallelism.

- Communication: We count the total number of bytes sent across the network by the processors, and use this as a metric of the overall communication cost. We focus on the communication cost generated by the actual ER process, and do not count the initial distribution of records to processors, nor the gathering of the result after termination.

- Storage: We measure the maximum number of records in the R'_i set of each processor during the execution, and sum across all processors to get the overall storage cost.

5.2 No Domain Knowledge

We start by studying the performance of the schemes we presented in Section 4.1, which do not assume any form of domain knowledge. We ran D-Swoosh on a 10,000 record dataset, generated as described in Section 5.1. All the schemes compute the same ER result, which contains 9,715 records. This indicates that relatively few record matches were found—a fact we attribute to the random sampling, which can greatly reduce the percentage of duplicate records in a set. We compared the Full Replication, Majority, and Grid schemes, while varying the number of processors from 1 to 15.

Figure 5 gives the aggregated computation as a function of the number of processors, for each of the schemes. Note that the (approximately) 50 million comparisons observed for one processor corresponds to the number of comparisons performed by the sequential R-Swoosh algorithm. A few of the schemes perform slightly fewer comparisons. This is surprising, considering that the sequential algorithm is optimal. However, the optimality of the sequential algorithm refers to the expected number of computations when all orderings of the input are considered, not on one given ordering. Distributing the records across different processors can result in a more "lucky" ordering that reduces the number of comparisons.

The vertical scale in Figure 5 starts at 49.6 million comparisons, so the good news is that the extra work as we distribute the load is not excessive. In the worst case we perform roughly 2% extra comparisons. As the number of processors grows, each of the schemes starts performing redundant comparisons, even though each unique comparison is the responsibility of one and only one processor. Intuitively, the sequential R-Swoosh algorithm is efficient because it is able to perform merges and deletions as early as possible. This benefit is gradually lost as records are spread out randomly across processors, as discussed in Section 4.3. In the figure we see that the different distribution schemes generate similar workloads, and that surprisingly the Full Replication scheme does very well (with a higher storage cost, see below).

We now consider runtime, which captures the true benefit of parallelism obtained from distributing ER across multiple processors. As shown by Figure 6, all schemes benefit from having more processors by reducing the computational cost for each processor, and therefore for the ER computation overall. The runtime of a single processor goes down from around 700 seconds for the sequential case, to about 150 seconds with 15 processors. The lowest line in the graph suggests how an algorithm would perform if it had perfect scalability. That is, if the runtime of the algorithm were simply the runtime for a single processor divided by the total number of processors. The Majority and Grid schemes both stay quite close to this line up through 15 processors.

The Full Replication scheme shows the most improvement for small numbers of processors, but as expected, the scheme does not scale well to larger numbers of processors, as the increase in network bandwidth begins to have an effect on the runtime. The irregularity of the Majority scheme is due to the fact that the scheme works better for odd numbers of processors than for even numbers. For low numbers of processors, the Grid scheme is slightly less efficient than the two others because the processors on the diagonal of the grid perform less work than the others. This difference evens out as the number of processors increases, hence the runtime of the Grid scheme fares better with larger numbers of processors.

Figure 7 gives the storage cost for each scheme as a function of the number of processors. The Full Replication strategy sends every record to every processor, so its storage cost grows linearly. In the Majority scheme, every

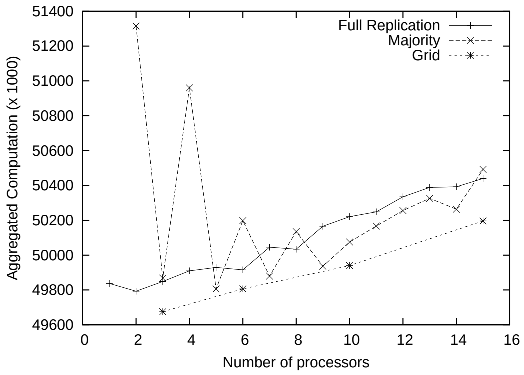

Figure 5: Aggregated computation cost for strategies without domain knowledge

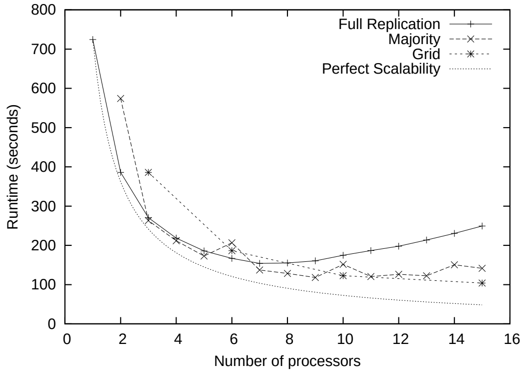

Figure 6: Runtime; no domain knowledge

record is sent to a majority of the processors, so the storage cost also grows linearly, but only stepping up on transitions from odd to even numbers of processors. The Grid scheme performs much better, as the storage cost grows only proportionally to $\sqrt{p}$. It is in fact the only scheme where the storage requirements *per processor* decreases arbitrarily as the number of processors increases.

