A Scalable Parallel Deduplication Algorithm

Walter Santos* Thiago Teixeira* Carla Machado† Wagner Meira Jr.*
Altigran S. Da Silva‡ Renato Ferreira* Dorgival Guedes*

*Department of Computer Science
†Department of Demography
Universidade Federal de Minas Gerais, Brazil

‡Department of Computer Science
Universidade Federal do Amazonas, Brazil

Abstract

The identification of replicas in a database is fundamental to improve the quality of the information. Deduplication is the task of identifying replicas in a database that refer to the same real world entity. This process is not always trivial, because data may be corrupted during their gathering, storing or even manipulation. Problems such as misspelled names, data truncation, data input in a wrong format, lack of conventions (like how to abbreviate a name), missing data or even fraud may lead to the insertion of replicas in a database.

The deduplication process may be very hard, if not impossible, to be performed manually, since actual databases may have hundreds of millions of records. In this paper, we present our parallel deduplication algorithm, called FER-APARDA. By using probabilistic record linkage, we were able to successfully detect replicas in synthetic datasets with more than 1 million records in about 7 minutes using a 20-computer cluster, achieving an almost linear speedup. We believe that our results do not have similar in the literature when it comes to the size of the data set and the processing time.

1 Introduction

Reliable and consistent databases are key for tasks such as decision support, trend analysis, fraud detection, and business intelligence. However, actual databases present problems such as missing, invalid, or even duplicate data. In this paper we tackle this last problem, that is, how to detect duplicate entries in databases. Application scenarios include eliminating replicas in digital libraries, health records, customers and financial databases, among others. In all cases, we may want to find all occurences of a given entity, despite the noisy data and the absence of an effective identifier. This problem is known as *Entity Resolution (ER)* and the process of eliminating replicas is called *deduplication*. As we discuss, this problem becomes harder as we increase the size of the databases or the number of sources (usually heterogeneous) used to create the database [12].

There are several strategies for performing the deduplication of a database. A simple approach is the *deterministic (exact) linkage*, where we assume that a set of record attributes may be used as an identifier. In practice, a set of rules is used to identify replicas [5]. Although usually efficient and possible to implement in database management systems, this strategy does not deal well with a variable amount of noise, what is usually the case in actual databases. One strategy that addresses this last issue is the *probabilistic linkage* [7], which is based on attribute-based probabilities of true and false matches to determine the probability that two records refer to the same entity [6]. We may divide this strategy into three major tasks. The first task is to determine which records must be compared in order to determine whether they are duplicate or not. The second task is to perform the comparison in a per-attribute basis, which should take into consideration their different type (e.g., number or string) and other characteristics. The third task is to check the per-attribute comparison results and decide whether there is a duplicate. It is important to note that the worst case scenario of the problem has a quadratic cost, when we compare all pairs of records in the database.

Some challenges must be addressed to achieve scalability while performing probabilistic linkage. The first one is the computational cost, associated with both the determination of the pairs and their comparison. The second challenge is the often large storage demand associated with these large databases, which do not fit in a single machine. The third challenge is that the application is completely irregular, so that the amount of computation and storage varies with the nature of the input. In this paper we present a scalable parallelization of the probabilistic linkage algo-

1550-6533/07 $25.00 © 2007 IEEE
DOI 10.1109/SBAC-PAD.2007.32

rithm, which addresses all three challenges. The computational and storage-related challenge is addressed by a transparent partition of the data and the tasks to be performed. The third challenge is addressed by maximizing the overlap between computation and communication, as well as employing replication and message coalescing techniques.

To our best knowledge, only three solutions, Febrl [13], a family of algorithms (p-DC, p-DD1 and P-DD2) [10] and P-Swoosh [11] have common objectives with this work. But none of these solutions provide better scalability and efficiency than FERAPARDA, in terms of number of records and comparison rate.

