# Real-world Data is Dirty: Data Cleansing and The Merge/Purge Problem [*]

Mauricio A. Hernández[†]
mauricio@cs.columbia.edu

Salvatore J. Stolfo
sal@cs.columbia.edu

Department of Computer Science
Columbia University
New York, NY 10027

**Abstract**

The problem of merging multiple databases of information about common entities is frequently encountered in KDD and decision support applications in large commercial and government organizations. The problem we study is often called the Merge/Purge problem and is difficult to solve both in scale and accuracy. Large repositories of data typically have numerous duplicate information entries about the same entities that are difficult to cull together without an intelligent "equational theory" that identifies equivalent items by a complex, domain-dependent matching process. We have developed a system for accomplishing this Data Cleansing task and demonstrate its use for cleansing lists of names of potential customers in a direct marketing-type application. Our results for statistically generated data are shown to be accurate and effective when processing the data multiple times using different keys for sorting on each successive pass. Combing results of individual passes using transitive closure over the independent results, produces far more accurate results at lower cost. The system provides a rule programming module that is easy to program and quite good at finding duplicates especially in an environment with massive amounts of data. This paper details improvements in our system, and reports on the successful implementation for a "real-world" database that conclusively validates our results previously achieved for statistically generated data.

**Keywords: data cleaning, data cleansing, duplicate elimination, semantic integration**

[*]This work has been supported in part by the New York State Science and Technology Foundation through the Center for Advanced Technology in Telecommunications at Polytechnic University, by NSF under grant IRI-94-13847, and by Citicorp.

[†]This author is now at the University of Illinois at Springfied. Work at Columbia University was supported by an AT&T Cooperative Research Program Fellowship.

## 1 Introduction

Merging large databases acquired from different sources with heterogeneous representations of information has become an increasingly important and difficult problem for many organizations. Instances of this problem appearing in the literature have been called *record linkage* [12], the *semantic integration* problem [1] or the *instance identification* problem [23], and more recently the *data cleansing* problem regarded as a crucial first step in a KDD/DM process [11]. Business organizations call this problem the *merge/purge* problem.

In this paper we consider the data cleansing of very large databases of information that need to be processed as quickly, efficiently, and accurately as possible. For instance, one month is a typical business cycle in certain direct marketing operations. This means that sources of data need to be identified, acquired, conditioned, and then correlated or merged within a small portion of a month in order to prepare mailings and response analyses. It is not uncommon for large businesses to acquire scores of databases each month, with a total size of hundreds of millions to over a billion records, that need to be analyzed within a few days. In a more general setting, data mining applications depend upon a conditioned sample of data that is correlated with multiple sources of information and hence accurate database merging operations are highly desirable. Within any single data set the problem is also crucial for accurate statistical analyses. Without accurate identification of duplicated information, frequency distributions and various other aggregations will produce false or misleading statistics leading to perhaps untrustworthy new knowledge.

Large organizations have grappled with this problem for many years when dealing with lists of names and addresses and other identifying information. Credit card companies, for example, need to assess the financial risk of potential new customers who may purposely hide their true identities (and thus their history) or manufacture new ones. The Justice Department and other law enforcement agencies seek to discover crucial links in complex webs of financial transactions to uncover sophisticated money laundering activities [22]. Errors due to data entry mistakes, faulty sensor readings or more malicious activities, provide scores of erroneous datasets that propagate errors in each successive generation of data.

The problem of merging two or more databases has been tackled in a straightforward fashion by a simple sort of the concatenated data sets followed by a duplicate elimination phase over the sorted list [4]. However, when the databases involved are heterogeneous, meaning they do not share the same schema, or that the same real-world entities are represented differently in the datasets, the problem of merging becomes more difficult. The first issue, where databases have different schema, has been addressed extensively in the literature and is known as the *schema integration* problem [3]. We are primarily interested in the second problem: heterogeneous representations of data and its implication when merging or joining multiple datasets.

The fundamental problem in merge/purge is that the data supplied by various sources typically include identifiers or string data, that are either different among different datasets or simply erroneous due to a variety of reasons (including typographical or transcription errors, or purposeful fraudulent activity (aliases) in the case of names). Hence, the equality of two values over the domain of the common join attribute is not specified as a simple arithmetic predicate, but rather by a set of equational axioms that define equivalence, i.e., by an *equational theory*. Determining that two records from two databases provide information about the same entity can be highly complex. We use a rule-based knowledge base to implement an equational theory.

The problem of identifying similar instances of the same real-world entity by means of an inexact match has been studied by the Fuzzy Database[5] community. Much of the work has concentrated on the problem of executing a query $Q$ over a fuzzy relational database. The answer for $Q$ is the set of all tuples satisfying $Q$ in a non-fuzzy relational database and all tuples that satisfy $Q$ within a threshold value. Fuzzy relational databases can explicitly store possibility distributions for each value in a tuple, or use possibility-based relations to determine how strongly records belong to the fuzzy set defined by a query [14]. The problem we study in this paper is closely related to the problem studied by the fuzzy database community. However, while fuzzy querying systems are concerned with the accurate and efficient fuzzy retrieval of tuples given a query $Q$, we are concerned with the pre-processing of the entire data set before it is even ready for querying. The process we study is off-line and involves clustering *all tuples* into equivalence classes. This clustering is guided by the equational theory which can include fuzzy matching techniques.

Since we are dealing with large databases, we seek to reduce the complexity of the problem by partitioning the database into partitions or *clusters* in such a way that the potentially matching records are assigned to the same cluster. (Here we use the term cluster in line with the common terminology of statistical pattern recognition.) In this paper we discuss solutions to merge/purge in which sorting of the entire data-set is used to bring the matching records close together in a bounded neighborhood in a linear list, as well as an optimization of this basic technique that seeks to eliminate records during sorting with exact duplicate keys. Elsewhere we have treated the case of clustering in which sorting is replaced by a single-scan process[16]. This clustering resembles the hierarchical clustering strategy proposed in [6] to efficiently perform queries over large fuzzy relational databases. However, we demonstrate that, as one may expect, none of these basic approaches alone can guarantee high accuracy. Here, accuracy means how many of the actual duplicates appearing in the data have been matched and merged correctly.