Figure 8 gives the total number of bytes sent during ER, as a function of the number of processors, for each of our schemes. Intuitively, there are two factors that impact communication costs. More record replication implies more communications, as merged records have to be sent to more processors. On the other hand, as we have discussed, replication may cause merges to occur earlier, which reduces redundant work and unnecessary communications. Our experiments show, however, that this second factor does not play a significant role in the communication cost. The communication cost for the algorithms appears strictly proportional to the amount of record replication.

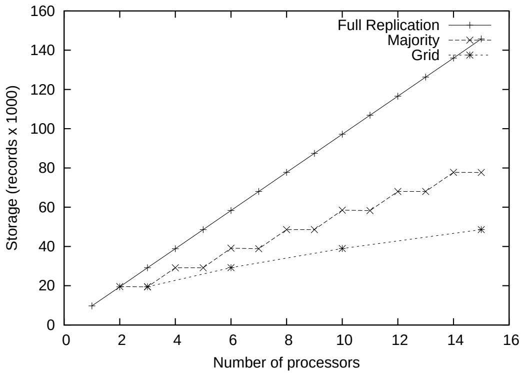

Figure 7: Storage cost; no domain knowledge

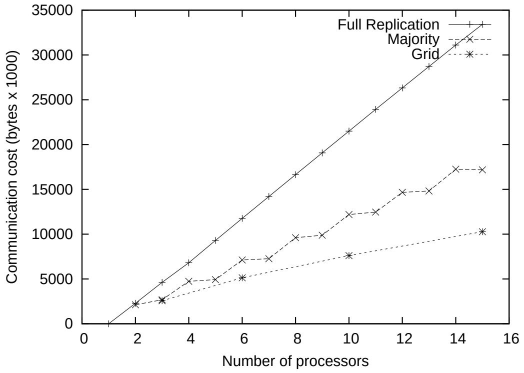

Figure 8: Communication cost; no domain knowledge

To summarize the results of this section, we find that parallelism is very effective for increasing the performance of the ER process. Our three strategies all perform quite well, however Majority seems to be the best selection for a small number of processors, and Grid is best for larger number of processors.

5.3 With Domain Knowledge

We now consider schemes with domain knowledge, and focus here on Linear Ordering on the price and Value Equality on a product category field. For these experiments, we used the 10,000 record data set created as described in Section 5.1. These records fall into a total of 48 categories. (A product is only in one category in our input data.)

For the Linear Ordering scheme we partition the total price range into p groups corresponding to intervals of equal size. Since the items vary in price from \$0 to \$200, we settle on a value of \$2 for the window size. For Value Equality, we partition at random (but as evenly

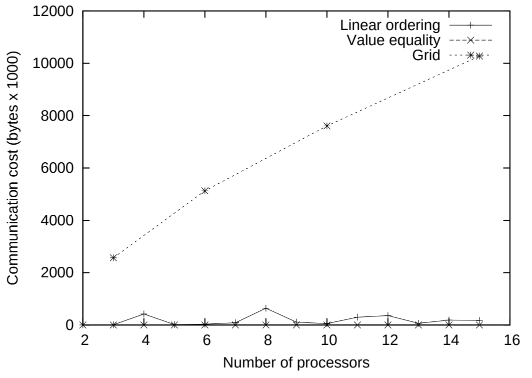

Figure 9: Communication cost for domain knowledge schemes

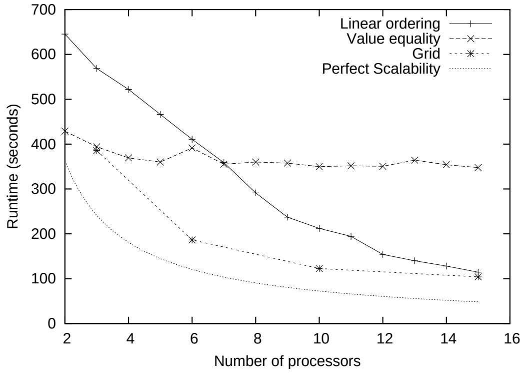

Figure 10: Runtime for domain knowledge schemes

as possible) the 48 categories into p groups. Note that some groups may get one more category than another group, and that some processors may get more popular categories. The same is true with Linear Ordering: some price partitions may have many more records than others. Note that these approaches may miss results that would have been found in the schemes that do not exploit semantic knowledge.