2 Problem Formulation

Consider a dataset $R = \{r_1, r_2, r_3, \ldots, r_n\}$ formed by records with attributes $A = \{R.A_1, R.A_2, \ldots, R.A_m\}$. Let $\epsilon = \{e_1, e_2, \ldots, e_j\}$ be the set formed by the distinct real world entities from dataset R. Consider also a function $E(r_i)$ that maps a record $r_i \in R$ to an entity $e_j \in \epsilon$. We say that a record a is a replica of a record b if $E(r_a) = E(r_b)$. In general, the mapping $E(r_i)$ is only provided by the dataset when there is a reliable record identifier formed by a subset of A. In these cases, a simple *join* operation is enough to eliminate replicas. In many cases, however, there is no unique identifier or, further, the size of ϵ is not even known [2]. In these cases, it is necessary to develop more elaborate techniques. The deduplication process tries to identify the ϵ set with as few misses as possible.

2.1. The Deduplication Process

Real world data is generally dirty: it contains incomplete, misspelled and noisy information [9]. When deduplicating a dataset, the first step is to clean and to standardize it. The cleaning and standardization task converts raw input data into a consistent and well formed form. Although such tasks are part of the deduplication process, we will leave them as future work. The deduplication process is shown in Figure 1.

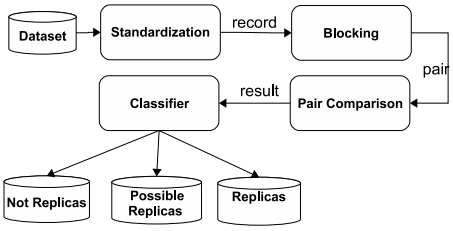

Figure 1. The deduplication process

The next task is known as *blocking*. The objective of this task is to limit the number of comparisons. $O(n^2)$ comparisons are necessary in the general case, but, in practice, most of them can be eliminated. Many works [1, 3] discuss different blocking strategies. In this work, we consider only the *standard* or *classic* blocking, which defines a *blocking predicate* of a disjunction of *conjunctions*. Each part of a conjunction defines a transformation function over one or more attributes of the record. An example of predicate is $P = (first\_name \wedge year\_of\_birth) \cup (last\_name \wedge country)$. When applied to a record, the blocking predicate will generate a *blocking key* for each conjunction. Comparisons are only executed for items with the same blocking key.

There is a trade off in using this blocking mechanism. A strict blocking predicate that generates several small blocks can leave a pair (r_a, r_b) out of the comparison process because the records did not generate any matching blocking key. On the other hand, generating few large blocks, the number of record comparisons grows quadratically, implying similar grow in execution time.

The pair comparison stage takes the pairs generated by the cartesian product within each block and compares the records attributes. There are, again, several alternatives in the literature on how to perform the record comparison, which can be based on the attributes [7], on string and encoding functions, and, recently, using TF-IDF schemes [4]. The comparison functions can be simple, as exact string or number comparison or take into account several typographical errors. Each comparison function returns a numerical value, that can be normalized between zero and one, where zero indicates disagreement, and a value greater than a minimum is considered agreement (to tolerate small errors).

The last stage, *classification*, involves using some similarity function which summarizes the results of the pairwise comparison and classifies the pairs in *matches*, *non-matches* and *possible matches*. The similarity function is *probabilistic* and decides if a pair is a match if the result of the similarity function is greater than or equal to an upper limit T_{upper}. The pair is not a match if the result is less than a lower limit T_{lower}. If the result of the similarity function is between the two limits, the pair is considered a possible match.

2.2 The Sequential Algorithm

The sequential algorithm is straight forward and its pseudo-code is shown in Algorithm 1. Each record is read from the database, and, for each conjunction in blocking predicate, it generates a blocking key (lines 6-7). If there is no existing block associated to the blocking key, a new block is created (lines 9-11) and a reference to the record is stored in it. Otherwise, the block associated with the generated key is retrieved, the record is compared to all the records already stored in the block, and a reference to the current record is appended to the block.