This paper is organized as follows. In section 2 we detail a system we have implemented that performs a generic Merge/Purge process that includes a declarative rule language for specifying an equational theory making it easier to experiment and modify the criteria for equivalence. (This is a very important desideratum of commercial organizations that work under strict time constraints and thus have precious little time to experiment with alternative matching criteria.) Then in section 3 we demonstrate that no single pass over the data using one particular scheme as a sorting key performs as well as computing the transitive closure over several independent runs each using a different sorting key for ordering data. The moral is simply that several distinct "cheap" passes over the data produce more accurate results than one "expensive" pass over the data. This result was verified independently by Monge and Elkan [19] who recently studied the same problem using a domain-independent matching algorithm as an equational theory.

In section 4 we provide a detailed treatment of a real-world data set, provided by the Child Welfare Department of the State of Washington, which was used to establish the validity of these results. Our work using statistically generated databases allowed us to devise controlled studies whereby the optimal accuracy of the results were known a priori. In real world datasets, obviously one can not know the best attainable results with high precision without a time consuming and expensive human inspection and validation process. In cases where the datasets are huge, this may not be feasible. Therefore, the results reported here are due to the human inspection of a small but substantial sample of data relative to the entire data set. The results on the real-world data validate our previous predictions as being quite accurate. (One may view the formal results of this comparative evaluation by browsing the site http://www.cs.columbia.edu/~sal.)

Finally, in section 5, we present initial results on an Incremental Merge/Purge algorithm. The basic Merge/Purge procedure presented in section 2 assumes a single data set. If a new data set arrives, it must be concatenated to the previously processed data set and the basic Merge/Purge procedure executed over this entire data set. The Incremental algorithm removes this restriction by using information gathered from previous Merge/Purge executions. Several strategies for determining what information to gather at the end of each execution of the incremental algorithm are proposed. We present initial experimental results showing that the incremental algorithm reduces the time needed to execute a Merge/Purge procedure when compared with the basic algorithm.

## 2 Basic Data Cleansing Solutions

In our previous work we introduced the basic "sorted-neighborhood method" for solving merge/purge as well as a variant "duplicate elimination" method. Here we describe in detail this basic approach, followed by a description of an incremental variant that merges a new (smaller) increment of data with an existing previously cleansed dataset.

### 2.1 The Basic Sorted-Neighborhood Method

Given a collection of two or more databases, we first concatenate them into one sequential list of $N$ records (after conditioning the records) and then apply the sorted-neighborhood method. The sorted-neighborhood method for solving the merge/purge problem can be summarized in three phases:

1. **Create Keys** : Compute a key for each record in the list by extracting relevant fields or portions of fields. The choice of the key depends upon an "error model" that may

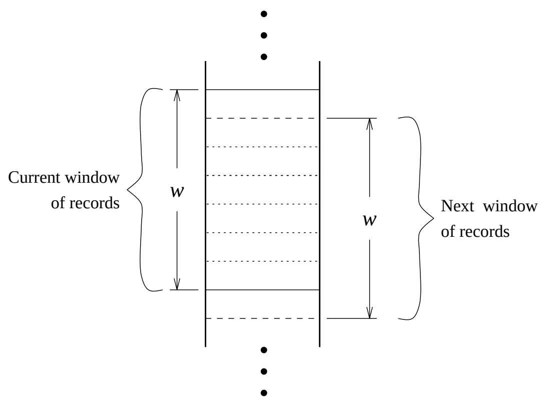


Figure 1: Window Scan during Data Cleansing

be viewed as knowledge intensive and domain-specific; the effectiveness of the sorted-neighborhood method highly depends on a properly chosen key with the intent that common but erroneous data will have closely matching keys. We discuss the effect of the choice of the key in section 2.2.

2. **Sort Data** : Sort the records in the data list using the key of step 1.

3. **Merge** : Move a fixed size window through the sequential list of records limiting the comparisons for matching records to those records in the window. If the size of the window is $w$ records, then every new record entering the window is compared with the previous $w - 1$ records to find "matching" records. The first record in the window slides out of the window (See figure 1).

When this procedure is executed serially as a main-memory based process, the create keys phase is an $O(N)$ operation, the sorting phase is $O(N \log N)$, and the merging phase is $O(wN)$, where $N$ is the number of records in the database. Thus, the total time complexity of this method is $O(N \log N)$ if $w < \lceil \log N \rceil$, $O(wN)$ otherwise. However, the constants in the equations differ greatly. It could be relatively expensive to extract relevant key values from a record during the create key phase. Sorting requires a few machine instructions to compare the keys. The merge phase requires the application of a potentially large number of rules to compare two records, and thus has the potential for the largest constant factor.

Notice that $w$ is a *parameter* of the window-scanning procedure. The legitimate values of $w$ may range from 2 (whereby only two consecutive elements are compared) to N (whereby each element is compared to all others). The latter case pertains to the full quadratic ($O(N^2)$) time process at the maximal potential accuracy (as defined by the equational theory to be the percentage of all duplicates correctly found in the merging process). The former case (where $w$ may be viewed as a small constant relative to $N$) pertains to optimal time performance (only $O(N)$ time) but at minimal accuracy. The fundamental question is *what are the optimal settings for $w$ to maximize accuracy while minimizing computational cost?*

Note, however, that for very large databases the dominant cost is likely disk I/O, and hence the number of passes over the data set. In this case, at least three passes would be needed, one pass for conditioning the data and preparing keys, at least a second pass, likely more, for a high speed sort like, for example, the AlphaSort [20], and a final pass for window processing and application of the rule program for each record entering the sliding window. Depending upon the complexity of the rule program and window size $w$, the last pass may indeed be the dominant cost. We introduced elsewhere [16] the means of speeding up this phase by processing "parallel windows" in the sorted list.

We note with interest that the sorts of optimizations detailed in the AlphaSort paper [20] may of course be fruitfully applied here. We are more concerned with alternative process architectures that lead to higher accuracies in the computed results while also reducing the time complexity. Thus, we consider alternative metrics for the purposes of merge/purge to include how *accurately* can you data cleanse for a fixed dollar and given time constraint, rather than the specific cost- and time-based metrics proposed in [20].