Figure 9 illustrates the communication costs incurred by both strategies, with the Grid line included for comparison. Because the category of records is preserved through merges, groups remain disjoint, and the Value Equality scheme has no communication cost. Linear ordering does result in some communication, but it is relatively small and constant up through 15 processors. We would expect an increase in communication as the number of processors goes higher, since the constant window size becomes larger relative to the shrinking bucket size. However, this effect appears to be negligible in our experiments.

Figure 10 gives the runtime for both strategies as a function of the number of processors. Again, we included the Grid line from Figure 6 for comparison. The results shown are highly surprising. Not only do the two algorithms poorly utilize the extra processors, but they also fail to perform better than the Grid scheme, which makes no use of domain knowledge. Ironically, making use of extra information seems to have hurt, rather than helped, our performance.

The main reason for this behavior is that processing load is not evenly distributed: although there were 48 categories, one of those categories contained over one third of the records in the data set. In our experiments with Value Equality, all of the processors consistently ended up waiting for the processor with the huge category to complete. For the Linear Ordering scheme as well, there is a high concentration of products in some small price ranges, so extra processors are not as efficiently used as they are in the Grid scheme.

Our results show that fully exploiting domain knowledge may be tricky. One solution may be to adjust the partitions: if value distributions are stable, and we know the number of processors in advance, we may analyze the data and determine good semantic partitions that even out the load. Another possibility is to use a hybrid scheme, e.g., using the Grid scheme to distribute the work of heavily loaded processors onto multiple processors. But if runtime is the critical metric, it may be simpler to just use the “dumb” Grid scheme, which does extremely well.

Results obtained for other metrics are omitted here for space reasons. To summarize, the storage cost of the Value Equality scheme is essentially constant, as groups do not overlap. For the Linear Ordering, it is proportional to the window size α, and grows roughly linearly with the number of processors.

5.4 Scalability

To assess the scalability of our algorithms, we compared performance on input datasets of different sizes. We started by generating a 138,000-record set as described in Section 5.1. We then generated 5,000, 10,000, 20,000, 40,000, and 80,000-record sets by randomly removing records from the first set. Thus, roughly the same *fraction* of records appears in each category (and price range) in all four datasets

In Figure 11 we show the runtime as a function of the size of the initial dataset. The curve labeled “sequential” is for one processor; the other curves are for a 10-processor system using Grid, Linear Ordering and Value Equality schemes. We stopped testing at the 40,000 record point to save time, but the Grid scheme was fast enough

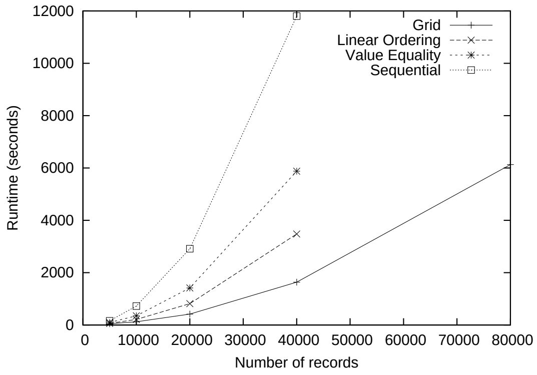

Figure 11: Runtime evolution with dataset size (10 processors)

to do a final test with 80,000 records. For all algorithms, the cost evolves quadratically with the number of records, and this is an inherent characteristic of ER. However, this evolution is much slower for parallel strategies than for the sequential one, and the gain from using parallelism increases with the size of the dataset. Here, for 20,000 records, the Grid scheme beats the sequential scheme by more than a factor of 6.

Finally, note that the *relative* ordering of the schemes in Figure 11 (and in other scalability experiments we conducted in Section 6) remains fixed. This fact suggests that our conclusions regarding the strengths and weaknesses of the schemes will hold for datasets larger than what we were able to consider in our study.

6 Simulated Experiments

We also implemented a variant of the D-Swoosh algorithm that runs on a single CPU and simulates the results of running the algorithm on an actual set of CPUs. In this section, we describe this implementation and report the findings of our simulated experiments.

6.1 Simulated Experimental Setting

We ran our experiments on subsets of the Yahoo! comparison shopping dataset. The datasets are not identical to those in Section 5, but they were created using a similar process. The match function we used was identical, except for the use of a different value for the price threshold α: 0.33.

We implemented a variant of the D-Swoosh algorithm where processors work in rounds. During a round, a processor compares all pairs of records received in the previous round (for which it is responsible), and sends out all messages in a batch at the end of the round. Rounds are advantageous because there are fewer context switches: a processor can do more work without constantly sending and receiving messages. Similarly, communication overhead is lower because messages are batched. On the other hand, working in rounds may be counter productive because merged records are propagated later in time, possibly causing some redundant work at other processors. A processor performs all comparisons it can with its local records, and the round ends when all processors have no more records to compare. It is possible to have shorter rounds (and in the extreme the one-message-per-merge approach of D-Swoosh), but we do not evaluate the options here.