Only records having the same blocking key for the same

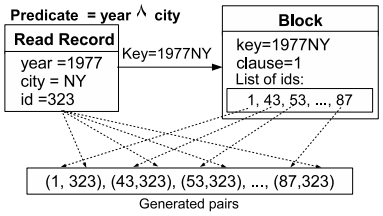

Figure 2. Pair generation

conjunction will be compared. Figure 2 illustrates the blocking stage. In the example, the blocking predicate, defined as the concatenation of the record year and city, generates a key with value *1977NY*. The identifiers of the records already processed are listed within the block (*1, 43, 53, ... and 87*). When record *323* is read from the input, a blocking key with value *1977NY* is generated and the block for the key is retrieved from the hash table. The algorithm then creates pairs with record 323 and all the records stored within the block and executes the comparison and the classification (lines 13-16). That record, whose identifier is 323, is then appended into the block (line 17).

```
1  Deduplicate (Dataset, Configuration):
2  HashTable ← ∅
3  foreach Record ∈ Dataset do
4      foreach Conjunction ∈ Predicate do
5          Key ← ∅
6          foreach Transformation ∈ Conjunction do
7              Key ← concatenate(Key,
               transform(Record))
8          /*Creating a new block*/
9          if Block ← get(HashTable, Key) == NULL then
10             Block ← createBlock()
11             put(HashTable, Key, Block)
12         /*Comparing*/
13         foreach OldRec ∈ Block do
14             Pair ← (Record, OldRec)
15             Result ← compare(Configuration, Pair)
16             isPair ← classify(Result)
17         Block.Append(Record)
```

Algorithm 1: Sequential deduplication algorithm

3 The Parallel Algorithm

In this section we present our parallel version of the deduplication algorithm. We start by presenting the Anthill environment, which is used as a basis for the parallelization implemented. After, our proposed algorithm is discussed. We discuss the parallelization dimensions exploited and advantages, as well as implemented optimizations.

```
1  Reader (Dataset, Configuration, Rank, Instances):
2  sequence ← 0
3  foreach Records ∈ Dataset do
4      Key ← ∅
5      Record.id ← sequence ∗ Instances + rank
6      foreach Conjunction ∈ Predicate do
7          foreach Transformation ∈ Conjunction do
8              Key ← concatenate(Key,
               transform(Record))
9          sendToblocker(Record.id, Key, Conjunction)
10     sequence ← sequence + 1
```

Algorithm 2: Reader filter algorithm

```
1  Blocking ():
2  HashTable ← ∅
3  foreach Message from Reader do
4      Key ← Message.key
5      Conjunction ← Message.conjunction
6      if Block ← get(HashTable, Key, Conjunction) ==
       NULL then
7          Block ← createBlock()
8          put(HashTable, Key, Conjunction, Message.id)
9      foreach OldRec ∈ Block do
10         Pair ← sort(OldRec, Message.id)
           sendTomerger(Pair)
```

Algorithm 3: Blocking filter algorithm

3.1 Anthill

Building applications that may efficiently exploit parallelism while maintaining good performance is a challenge. In this scenario, given their size, datasets are usually distributed across several machines in the system to improve access bandwidth. as Success in this approach depends on the application being divided into portions that may be instantiated on different nodes of the system for execution. Each of these portions performs part of the transformation on the data starting from the input dataset and proceeding until the resulting dataset is produced, in what is called the *filter-stream model*.

In the filter-stream model, filters are the representation of each stage of the computation, where data is transformed, and streams are abstractions for communication which allow fixed-size untyped data buffers to be transferred from one filter to the next. Creating an application is a process referred to as filter decomposition. In this process, the application is modeled as a dataflow computation and then broken into a network of filters, creating task parallelism as in a pipeline. At execution time, multiple (transparent) copies of each of the filters that compose the application are instantiated on several machines of the system and the streams are connected from sources to destinations.

```
1  Merger ():
2  HashTable ← ∅
3  foreach Message from Blocker do
4      Pair ← Message.pair
5      Key ← f(Pair)
6      if get(HashTable, Key) == NULL then
7          sendTocomparator(Message.pair)
8          put(HashTable, Key)
```

Algorithm 4: Merger filter algorithm

Our run-time system, Anthill [8], tries to exploit the maximum parallelism in applications structured in the filter-stream model by exploiting three potential sources of parallelism: task parallelism, data parallelism, and asynchrony. By dividing the computation into multiple pipeline stages (task parallelism), each one replicated multiple times (to handle data in parallel), we can have a very fine-grained parallelism and, since all this is happening asynchronously, the execution is mostly bottleneck free. In order to reduce latency, the grain of the parallelism should be defined by the application designer at run-time.

In our experience, we observed that when we decomposed our data intensive applications into filters, the natural solution was often a cyclic graph, where the execution consisted of multiple iterations over the filters. An application would start with data representing an initial set of possible solutions and as those passed down the filters new candidate solutions would be created. Those, in turn, would have to be passed through the network to be processed themselves. Also, we noticed that this behavior led to asynchronous executions, in the sense that several solutions (possibly from different iterations) might be tested simultaneously at run-time.