## 2.2 Selection of Keys

The effectiveness of the sorted-neighborhood method highly depends on the key selected to sort the records. Here a key is defined to be a sequence of a subset of attributes, or substrings within the attributes, chosen from the record. For example, consider the four records displayed in table 1. For this particular application, suppose the "key designer" for the sorting phase has determined that for a typical data set the following keys should be extracted from the data since they provide sufficient discriminating power in identifying

| First | Last | Address | ID | Key |
|---|---|---|---|---|
| Sal | Stolfo | 123 First Street | 45678987 | STLSAL123FRST456 |
| Sal | Stolfo | 123 First Street | 45678987 | STLSAL123FRST456 |
| Sal | Stolpho | 123 First Street | 45678987 | STLSAL123FRST456 |
| Sal | Stiles | 123 Forest Street | 45654321 | STLSAL123FRST456 |

Table 1: Example Records and Keys

likely candidates for matching. The key consists of the concatenation of several ordered fields (or attributes) in the data: The first three consonants of a last name are concatenated with the first three letters of the first name field, followed by the address number field, and all of the consonants of the street name. This is followed by the first three digits of the social security field. These choices are made since the key designer determined that last names are typically misspelled (due to mistakes in vocalized sounds, vowels), but first names are typically more common and less prone to being misunderstood and hence less likely to be recorded incorrectly. The keys are now used for sorting the entire dataset with the intention that all equivalent or matching data will appear close to each other in the final sorted list. Notice how the first and second records are exact duplicates, while the third is likely the same person but with a misspelled last name. We would expect that this "phonetically-based" mistake will be caught by a reasonable equational theory. However, the fourth record, although having the exact same key as the prior three records, appears unlikely to be the same person.

## 2.3 Equational theory

The comparison of records, during the merge phase, to determine their equivalence is a complex inferential process that considers much more information in the compared records than the keys used for sorting. For example, suppose two person names are spelled nearly (but not) identically, and have the exact same address. We might infer they are the same person. On the other hand, suppose two records have exactly the same social security numbers, but the names and addresses are completely different. We could either assume the records represent the same person who changed his name and moved, or the records represent different persons, and the social security number field is incorrect for one of them. Without any further information, we may perhaps assume the latter. The more information there is in the records, the better inferences can be made. For example, `Michael Smith` and `Michele Smith` could have the same address, and their names are "reasonably close". If gender and age information is available in some field of the data, we could perhaps infer that `Michael` and `Michele` are either married or siblings.

What we need to specify for these inferences is an equational theory that dictates the logic of domain equivalence, not simply value or string equivalence. Users of a general purpose data cleansing facility benefit from higher level formalisms and languages permitting ease of experimentation and modification. For these reasons, a natural approach to specifying an equational theory and making it practical would be the use of a declarative rule language. Rule languages have been effectively used in a wide range of applications requiring inference over large data sets. Much research has been conducted to provide efficient means for their compilation and evaluation, and this technology can be exploited here for purposes of data cleansing efficiently.

As an example, here is a simplified rule in English that exemplifies one axiom of our equational theory relevant to our idealized employee database:

```
Given two records, r1 and r2.
IF the last name of r1 equals the last name of r2,
       AND the first names differ slightly,
       AND the address of r1 equals the address of r2
THEN
       r1 is equivalent to r2.
```

The implementation of "`differ slightly`" specified here in English is based upon the computation of a *distance function* applied to the first name fields of two records, and the comparison of its results to a threshold to capture obvious typographical errors that may occur in the data. The selection of a distance function and a proper threshold is also a knowledge intensive activity that demands experimental evaluation. An improperly chosen threshold will lead to either an increase in the number of falsely matched records or to a decrease in the number of matching records that should be merged. A number of alternative distance functions for typographical mistakes were implemented and tested in the experiments reported below including distances based upon *edit distance*, *phonetic distance* and *"typewriter" distance*. The results displayed in section 3 are based upon edit distance computation since the outcome of the program did not vary much among the different distance functions for the particular databases used in our study.

Notice that rules do not necessarily need to compare values from the same attribute (or same domain). For instance, to detect a transposition in a person's name we could write a rule that compares the first name of one record with the last name of the second record and the last name of the first record with the first name of the second record (see appendix A for such an example rule). Modern object-relational databases allow users to add complex data types (and functions to manipulate values in the domain of the data type) to the database engine. Functions to compare these complex data types (e.g., sets, images, sound, etc.) could also be used within rules to perform the matching of complex tuples.

For the purpose of experimental study, we wrote an OPS5 [13] rule program consisting of 26 rules for this particular domain of employee records and was tested repeatedly over relatively small databases of records. Once we were satisfied with the performance of our rules, distance functions, and thresholds, we recoded the rules directly in C to obtain speed-up over the OPS5 implementation.

Appendix A shows the OPS5 version of the equational theory implemented for this work. Only those rules used encoding the knowledge of the equational theory are shown in the appendix.

The inference process encoded in the rules is divided into three stages. In the first stage, all records within a window are compared to see if they have "similar" fields, namely, the social security field, the name field, and the street address field. In the second stage, the information gathered during the first stage is combined to see if can merge pairs of records. For example, if a pair of records have similar social security numbers and similar names then the rule `similar-ssn-and-names` declares them merged. For those pair of records that could not be merged because not enough information was gathered on the first stage, the rule program takes a closer look at other fields like the city name, state and zipcode to see if a merge can be done. Otherwise, in the third stage, more precise "edit-distance" functions are

| SSN | Name (First, Initial, Last) | Address |
|---|---|---|
| 334600443 | Lisa Boardman | 144 Wars St. |
| 334600443 | Lisa Brown | 144 Ward St. |
| 525520001 | Ramon Bonilla | 38 Ward St. |
| 525250001 | Raymond Bonilla | 38 Ward St. |
| 0 | Diana D. Ambrosion | 40 Brik Church Av. |
| 0 | Diana A. Dambrosion | 40 Brick Church Av. |
| 0 | Colette Johnen | 600 113th St. apt. 5a5 |
| 0 | John Colette | 600 113th St. ap. 585 |
| 850982319 | Ivette A Keegan | 23 Florida Av. |
| 950982319 | Yvette A Kegan | 23 Florida St. |

Table 2: Example of matching records detected by our equational theory rule base.

used over some fields as a last attempt for merging a pair of records. Table 2 demonstrates a number of actual records the rule-program correctly deems equivalent.