Our code was run in an emulation environment where it was easier to instrument. That is, we emulated the p processors within a single processor. When one emulated processor completed its round, control was given to the next processor, and so on. Messages were exchanged through shared memory. The emulation environment kept track of statistics like the number of comparisons performed by each processor and the number of messages exchanged. The emulation environment and the ER code were all implemented in Java, and our experiments were run on a quad processor server with AMD Opteron Dual Core processors and 16GB of RAM.

To evaluate the performance of D-Swoosh in the emulation environment, we use the same metrics as in Section 5, but with the following modifications:

- Maximum Effort: In the emulation environment, a single processor does all of the processing, so runtime is not a good metric for determining how well the algorithms scales with greater numbers of processors. Instead of runtime, therefore, we use a “maximum effort” instead. In each round, processors wait for the processor that performs the largest number of comparisons. (Recall that all comparisons within a round are done independently from the other processors.) Thus, the maximum number of comparisons (over all processors) reflects the time a round would take. If we sum these numbers over all rounds, we get a value that reflects the total running time. We call our metric “maximum effort” since its units is comparisons and not seconds.

- Communication: Although communication is still an important metric in the emulation environment, it is difficult to measure communication cost in terms of bytes, since the messages are exchanged via shared memory. Therefore, we count the total number of messages (additions and deletions of records) exchanged by the processors in each communication round, and sum across all communication rounds to get the overall communication cost. As before, we do not count the initial distribution of records to processors, nor the gathering of the result after termination.

6.2 Simulated No Domain Knowledge

We start by studying the performance of the schemes we presented in Section 4.1, which do not assume any form of domain knowledge. We ran D-Swoosh on 5000 records which all contain the string "iPod" in their title. All the schemes compute the same ER result, which contains 3983 records. We compared the Full Replication, Majority, and Grid schemes, while varying the number of processors from 1 to 15.

Figure 12 gives the aggregated computational cost as a function of the number of processors, for each of the schemes. Note that the 12 million comparisons observed for one processor corresponds to the number of comparisons performed by the sequential R-Swoosh algorithm. This figure displays characteristics similar to those in Figure 5 from Section 5.2. However, the increases in aggregated computation are greater in our simulation, which is likely due to the use of rounds. The reduced aggregated computation from the experiments in Section 5 suggests that the D-Swoosh algorithm benefits from instantaneous propagation of messages.

Note that the vertical scale in Figure 12 starts at 12 million comparisons, so the extra work is still not excessive. In the worst case (Majority, 15 processors) we perform roughly double the number of comparisons. Of course, in that case we have 15 processors that should be easily able to accommodate the extra load.

We now consider the maximum effort metric in Figure 13. All of the schemes scale well, as the maximum number of comparisons performed by a processor goes down from 12 million for the sequential case, to about 2 million with 15 processors.

These results are again comparable to those in Section 5.2. The difference here is that the Full Replication scheme maintains its lead over the other schemes, as expected. Unlkike runtime, the maximum effort metric does not take into account the cost of communication, which grows quickly with the Full Replication scheme.

In Figures 14 and 15, we present the storage and communication costs of the different schemes. The results are largely the same as those in Section 5.2.

Our previous experiments all used the exact same match function and produced the same final answer. By varying the selectivity of the match function (t) we can "simulate" different applications where more or fewer records match. In Figure 16 we show the results of such an experiment: we modify the threshold on the title similarity, and measure, for a fixed number of processors (6), the maximum effort and communication cost for all three strategies. The results show that all schemes behave in a similar fashion as the selectivity increases: The maximum effort decreases, because there are fewer merges, and hence fewer records to compare, and the communication cost also decreases, as there are fewer messages to propagate record creations and deletions. Similar results are obtained by varying the price threshold. Also note that the relative ordering of the schemes remains fixed. Thus, this result (and others we obtained) suggest that the strengths and weaknesses we have observed for the different schemes would continue to hold in other application