One important characteristic distinguishes the Anthill environment from its predecessors: given the cyclic nature of the computation, there are often dependencies among different data that flow through the cycle. Since each stage of computation may have multiple replicas, we must have a way to state that the result of a computation in a given cycle be fed back to a specific instance of another stage. That may be necessary because, for example, there may be some state associated with all inter-dependent pieces of data and that can reside in only one filter instance, so all dependent data must be routed through that specific instance. The abstraction for that in the Anthill is called a labeled stream. It is created by allowing the programmer to associate a label with each message and a mapping function (hash) that associates each possible label with a specific filter instance.

This mechanism gives the application total control over the routing of its messages. Because the hash function is called at runtime, the actual routing decision is taken individually for each message and can change dynamically as

```
1  Comparator (DatasetPartition):
2  SentCache ← ∅
3  RecCache ← ∅
4  foreach Message received do
5      id1 ← Message.pair.id1
6      id2 ← Message.pair.id2
7      /*It Always has this record*/
8      Record1 ← get(DatasetPartition, id1)
9      if Message.type = Compare then
10         /*Instance has both records?*/
11         if id2 ∈ DatasetPartition then
12             Record2 ← get(DatasetPartition, id2)
13             R ← compare(Record1, Record2)
14             sendToclassifier(R)
15         else if id2 ∈ RecCache then
16             R ← compare(id1, id2)
17             sendToclassifier(R)
18         else if id2 ∈ SentCache then
19             sendToprocess(owner(id2), id1, id2)
20         else
21             sendRecord(owner(id2), record(id1), id2)
22             put(SentCache, id1, owner(id2))
23     else if Message.type = CRRmsg then
24         Record2 ← get(RecCache, id2)
25         R ← compare(Record1, Record2)
26         sendToclassifier(R)
27     else if Message.type = RCRmsg then
28         Record2 ← Message.Record
29         R ← compare(Record1, Record2)
30         sendToclassifier(R)
31         put(RecCache, id2)
```

Algorithm 5: Comparator filter algorithm

the execution progresses. This feature conveniently allows dynamic reconfiguration, which is particularly useful to balance the load on dynamic and irregular applications. The hash function may also be slightly relaxed, in the sense that its output does not have to be one single instance. Instead, it can output a list of those. In that case, a message may be replicated (multicast) or even broadcast. This is particularly useful for applications in which one single input data element influences several output data elements.

3.2 Parallelization Strategy

In this section we describe our parallel implementation of the deduplication algorithm. The parallelization is based on four filters: ReaderComparator, Blocking, Classifier, and Merger.

The **ReaderComparator** filter (Algorithms 2 and 5) is responsible for reading each record of the dataset, assigning an identifier to the record and generating a *blocking*

```
1  Classifier ():
2  foreach Message from Comparator do
3      C ← Message.c
4      C' ← f(C)
5      if C' > upperthreshold then
6          Pair is a match.
7      else if C' < lowerthreshold then
8          Pair is not a match.
9      else
10         Can not say anything.
```

Algorithm 6: Classifier filter algorithm

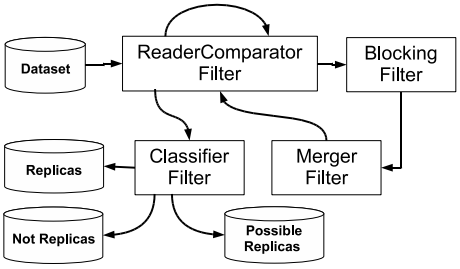

Figure 3. Filter pipeline

key for each conjunction in the blocking predicate. The record identifier is generated in a such way that it is possible to identify which filter instance read it (by using $id = totalOfInstances * sequence + rank$), what is necessary for later stages of the algorithm.

Once a blocking key is generated, it is sent together with the record identifier and the conjunction identifier to the **Blocking** filter (Algorithm 2, line 9). This communication employs a *labeled stream* based on the blocking key, so that every message with the same blocking key will be sent to the proper instance of the Blocking filter. The Blocking filter (Algorithm 3) keeps a list of record identifiers for all records that generate the same blocking key. When a new message arrives, the Blocking filter identifies whether it has to create a new block or simply append to an existing one. Further, during message reception, the Blocking filter generates a pair of record identifiers formed by the identifier from the message and each identifier present in the list, as can be seen in Figure 2.