Appendix B shows the `C` version of the equational theory. The appendix only shows the subroutine `rule_program()` which is the main code for the rule implementation in `C`. The comments in the code show where each rule of the OPS5 version is implemented.

It is important to note that the essence of the approach proposed here permits a wide range of equational theories on various data types. We chose to use string data in this study (e.g., names, addresses) for pedagogical reasons (after all everyone gets "faulty" junk mail). We could equally as well demonstrate the concepts using alternative databases of different typed objects and correspondingly different rule sets.

Table 2 displays records with such errors that may commonly be found in mailing lists, for example. (Indeed, poor implementations of the merge/purge task by commercial organizations typically lead to several pieces of the same mail being mailed at obviously greater expense to the same household, as nearly everyone has experienced.) These records are identified by our rule base as equivalent.

The process of creating a good equational theory is similar to the process of creating a good knowledge-base for an expert system. In complex problems, an expert is needed to describe the matching process. A knowledge engineer will then encode the expert's knowledge as rules. The rules will then be tested and the results discussed with the expert. Several sessions between the expert and the knowledge-engineer might be needed before the rule set is completed.

## 2.4 Computing the transitive closure over the results of independent runs

In general, no single key will be sufficient to catch all matching records. The attributes or fields that appear first in the key have higher discriminating power than those appearing after them. Hence, if the error in a record occurs in the particular field or portion of the field that is the most important part of the key, there may be little chance a record will end up close to a matching record after sorting. For instance, if an employee has two records in the database, one with social security number `193456782` and another with social security number `913456782` (the first two numbers were transposed), and if the social security number is used as the principal field of the key, then it is very unlikely both records will fall under the same window, i.e. the two records with transposed social security numbers will be far apart in the sorted list and hence they may not be merged. As we will show in the next section, the number of matching records missed by one run of the sorted-neighborhood method can be large unless the neighborhood grows very large.

To increase the number of similar records merged, two options were explored. The first is simply widening the scanning window size by increasing $w$. Clearly this increases the computational complexity, and, as discussed in the next section, does not increase dramatically the number of similar records merged in the test cases we ran (unless of course the window spans the entire database which we have presumed is infeasible under strict time and cost constraints).

The alternative strategy we implemented is to execute several independent runs of the sorted-neighborhood method, each time using a different key and a *relatively small window*. We call this strategy the *multi-pass approach*. For instance, in one run, we use the address as the principal part of the key while in another run we use the last name of the employee as the principal part of the key. Each independent run will produce a set of pairs of records which can be merged. We then apply the transitive closure to those pairs of records. The results will be a union of all pairs discovered by all independent runs, with no duplicates, plus all those pairs that can be inferred by transitivity of equality.

The reason this approach works for the test cases explored here has much to do with the nature of the errors in the data. Transposing the first two digits of the social security number leads to non-mergeable records as we noted. However, in such records, the variability or error appearing in another field of the records may indeed not be so large. Therefore, although the social security numbers in two records are grossly in error, the name fields may not be. Hence, first sorting on the name fields as the primary key will bring these two records closer together lessening the negative effects of a gross error in the social security field.

Notice that the use of a transitive closure step is not limited to the *multi-pass* approach. We can improve the accuracy of a single pass by computing the transitive closure of the results. If records `a` and `b` are found to be similar and, at the same time, records `b` and `c` are also found to be similar, the transitive closure step can mark `a` and `c` to be similar if this relation was not detected by the equational theory. Moreover, records `a` and `b` must be within $w$ records to be marked as similar by the equational theory. The same is true for records `b` and `c`. But, if the transitive closure step is used, `a` and `c` need not be within $w$ records to be detected as similar. The use of a transitive closure at the end of any single-pass run of the sorted-neighborhood method should allow us to reduce the size of the scanning window $w$ and still detect a comparable number of similar pairs as we would find without a final closure phase and a larger $w$. All single run results reported in the next section include a final closure phase.

The utility of this approach is therefore determined by the nature and occurrences of the errors appearing in the data. The choice of keys for sorting, their order, and the extraction of relevant information from a key field is a knowledge intensive activity that must be explored and carefully evaluated prior to running a data cleansing process.

In the next section we will show how the *multi-pass* approach can drastically improve the accuracy of the results of only one run of the sorted-neighborhood method with varying large windows. Of particular interest is the observation that only a small search window was needed for the *multi-pass* approach to obtain high accuracy while no individual run with a single key for sorting produced comparable accuracy results with a large window (other than window sizes approaching the size of the full database). These results were found consistently over a variety of generated databases with variable errors introduced in all fields in a systematic fashion.

## 3 Experimental Results

### 3.1 Generating the databases

All databases used to test these methods were generated automatically by a database generator that allows us to perform controlled studies and to establish the accuracy of the solution method. This database generator provides a user with a large number of parameters that they may set including, the size of the database, the percentage of duplicate records in the database, and the amount of error to be introduced in the duplicated records in any of the attribute fields. Accuracy is measured as the percentage of the number of duplicates correctly found by the process. False positives are measured as the percentage of records claimed to be equivalent but which are not actual duplicates.

Here, each generated database is viewed as the concatenation of multiple databases. The merging of records in the resultant single database is the object of study in these experiments. Each record generated consists of the following fields, some of which can be empty: social security number, first name, initial, last name, address, apartment, city, state, and zip code. The names were chosen randomly from a list of 63000 real names[1]. The cities, states, and zip codes (all from the U.S.A) come from publicly available lists[2].

The data generated was intended to be a good model of what might actually be processed in real-world datasets. The errors introduced in the duplicate records range from small typographical mistakes, to complete change of last names and addresses. When setting the parameters for typographical errors, we used known frequencies from studies in spelling correction algorithms [21, 7, 17]. For this study, the generator selected from 10% to 50% of the generated records for duplication with errors, where the error in the spelling of words,

[1]See `ftp://ftp.denet.dk/pub/wordlists`

[2]Ftp into `cdrom.com` and `cd /pub/FreeBSD/FreeBSD-current/src/share/misc` .

names and cities was controlled according to these published statistics found for common real world datasets.