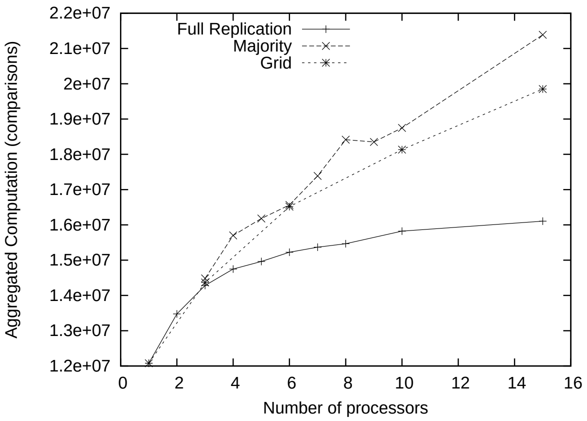

Figure 12: Aggregated computation cost for strategies without domain knowledge

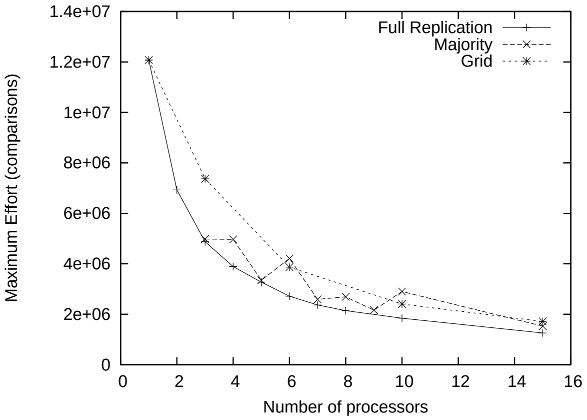

Figure 13: Maximum effort cost; no domain knowledge

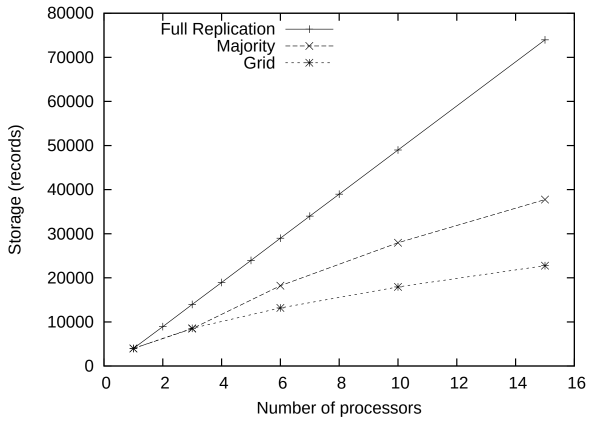

Figure 14: Storage cost; no domain knowledge

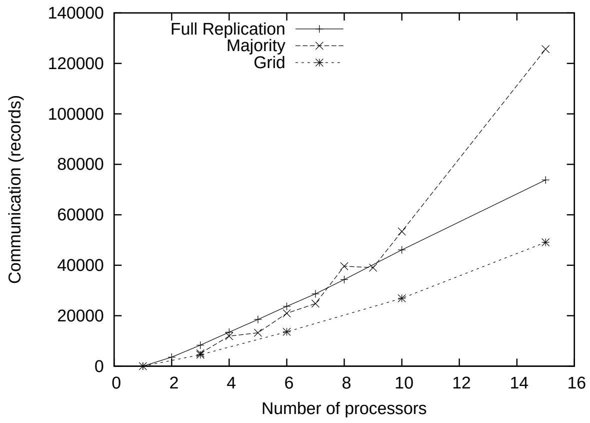

Figure 15: Communication cost; no domain knowledge

domains.

6.3 Simulated With Domain Knowledge

We now consider the Linear Ordering and Value Equality schemes in the simulation environment. We extend our match function so that only similar records ($t = 0.95$, $\alpha = 0.5$) in the same category can match. We use a new dataset that is more heterogeneous and hence amenable to partitioning by category: We extracted 5000 records which contain either “iPod”, “mp3”, “dvd” or “book” in their title. These records fall into a total of 37 product categories. (A product is only in one category in our input data.)

Figure 17 gives the maximum effort cost for both strategies (as points), and the communication cost for Value Equality (as bars, scale on the right vertical axis), as a function of the number of processors. Because the category of records is preserved through merges, groups remain disjoint, and the Value Equality scheme has no communication cost.

Title	**Maximum effort cost**		
threshold	Full replication	Majority	Grid
0.9	2886786	5318328	4752259
0.95	2716307	4209592	3864668
0.99	2605656	4083153	3620652

Title	**Communication cost**		
threshold	Full replication	Majority	Grid
0.9	44485	53079	31053
0.95	23730	20932	13633
0.99	17545	15756	9798

Figure 16: Maximum effort and communication cost as a function of selectivity

When this data set is resolved on a single processor that does not exploit domain knowledge, the total number of comparisons is 1.2 million. In Figure 17 we see that a single processor can do much better with semantic knowledge: fewer than 2 million comparisons with Value Equality, and fewer than 4 million with Linear Ordering.

The maximum effort decreases as we add processors, but notice that the improvements are not dramatic after a few processors. In particular, for Value Equality there are minimal gains beyond 4 processors. This confirms the results of Section 5.3, that unevenly distributed records can foil the schemes that use domain knowledge as the sole method of distribution.