Predicate conjunctions may overlap and generate redundant pairs, which would result in redundant processing. To avoid this, we employ a **Merger** filter (Algorithm 4), which eliminates redundant pairs. To identify redundant pairs, the Merger filter keeps a hashtable defined by the combination of the identifiers, where the smaller one is always used first. An interesting optimization in this filter arises from the fact that the record identifier generation is always growing, thus, after a while, no pair will be associated with that identifier anymore. In early experiments, when we were keeping all pairs in this filter, memory utilization was high. Now, we defined a circular list that keeps the hash keys and limits the hashtable size, while it guarantees access to all relevant keys.

All pairs that have never been processed are sent back to the **ReaderComparator** filter. The Merger filter uses labeled streams based on the larger record identifier. We should recall that, through an identifier, it is possible to know which ReaderComparator instance generated it. We always use the larger identifier because we assume that the ReaderComparator filter that generated it will be able to compare the pair. Note that the Merger filter will always send a pair to a ReaderComparator instance that has at least one of the records.

Figure 3 shows this loop. We decided to implement Reader and Comparator as the same filter because during the comparison stage it already has all needed records in its memory. It is important to say that, in this work, we are assuming that the database partition is stored in memory, and future work will address the cases where such premise is not valid.

It is important to notice that we could not find an efficient strategy that produces a perfect partition of an arbitrary ordered dataset that guarantees a balanced load among ReaderComparator filters. Thus, a pair of records to be compared may be in any of the instances and we do not know in advance whether it is possible to exploit any locality of reference. When a ReaderComparator receives a message of type *Compare* from the Merger filter, it checks whether it has both records in its dataset partition. If this is the case, it compares them and send a message with the comparison result to the Classifier filter (lines 11-14, Algorithm 5). When a ReaderComparator instance has only one of the records, it has to send its record to the instance that owns the other one, which, of course, involves communication. Having this in mind, we made several optimizations to reduce communication. We defined two new types of messages: *ReceiveAndCompareRecord (RCRmsg)* and *CompareAlreadyReceivedRecord (CRRmsg)*. *ReceiveAndCompareRecord* message carries the entire record and does not reduce costs by itself (lines 20-22). However, after a record has been sent to another ReaderComparator instance, it will be never sent again to that instance. Whenever a new pair arrives at a ReaderComparator instance and it does not have both records, it checks whether it has already sent its record to the other instance. If this is the case, it sends just a *CompareAlreadyReceivedRecord* message (lines 18-19). Finally, there is another optimization: if a ReaderComparator instance receives a pair, it holds just one of the records, and the other record is in its receive cache, no communication is needed and the result of the comparison is sent to the classifier (lines 15-17).

The last filter is the **Classifier** (Algorithm 6). It will classify the pair of records according to the result of the ReaderComparator filter and says whether the pair is a match, is not a match, or is a possible match (needs human supervision).

3.3 Discussion

Our parallel implementation of the deduplication algorithm exploit the task parallelism, data parallelism, and asynchrony, through its run-time system, Anthill.

Task parallelism is implemented through the pipeline. The pipeline is controlled by the data dependencies. The ReaderComparator does not need to read the entire database before sending a record to Blocking filter. In this case, as soon as a record is read, it is put in the pipeline for processing. As a result, multiple tasks may be performed simultaneously.

By using multiple instances of a given filter that processes a partition of the dataset, we are taking advantage of the data parallelism. The ReaderComparator filter is the best candidate to have multiple instances. Once it has both records, the processing does not depend on other data. Other good candidate to have multiple instances is the Blocking filter. The Blocking filter iterates over a list of record identifiers that had the same blocking key in $O(n)$, where n is the size of the list. When using a larger number of instances of ReaderComparator, the number of messages arriving to the Blocking filter may overload it.

There are two asynchrony opportunities exploited by the algorithm. The first one is when a comparison pair is sent to the ReaderComparator as soon as the MergerFilter learns about it. In this case, if the same pair comes later from a different block, it is not sent again. The second opportunity is when more than one instance of the same filter may run on the same processor, in order to exploit the asynchrony. While one instance is sending data through the network, other instance may be using the CPU. In our experiments, we could observe that using multiprogramming for the filters helped in optimize specially the CPU use.