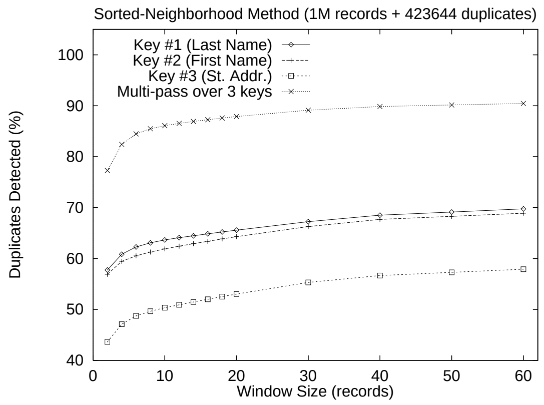
In this paper, the performance measurement of accuracy (percentage of duplicates captured) using this “standard error model” is plotted over varying sized windows so that we may better understand the relationship and tradeoffs between computational complexity and accuracy. We do not believe the results will be substantially different for different databases with the same sorts of errors in the duplicated records. Future work will help to better establish this conjecture over widely varying error models, afforded by our database generator. However, other statistically generated databases may bear no direct relationship to real data. We believe the present experiments are more realistic. Section 5 provides substantial evidence for this case.

## 3.2 Results on accuracy

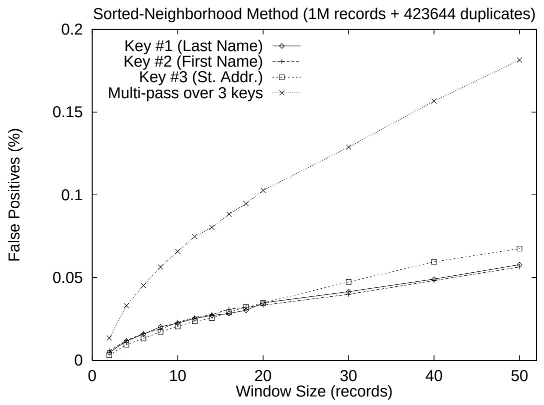
The purpose of this first experiment was to determine baseline accuracy of the sorted-neighborhood method. We ran three independent runs of the sorted-neighborhood method over each database, and used a different key during the sorting phase of each independent run. On the first run the last name was the principal field of the key (i.e., the last name was the first attribute in the key). On the second run, the last name was the principal field, while, in the last run, the street address was the principal field. Our selection of the attribute ordering of the keys was purely arbitrary. We could have used the social-security number instead of, say, the street address. We assume all fields are noisy (and under the control of our data generator to be made so) and therefore it does not matter what field ordering we select for purposes of this study.

Figure 2(a) shows the effect of varying the window size from 2 to 60 records in a database with 1,000,000 records and with an additional 423644 duplicate records with varying errors. A record may be duplicated more than once. Notice that each independent run found from 50% to 70% of the duplicated pairs. Notice also that increasing the window size does not help much and taking in consideration that the time complexity of the procedure goes up as the window size increases, it is obviously fruitless at some point to use a large window.

The line marked as *Multi-pass over 3 keys* in figure 2(a) shows our results when the



(a) Percent of correctly detected duplicated pairs



(b) Percent of incorrectly detected duplicated pairs

Figure 2: Accuracy results for a 1,000,000 records database

program computes the transitive closure over the pairs found by the three independent runs. The percent of duplicates found goes up to almost 90%. A manual inspection of those records not found as equivalent revealed that most of them are pairs that would be hard for a human to identify without further information.

The equational theory is not completely trustworthy. It can decide that two records are similar or equivalent even though they may not represent the same real-world entity; these incorrectly paired records are called "false-positives". Figure 2(b) shows the percent of those records incorrectly marked as duplicates as a function of the window size. The percent of false positives is almost insignificant for each independent run and grows slowly as the window size increases. The percent of false positives after the transitive closure is also very small, but grows faster than each individual run alone. This suggests that the transitive-closure may not be as accurate if the window size of each constituent pass is very large!

The number of independent runs needed to obtain good results with the computation of the transitive closure depends on how corrupt the data is and the keys selected. The more corrupted the data, more runs might be needed to capture the matching records. The transitive closure, however, is executed on pairs of tuple id's, each at most 30 bits, and fast

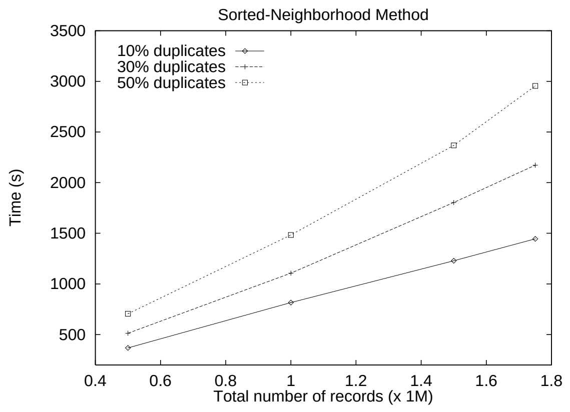


Figure 3: Time performance of the sorted-neighborhood methods for different size databases.

solutions to compute transitive closure exist [2]. From observing real world scenarios, the size of the data set over which the closure is computed is at least one order of magnitude smaller than the corresponding database of records, and thus does not contribute a large cost. But note we pay a heavy price due to the number of sorts or clusterings of the original large data set. We presented some parallel implementation alternatives to reduce this cost in [16].

### 3.2.1 Scaling Up

Finally, we demonstrate that the sorted-neighborhood method scales well as the size of the database increases. Due to the limitations of our available disk space, we could only grow our databases to about 3,000,000 records. We again ran three independent runs of the sorted-neighborhood method, each with a different key, and then computed the transitive closure of the results. We did this for the 12 databases in Table 3.2.1. We started with four (4) "no-duplicate databases" and for each we created duplicates for 10%, 30%, and 50% of the records, for a total of twelve (12) distinct databases. The results are shown in Figure 3. For these relatively large size databases, the time seems to increase linearly as the size of the databases increase independent of the duplication factor.

| Original number of records | Total records | | | Total size (Mbytes) | | |
|---|---|---|---|---|---|---|
| | 10% | 30% | 50% | 10% | 30% | 50% |
| 500000 | 584495 | 754354 | 924029 | 45.4 | 58.6 | 71.8 |
| 1000000 | 1169238 | 1508681 | 1847606 | 91.3 | 118.1 | 144.8 |
| 1500000 | 1753892 | 2262808 | 2770641 | 138.1 | 178.4 | 218.7 |
| 1750000 | 2046550 | 2639892 | 3232258 | 161.6 | 208.7 | 255.7 |

Table 3: Database sizes

## 3.3 Analysis

The natural question to pose is when is the *multi-pass* approach superior to the single-pass case? The answer to this question lies in the complexity of the two approaches for a *fixed accuracy rate* (here we consider the percentage of correctly found matches).