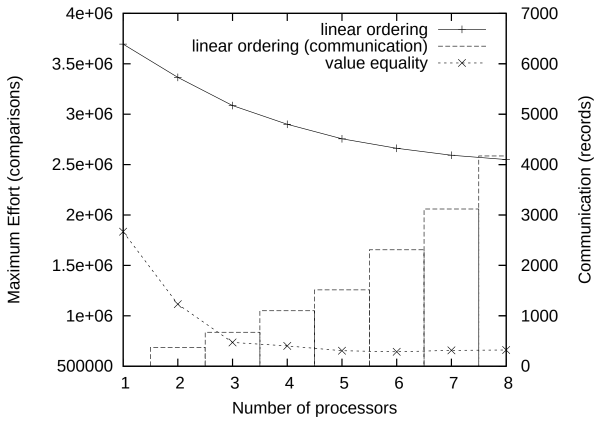

Figure 17: Parallel computation and communication costs for domain knowledge schemes

Results obtained for other metrics are omitted here for space reasons. To summarize, the storage cost of the Value Equality scheme is essentially constant, as groups do not overlap. For the linear ordering, it is proportional to the window size α, and grows linearly with the number of processors.

6.4 Simulated Scalability

To assess the scalability of our algorithms in the simulation environment, we compared performance on input datasets of different sizes. The process for creating the datasets was similar to that in Section 5.4, but we started with a smaller, 20,000-record set.

To illustrate a scenario where domain knowledge schemes miss matches, we return to our initial match function that only compares titles and prices ($t = 0.95, \alpha = 0.5$). Now records with differing category fields may match, but these matches will be missed by the Value Equality scheme (it only compares records within one group).

In Figure 18 we show the maximum effort as a function of the size of the initial dataset. (We also experimented with different numbers of processors and the remaining strategies; the results are analogous and not shown.)

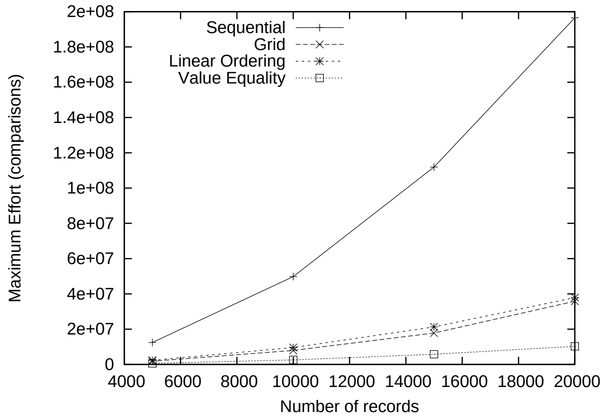

Figure 18: Maximum effort cost evolution with dataset size (10 processors)

The Value Equality scheme performs even better than the Grid and Linear Ordering, but there is a trade-off: because this scheme misses comparisons, it has lower recall. To illustrate, Figure 19 gives the size of the final answer for the different scenarios. The difference between the Grid/Linear and the Value Equality columns represents the number of records that Value Equality misses. The trade-off illustrated here is typical of what practitioners face: one can improve performance by pruning some comparisons and hence by reducing recall.

Initial records	Grid/Linear	Value Equality
5000	4717	4729
10000	9057	9102
15000	13553	13620
20000	17109	17247

Figure 19: Effect of Value Equality on ER result size

7 Related Work

Originally introduced by Newcombe et al. [17] as "record linkage", the ER problem was then studied under various names, such as Merge/Purge [12], deduplication [19], reference reconciliation [9], object identification [20], and others. Most approaches focus on the "matching" part, i.e. accurately finding the records that represent the same entities, using a variety of techniques, such as Fellegi & Sunter's probabilistic linkage rules [10], Bayesian networks [22], or clustering [16, 7]. Our approach encapsulates the outcome of such complex decision processes into a Boolean match function that decides whether two records represent the same entity or not. Iterative approaches [4, 9] identified the need for a "feedback loop" that compare merged records in order to discover more matches. Our approach eliminates redundant comparisons by tightly integrating matches and merges.

"Blocking" techniques use domain knowledge to prune the space of comparisons when performing ER. Canopies [15] construct overlapping subsets of the records similarly to our value equality scheme, and the sorted neighborhoods of [12] is very similar to our linear ordering scheme. An overview of such blocking methods can be found in [1]. With the exception of [12], these approaches assume a single processor. [12]'s "band join" parallelizes linear ordering like we do, but does not generalize to other forms of domain knowledge, and does not consider merging records.

In the distributed computing literature, we mentioned the connection of our "coverage condition", which guarantees the correctness of D-Swoosh to the distributed mutual exclusion problem [8], and schemes based on coteries [11]. Our majority scheme is based on the well-known principle of a quorum [21]. The grid scheme is a variant of Maekawa's construction in [14]. An important difference is that coteries designed for mutual exclusion try to maximize the number of operating nodes [18] or their availability [13]. By contrast, the coteries used in our strategies distribute work across processors, and are optimal when they minimize computation and communication.