4 Experimental Results

In this section we present an experimental evaluation of our parallel deduplication algorithm using synthetic datasets. The experiments were executed in a cluster of AMD Athlon 64 3200+ processors with 2GB RAM memory, connected by Gigabit Ethernet, and running Linux 2.6.

Given the difficulty of getting large and real datasets, we ran tests using a synthetic database generated by Febrl data generator (DsGen). It is, therefor, possible to reproduce the same characteristics in the data as can be encountered in the real world. DsGen creates datasets with records that contain randomly created names and addresses (using frequency files), dates, phone numbers, and identifier numbers. Duplicate records will then be created following a given probability distribution, with different single errors being introduced [13].

In order to evaluate the scalability, we defined a predicate P that would generate a large number of pairs:

$P = (phone\_number \wedge first\_name) \cup$
$(phone\_number \wedge year\_month\_birthdate) \cup$
$(dmetaphone(last\_name, 4) \wedge year\_birthday) \cup$
$(dmetaphone(first\_name, 4) \wedge birthdate) \cup$
$(zip\_code \wedge birthdate) \cup$
$(truncate(first\_name, 3) \wedge zip\_code) \cup$
$(nysiis(locality, 4) \wedge month\_birthdate)$.

In our first experiment we want to measure the scalability of our implementation. Figure 4 shows the results of our experiments varying the number of records in each dataset and the number of instances of *ReaderComparator* filter, considering one instance per processor. The executions varied from 1 to 15 instances and we used dataset containing 1 million, 500k, and 250k records.

After using `gprof` profiler to evaluate our sequential implementation, we noticed that about 90% of execution time was spent in the comparisons. For this reason, we based the speedup experiment only on the number of instances of ReaderComparator filter.

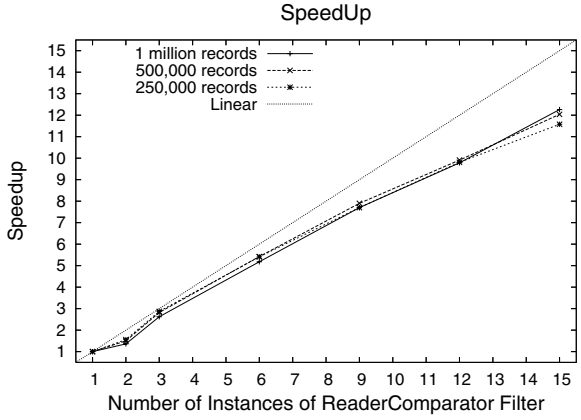

Figure 4. Feraparda Speedup

Analyzing the speedup graph, we can see that the scalability of the algorithm is almost the same for all the dataset sizes, achieving an efficiency of above 80% up to 15 processors. A significant part of this result is due to Anthill capabilities described in Subsection 3.3.

On the other hand, as the number of instances increases, the speedup tends to decrease, and we believe that the communication costs are the main cause. When using a large number of processors there is a significant communication

cost and a low computational demand, reducing the efficiency of the algorithm. This increase in communication was expected because the more nodes are used, the smaller are the odds of having both records to be compared on the same node.

Figure 5 shows this increase in number of messages exchanged when the number of instances of ReaderComparator filter increases. We use the 1 million record dataset for this experiment. Analyzing the graph, we see that the total number of messages exchanged in the 15-instance execution is almost twice the number of messages exchanged in the 1 instance execution. The graph shows that a total of almost 2.2 million messages were exchanged independently of the configuration. For 15 instances were additionally exchanged more than 1.5 million messages only between the instances of ReaderComparator filter.