Here we consider this question in the context of a main-memory based sequential process. The reason being that, as we shall see, clustering provides the opportunity to reduce the problem of sorting the entire disk-resident database to a sequence of smaller, main-memory based analysis tasks. The serial time complexity of the *multi-pass* approach (with $r$ passes) is given by the time to create the keys, the time to sort $r$ times, the time to window scan $r$ times (of window size $w$) plus the time to compute the transitive closure. In our experiments, the creation of the keys was integrated into the sorting phase. Therefore, we treat both phases as one in this analysis. Under the simplifying assumption that all data is memory resident (i.e., we are not I/O bound),

$$T_{multipass} = c_{sort} r N \log N + c_{wscan} r w N + T_{closure_{mp}}$$

where $r$ is the number of passes and $T_{closure_{mp}}$ is the time for the transitive closure. The constants depict the costs for comparison only and are related as $c_{wscan} = \alpha c_{sort}$, where $\alpha > 1$. From analyzing our experimental program, the window scanning phase contributes a constant, $c_{wscan}$, which is at least $\alpha = 6$ times as large as the comparisons performed in sorting. We replace the constants in term of the single constant $c$. The complexity of the closure is directly related to the accuracy rate of each pass and depends upon the duplication in the database. However, we assume the time to compute the transitive closure on a database that is orders of magnitude smaller than the input database to be less than the time to scan the input database once (i.e. it contributes a factor of $c_{closure}N < N$). Therefore,

$$T_{multipass} = crN \log N + \alpha crwN + T_{closure_{mp}}$$

for a window size of $w$. The complexity of the single pass sorted-neighborhood method is similarly given by:

$$T_{singlepass} = cN \log N + \alpha cWN + T_{closure_{sp}}$$

for a window size of W.

For a fixed accuracy rate, the question is then for what value of $W$ of the single pass sorted-neighborhood method does the *multi-pass* approach perform better in time, i.e.,

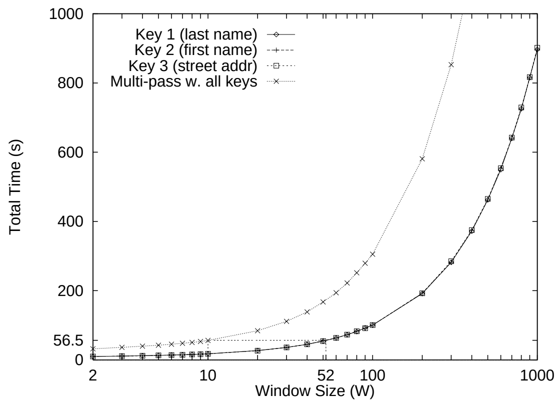
$$cN \log N + \alpha cWN + T_{closure_{sp}} > crN \log N + \alpha crwN + T_{closure_{mp}}$$

or

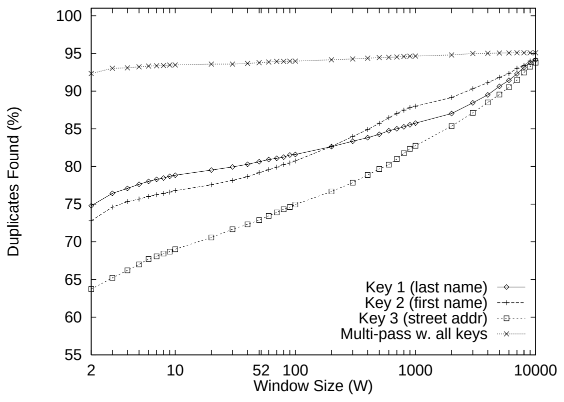
$$W > \frac{r-1}{\alpha} \log N + rw + \frac{1}{\alpha cN} \left( T_{closure_{mp}} - T_{closure_{sp}} \right)$$

To validate this model, we generated a small database of 13,751 records (7,500 original records, 50% selected for duplications, and 5 maximum duplicates per selected record. The total size of the database in bytes was approximately 1 MByte. Once read, the database stayed in core during all phases. We ran three independent single-pass runs using different keys and a multi-pass run using the results of the three single-pass runs. The parameters for this experiment were $N = 13751$ records and $r = 3$. For this particular case where $w = 10$, we have $\alpha \simeq 6$, $c \simeq 1.2 \times 10^{-5}$, $T_{closure_{sp}} = 1.2s$, and $T_{closure_{mp}} = 7$. (Time is specified in seconds ($s$).) Thus, the *multi-pass* approach dominates the single sort approach for these datasets when $W > 41$.

Figure 4(a) shows the time required to run each independent run of the on one processor, and the total time required for the *multi-pass* approach while figure 4(b) shows the accuracy of each independent run as well as the accuracy of the *multi-pass* approach (please note the logarithm scale). For $w = 10$, figure 4(a) shows that the *multi-pass* approach needed



(a) Time for each single-pass runs and the multi-pass run



(b) Ideal vs. Real Accuracy of each run

Figure 4: Time and Accuracy for a Small Database

56.3$s$ to produce an accuracy rate of 93.4% (figure 4(b)). Looking now at the times for each single-pass run, their total time is close to 56$s$ for $W = 52$, slightly higher than estimated with the above model. But the accuracy of all single-pass runs in figure 4(b) at $W = 52$ are from 73% to 80%, well below the 93.4% accuracy level of the *multi-pass* approach. Moreover, no single-pass run reaches an accuracy of more than 93% until $W > 7000$, at which point (not shown in figure 4(a)) their execution time are over 4,800 seconds (80 minutes).