Recently, Bilenko et al considered the ER problem on a comparison shopping dataset from Google [5]. Their focus is on continuously learning the appropriate match function for records, and is therefore complementary to our work, which focuses on executing matches and merges efficiently.

8 Conclusion

Entity resolution is an important problem that arises in many information integration scenarios. A lot of the work to date has focused on how to achieve semantically accurate results, e.g., how to ensure that the resulting records truly represent real-world entities in some application domain. Instead, we have focused on the also important problem of performance: given accurate (or accurate enough) match and merge functions, how can we distribute all the necessary work across multiple processors? We presented several schemes for distributed ER, both exploiting and not exploiting domain knowledge. Surprisingly, we discovered that simply minimizing the number of record replicas is not enough: schemes that make more copies may do better because they speed up the discovery of matching records. We also discovered that exploiting domain knowledge is tricky, as it requires a good distribution of records across semantic groups, and may involve a trade-off between good performance and lowered recall (missed answer records).

References

[1] R. Baxter, P. Christen, and T. Churches. A comparison of fast blocking methods for record linkage. In *Proc. of ACM SIGKDD'03 Workshop on Data Cleaning, Record Linkage, and Object Consolidation*, 2003.

[2] O. Benjelloun, H. Garcia-Molina, J. Jonas, S. E. Whang, Q. Su, and J. Widom. Swoosh: a generic approach to entity resolution. Submitted for journal publication. Available from http://dbpubs.stanford.edu/pub/2005-5.

[3] O. Benjelloun, H. Garcia-Molina, H. Kawai, T. E. Larson, D. Menestrina, and S. Thavisomboon. D-Swoosh: A Family of Algorithms for Generic, Distributed Entity Resolution (Extended version). Technical report, Stanford University, 2006. Available at http://dbpubs.stanford.edu/pub/2006-8.

[4] I. Bhattacharya and L. Getoor. Iterative record linkage for cleaning and integration. In *Proc. of SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery*, 2004.

[5] M. Bilenko, S. Basu, and M. Sahami. Adaptive Product Normalization: Using Online Learning for Record Linkage in Comparison Shopping . In *Proc. of IEEE Int. Conf. on Data Mining*, Houston, Texas, 2005.

[6] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[7] S. Chaudhuri, V. Ganti, and R. Motwani. Robust identification of fuzzy duplicates. In *Proc. of ICDE*, Tokyo, Japan, 2005.

[8] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965.

[9] X. Dong, A. Y. Halevy, and J. Madhavan. Reference reconciliation in complex information spaces. In *Proc. of ACM SIGMOD*, 2005.

[10] I. P. Fellegi and A. B. Sunter. A theory for record linkage. *Journal of the American Statistical Association*, 64(328):1183–1210, 1969.

[11] H. Garcia-Molina and D. Barbara. How to assign votes in a distributed system. *J. ACM*, 32(4):841–860, 1985.

[12] M. A. Hernández and S. J. Stolfo. The merge/purge problem for large databases. In *Proc. of ACM SIGMOD*, pages 127–138, 1995.

[13] T. Ibaraki, H. Nagamochi, and T. Kameda. Optimal coteries for rings and related networks. *Distrib. Comput.*, 8(4):191–201, 1995.

[14] M. Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Trans. Comput. Syst.*, 3(2):145–159, 1985.

[15] A. K. McCallum, K. Nigam, and L. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proc. of KDD*, pages 169–178, Boston, MA, 2000.

[16] A. E. Monge and C. Elkan. An efficient domain-independent algorithm for detecting approximately duplicate database records. In *Proc. of SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery*, pages 23–29, 1997.

[17] H. B. Newcombe, J. M. Kennedy, S. J. Axford, and A. P. James. Automatic linkage of vital records. *Science*, 130(3381):954–959, 1959.

[18] C. H. Papadimitriou and M. Sideri. Optimal coteries. In *PODC '91: Proceedings of the tenth annual ACM symposium on Principles of distributed computing*, pages 75–80, New York, NY, USA, 1991. ACM Press.

[19] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *Proc. of ACM SIGKDD*, Edmonton, Alberta, 2002.

[20] S. Tejada, C. A. Knoblock, and S. Minton. Learning object identification rules for information integration. *Information Systems Journal*, 26(8):635–656, 2001.

[21] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209, 1979.

[22] V. S. Verykios, G. V. Moustakides, and M. G. Elfeky. A bayesian decision model for cost optimal record matching. *The VLDB Journal*, 12(1):28–40, 2003.