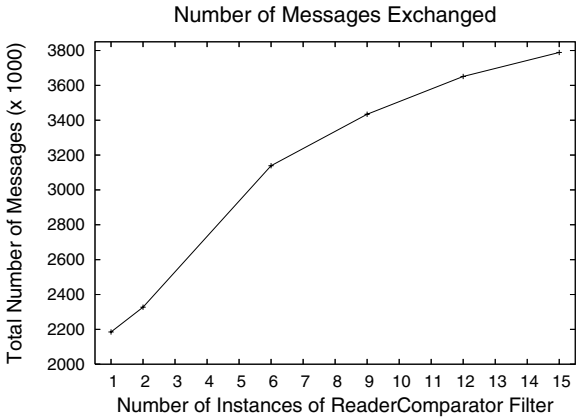

Figure 5. Messages exchanged in Feraparda

The use of a cache, as we proposed, significantly reduces the communication between instances of ReaderComparator filter. We only send records to instance once, and the next time we only send the meta-data of this record to reduce communication. Besides, we group meta-data from various records before sending, in an effort to further reduce communication overhead.

Figure 6 shows the absolute number of comparisons per second and the percentage of comparisons performed using each kind of message. In this experiment we used 3 instances of ReaderComparator filter. We see that more than 60% of comparisons, or more than 30,000 comparisons per second, were performed using messages with the aforementioned meta-data (*CompareAlreadyReceivedRecord* message type). Also, approximately 40% of the comparisons, or almost 15,000 comparisons per second, were performed using records in local datasets. It also shows that the algorithm has an initial warm-up during which there is a high ratio of records exchanged, but after some time it drops to almost 0. In certain way, label streams provide locality of reference because once a record is send it will not be sent again.

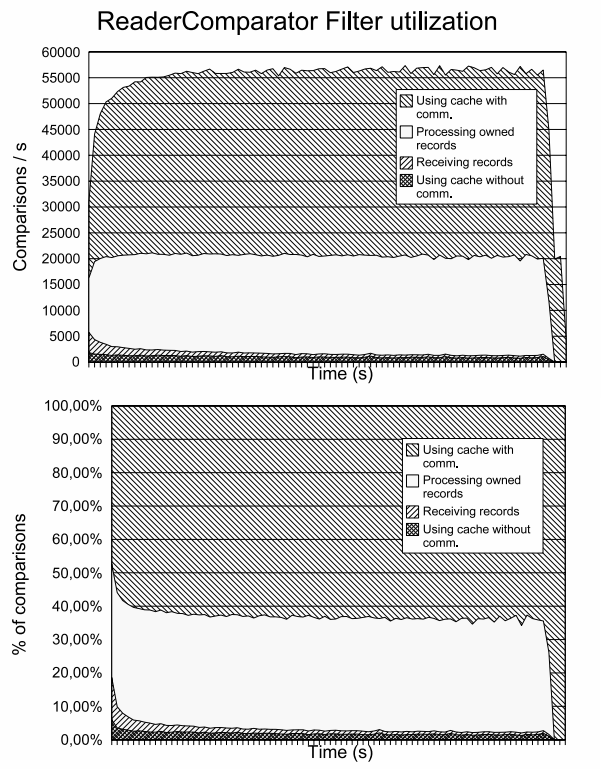

Figure 6. Comparisons performed by ReaderComparator filter with 3 instances

To evaluate the effectiveness of our cache, we collected statistics of its utilization. These statistics are presented in Table 1. We see that the number of comparisons performed using cache increases with the number of ReaderComparator instances. The more instances there are, the smaller is the local partition, the more communication is needed to perform comparisons, and the more cache is then used. We also notice that the number of comparisons varied for each configuration. This occurred because the merger filter has a cache with limited size of 10,000 and two identical pairs could be compared twice because when the second arrive, the first one may have already left the cache. A study about optimizing the cache is left as future work.

Another reason for achieving the observed speedup was the load balance between ReaderComparator instances. The number of comparisons done by each instance should be the same to achieve a perfect load balance. The graph from Figure 7 shows the percentage of the total comparisons performed by the worst instance (performed smaller number of comparisons), best instance (performed the larger number of comparisons), and the average between all instances. The