Let us now consider the issue when the process is I/O bound rather than a compute-bound main-memory process. Let $B$ be the number of disk blocks used by the input data set and $M$ the number of memory pages available. Each sorted-neighborhood method execution will access $2B \log_{M-1} B$ disk blocks[3], plus $B$ disk blocks will be read by the window scanning phase. The time for the sorted-neighborhood method can be expressed as:

$$T_{snm} = 2c_{sort} B \log_{M-1} B + c_{wscan} B$$

where $c_{sort}$ represents the CPU cost of sorting the data in one block and $c_{wscan}$ represents the CPU cost of applying the window-scan method to the data in one block.

[3]The 2 comes from the fact that we are counting both read and write operations.

Instead of sorting, we could divide the data into $C$ buckets (e.g., hashing the records or using a multi-dimensional partitioning strategy [15]). We call this modification the *clustering method.* Assuming $M = C+1$, (1 page for each bucket plus one page for processing an input block), we need one pass over the entire data to partition the records into $C$ buckets ($B$ blocks are read). Writing the records into the buckets requires, approximately, $B$ block writes. Assuming the partition algorithm is perfect, each bucket will use $\lceil \frac{B}{C} \rceil$ blocks. We must then sort ($2B \log_{M-1} \lceil \frac{B}{C} \rceil$ block accesses) and apply the window-scanning phase to each bucket independently (approximately $B$ block accesses). In total, the clustering method requires approximately $3B + 2B \log_{M-1} \lceil \frac{B}{C} \rceil$ block accesses. The time for one pass of the clustering method can be expressed as:

$$T_{cluster} = 2c_{cluster}B + 2c_{sort}B \log_{M-1} \lceil \frac{B}{C} \rceil + c_{wscan}B$$

where $c_{cluster}$ is the CPU cost of partitioning one block of data.

Finally, the I/O cost of the *multi-pass* approach will be a multiple of the I/O cost of the method we chose for each pass plus the time needed to compute the transitive closure step. For instance, if we use the clustering method for 3 passes, we should expect about a time of about $3T_{cluster} + T_{xclosure}$.

Figure 5 shows a time comparison between the clustering method and the sorted-neighborhood method. These results where gathered using a generated data set of 468,730 records ($B = 31,250$, block size = 1,024 bytes, $M = 33$ blocks). Notice that in all cases, the clustering method does better than the sorted-neighborhood method. However, the difference in time is not large. This is mainly due to the fact that the equational theory used involved a large number of comparisons making $c_{wscan}$ a lot larger than both $c_{sort}$ and $c_{cluster}$. Thus, even though there are some time savings in initially partitioning the data, the savings are small compared to the overall time cost.

In [16] we describe parallel variants of the basic techniques (including clustering) to show that with a modest amount of "cheap" parallel hardware, we can speed-up the *multi-pass* approach to a level comparable to the time to do a single-pass approach, but with a very high accuracy, i.e. a few small windows ultimately wins.

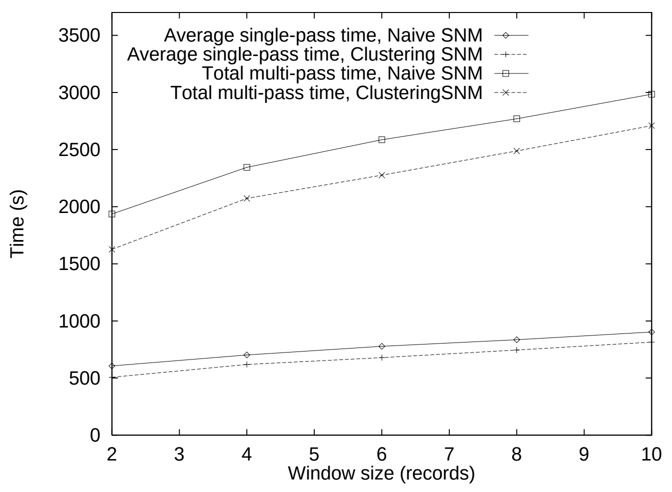


Figure 5: Clustering vs. Basic Sorted Neighborhood Method

## 4 Results on Real-World Data

Even though the results we have achieved on a wide variety of statistically controlled generated data indicate that the multi-pass approach is quite good, some may not regard this as a definitive validation of the efficacy of the techniques.

The State of Washington Department of Social and Health Services maintains large databases of transactions made over the years with state residents. In March of 1995 the Office of Children Administrative Research (OCAR) of the Department of Social and Health Services posted a request on the *KDD-nuggets*[8] asking for assistance analyzing one of their databases. We answered their request and this section details our results.

OCAR analyzes the database of payments by the State to families and businesses that provide services to needy children. OCAR's goal is to answer questions such as: "How many children are in foster care¿‘, "How long do children stay in foster care?" "How many different homes do children typically stay in?" To accurately answer such questions, the many computer records for payments and services must be identified for each child. (Obviously, without matching records with the appropriate individual client, the frequency distributions for such services will be grossly in error.) Because no unique identifier for an individual child exists, it must be generated and assigned by an algorithm that compares multiple service records in the database. The fields used in the records to help identify a child include name, birth date, case number, social security number, each of which is unreliable, containing misspellings, typographical errors, and incomplete information. This is not a unique situation in real-world databases. It was the need to develop computer processes that more accurately identified all records for a given child that spurred OCAR to seek assistance.

## 4.1 Database Description

Most of OCAR's data is stored in one relation that contains all payments to service providers since 1984. There are currently approximately 6,000,000 total records in the relation and this number grows by approximately 50,000 a month. The relation has 19 attributes, of which the most relevant (those carrying information that can be used to infer the identify of individual entities) are: First Name, Last Name, Birthday, Social Security Number, Case Number, Service ID, Service dates (beginning and ending dates), Gender and Race, Provider ID, Amount of Payment, Date of Payment, and Worker ID. Each record is 105 bytes long.

The typical problems with the OCAR data are as follows:

1. Names are frequently misspelled. Sometimes nicknames or "similar sounding" names are used instead of the real name. Also the parent or guardian's name is sometimes used instead of the child's name.

2. Social security numbers or birthdays are missing or clearly wrong (e.g., some records have the social security number "`999999999`"). Likewise, the parent or guardian's information is sometimes used instead of the child's proper information.

3. The case number, which should uniquely identify a family, often changes when a child's family moves to another part of the state, or is referred for service a second time after more than a couple years since the first referral.