[23] W. Winkler. The state of record linkage and current research problems. Technical report, Statistical Research Division, U.S. Bureau of the Census, Washington, DC, 1999.

```
input: A set $R$ of records
output: A set $R'$ of records

Initialization:
  $\forall P_i$, $R'_i \leftarrow \emptyset$
  $\forall P_i$, $D_i \leftarrow \emptyset$
  $\forall r \in R$ send($+(r), P_i$) to every processor $P_i$ in $S(r)$

on receive($+(r), P_i$):
  if $r$ not in ($R'_i \cup D_i$) then
    for all records $r'$ in $R'_i$ s.t. $resp(P_i, r, r')$ do
      if $r \approx r'$ then
        $r'' \leftarrow \langle r, r' \rangle$
        if $r'' = r$ then
          $\forall P_j \in scope(r')$, send($-(r')$, $P_j$)
        else
          if $r'' \neq r'$ then
            $\forall P_j \in scope(r')$, send($-(r')$, $P_j$)
            $\forall P_j \in scope(r'')$, send($+(r'')$, $P_j$)
          end if
          $\forall P_j \in scope(r)$, send($-(r)$, $P_j$)
          exit $+(r)$ handler
        end if
      end if
    end for
    add $r$ to $R'_i$
  end if

on receive($-(r), P_i$):
  remove $r$ from $R'_i$ if present
  add $r$ to $D_i$

Termination:
  Detect that all processes are idle
  return $\cup_{P_i} R'_i$
```

Algorithm 1: The D-Swoosh distributed ER algorithm

Table 1 Full Replication Scheme

Scope of r_i:
  Cardinality: p
  for $l := 0$ to $p - 1$, processor P_l is in scope(r_i).
Records r_i, r_j at:
  Number of overlap processors: $m = p$;
  for $l := 0$ to $m - 1$, processor P_l in scope of both r_i and r_j.
Resp(P_k, r_i, r_j) = true if $k = \min(i, j) \bmod p$.

Table 2 Majority Scheme

```
Notation:
    Number of buckets: $B$;
    Total number of processors: $p = B$;
    Record $r_i$ hashes to bucket $x$; record $r_j$ to bucket $y$;
    Assume $x \geq y$.
Scope of $r_i$:
    Cardinality: $\lfloor p/2 \rfloor + 1$;
    for $k := x$ to $x + \lfloor p/2 \rfloor$ (modulo $p$),
        processor $P_k$ is in $scope(r_i)$.
Records $r_i$, $r_j$ at:
    Case (a): $x - y < p/2$;
        Distance: $d = (x - y) \bmod p$;
        Number of overlap processors: $m = (\lfloor p/2 \rfloor + 1) - d$;
        for $k := x$ to $x + m - 1$ (mod p),
            processor $P_k$ is in scope of both $r_i$ and $r_j$;
        Resp($P_k, r_i, r_j$) = true if $k = x + (\text{min}(i, j)) \bmod m$).
    Case (b): $x - y > p/2$;
        Reverse $x$ and $y$ in equations for case (a).
    Case (c): $x - y = p/2$;
        Number of overlap processors: $m = 2$;
        Processors $P_x$ and $P_y$ are in scope of both $r_i$ and $r_j$;
        Resp($P_k, r_i, r_j$) = true if
            (min($i, j$) is even and $k = x$) OR
            (min($i, j$) is odd and $k = y$).
```

Table 3 Grid Scheme

```
Notation:
    Number of buckets: $B$;
    Total number of processors: $p = B(B + 1)/2$;
    If $i \geq j$, processor $P_{i,j}$ is processor $P_k$,
        where $k = i + jp - j(j + 1)/2$;
    If $i < j$, processor $P_{i,j}$ is processor $P_{j,i}$;
    Record $r_i$ hashes to bucket $x$; record $r_j$ to bucket $y$.
Scope of $r_i$:
    Cardinality: $B$;
    for $k := 0$ to $B - 1$, processor $P_{x,k}$ is in scope($r_i$).
Records $r_i$, $r_j$ at:
    Number of overlap processors:
        If $x \neq y$ then $m = 1$ else $m = B$;
    If $x \neq y$ then processor $P_{x,y}$ is in scope of both $r_i$ and $r_j$;
    If $x = y$ then scope($r_i$)= scope($r_j$).
Resp($P_k, r_i, r_j$) = true if $P_k = P_{x,y}$.
```

Table 4 Groups Scheme

```
Notation:
    Number of groups: $G$;
    Total number of processors: $p = G$;
    Processor $P_k$ gets records in group $g_k$;
    Record $r_i$ is in set of groups $X$; record $r_j$ is in set $Y$.
Scope of $r_i$:
    Cardinality: $|X|$;
    for all $g_k \in X$, processor $P_k$ is in scope($r_i$).
Records $r_i$, $r_j$ at:
    Number of overlap processors: $m = |X \cap Y|$;
    for $g_k \in X \cap Y$ , processor $P_k$ is in scope of both $r_i$ and $r_j$;
Resp($P_k, r_i, r_j$) = true if $g_k = l^{th}$ group in $X \cap Y$,
        where $l = (\text{min}(i, j)) \bmod m$).
```