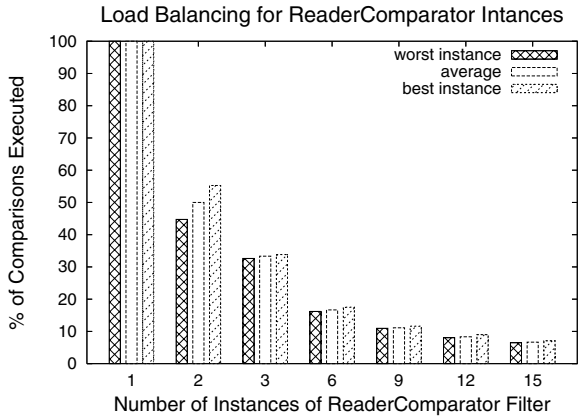

Figure 7. Load balance analysis

# Inst.	Total of Comparisons	Using Cache	%
1	270633791	0	0
2	270633788	134454100	49.68
3	270725451	178538727	65.95
6	271163034	221754408	81.78
9	271382940	235108278	86.63
12	271430184	240764988	88.70
15	271585380	243796260	89.77

Table 1. Statistics of comparisons in ReaderComparator filter (1 mi)

difference between the worst and best instances becomes insignificantly when the number of instances of ReaderComparator grows, presenting a very good load balancing.

5 Conclusions and Future Work

In this work we proposed and evaluated a parallel and scalable algorithm for the deduplication problem. The algorithm is implemented using the Anthill programming environment and exploits both task and data parallelism, multi-programming and message coalescing techniques to achieve scalability. The experiments show impressive numbers when compared to other solutions, achieving an almost linear speedup. As future work directions, we plan to evaluate other blocking techniques, their benefits and their parallelization. A more detailed study about the data distribution, use of caches and database replication is also planned. Finally, we will apply our solution to a real health record database (in progress) and evaluate the results.

References

[1] R. Baxter, P. Christen, and T. Churches. A comparison of fast blocking methods for record linkage. In *Proceedings of 9th ACM SIGKDD Workshop on Data Cleaning, Record Linkage and Object Consolidation*, 2003.

[2] I. Bhattacharya, L. Licamele, and L. Getoor. Relational clustering for entity resolution queries. In *ICML 2006 Workshop on Statistical Relational Learning (SRL)*, Pittsburgh, PA, USA, 2006.

[3] M. Bilenko, B. Kamath, and R. J. Mooney. Adaptive blocking: Learning to scale up record linkage. In *ICDM '06: Proceedings of the Sixth International Conference on Data Mining*, pages 87–96, Washington, DC, USA, 2006. IEEE Computer Society.

[4] M. Bilenko and R. J. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 39–48, New York, NY, USA, 2003. ACM Press.

[5] P. Christen and K. Goiser. *Quality and Complexity Measures for Data Linkage and Deduplication*, volume 43. Springer Berlin / Heidelberg, Secaucus, NJ, USA, 2007.

[6] M. G. Elfeky, A. K. Elmagarmid, and V. S. Verykios. Tailor: A record linkage tool box. In *ICDE '02: Proceedings of the 18th International Conference on Data Engineering*, pages 17–28, Washington, DC, USA, 2002. IEEE Computer Society.

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

[8] R. A. Ferreira, J. Wagner Meira, D. Guedes, L. M. A. Drummond, B. Coutinho, G. Teodoro, T. Tavares, R. Araujo, and G. T. Ferreira. Anthill: A scalable run-time environment for data mining applications. In *SBAC-PAD '05: Proceedings of the 17th International Symposium on Computer Architecture on High Performance Computing*, pages 159–167, Washington, DC, USA, 2005. IEEE Computer Society.

[9] M. A. Hernandez and S. J. Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Min. Knowl. Discov.*, 2(1):9–37, 1998.

[10] D. L. Hung-sik Kim and. Parallel Linkage. Technical report, The Pennsylvania State University, 2007.

[11] H. Kawai, H. Garcia-Molina, O. Benjelloun, D. Menestrina, E. Whang, and H. Gong. P-swoosh: Parallel algorithm for generic entity resolution. Technical Report, Stanford InfoLab, 2006.

[12] B.-W. On, E. Elmacioglu, D. Lee, J. Kang, and J. Pei. Improving grouped-entity resolution using quasi-cliques. In *ICDM '06: Proceedings of the Sixth International Conference on Data Mining*, pages 1008–1015, Washington, DC, USA, 2006. IEEE Computer Society.

[13] M. H. Peter Christen, Tim Churches. Febrl: A parallel open source data linkage system. In *Springer Lectures Notes in Artificial Intelligence, Proceedings of the 8th Pacific-Asia Conference, PAKDD 2004*, volume 3056, pages 638–647, Sidney, Australia, 2004. Springer.