4. There are records which cannot be assigned to any person because the name entered in the record was not the child's name, but that of the service provider. Also, names like "`Anonymous Male`" and "`Anonymous Female`" were used. (We call this last type of records *ghost records*.)

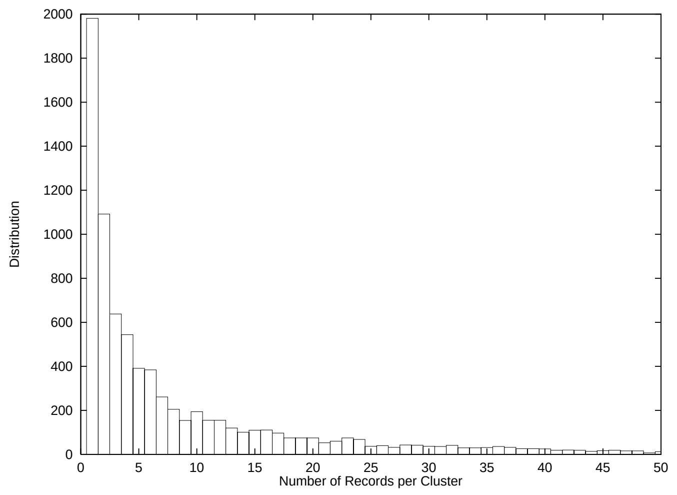


Figure 6: Number of records per Child as computed by OCAR (The graph is drawn only to a cluster size of 50. In actuality, it continues to 500!).

Because of the private nature of the data recorded in the database, we cannot produce sample records to illustrate each of the mentioned cases. Even so, any database administrator responsible for large corporate or agency databases will immediately see the parallels here to their data. (After all, real-world data is very dirty!)

OCAR provided a sample of their database to conduct this study. The sample, which contains the data from only one service office, has 128,438 records (13.6 Mbytes). They also provided us with their current individual identification number for each record in the sample (the number that should uniquely identify each child in the database) according to their own analysis. These OCAR-assigned identifiers serve as our basis for comparing the accuracy over varying window sizes.

Figure 6 shows the distribution of the number of records per individual detected by OCAR. Most individuals in the database are represented on average by 1 to 10 records in the database (approximately 2,000 individuals are represented by 1 record in the database). Note that individuals may be represented by as much as 30-40 records and, although not shown in figure 6, there are some individuals with more than 100 records, and one with about 500 records. Our task is to apply our data cleansing techniques to compute a new individual identification number for each record and compare its accuracy to that attained by the OCAR provided identification number.

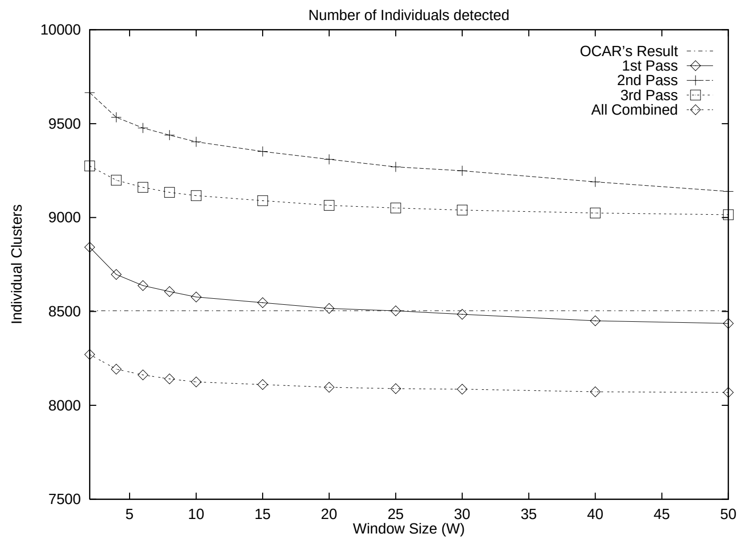


Figure 7: Number of individuals detected

## 4.2 Data Cleansing OCAR's data

OCAR's individual identification numbers are, of course, not perfect. This set of identifiers was computed by a single-scan clustering technique based upon hashing the records using the first four letters of the last name, the first three letters of the first name, the birth month and year, and the case number as hashing keys. This strategy identified 8,504 individuals in the sample database.

Our first task was to create an equational theory in consultation with an OCAR's expert[4]. The resultant rule base consists of 24 rules. We applied this equational theory to the data using the basic sorted-neighborhood method, as well as the multi-pass method for a rigorous comparative evaluation. We used the following keys for each independent run:

1. Last Name, First Name, Social Security Number, and Case Number.
2. First Name, Last Name, Social Security Number, and Case Number.
3. Case Number, First Name, Last Name, and Social Security Number.

[4]Timothy Clark, Computer Information Consultant for OCAR, provided the necessary expertise to define the rule base.

*OCAR's Analysis* : {R1,R2,R3} {R4,R5} {R6,R7,R8} {R9,R10}

*SNM's Analysis* : {R1,R2,R3} {R4} {R5} {R6,R7,R8,R9,R10}

The separation of {R4, R5} by the sorted-neighborhood method is counted as a "possible miss". The union of {R6, R7, R8} {R9, R10} is counted as a "possible false-positive".

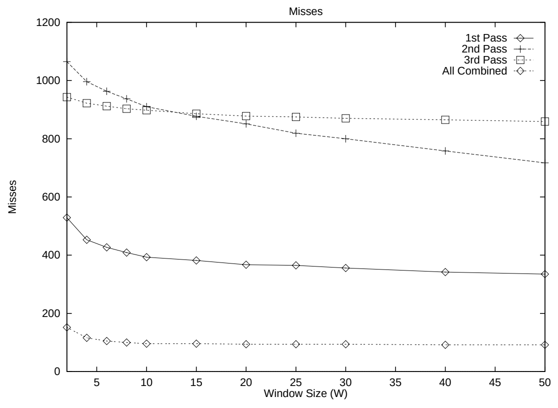
Figure 8: Example of the definition of "possible misses" and "possible false-positives"

Figure 7 displays the number of individuals detected by each independent pass of the basic sorted-neighborhood method and the number of individuals after the closure phase as a function of the window size used. The constant 8,504 individuals detected by OCAR are plotted as a straight line as a means of comparison. As with the statistically generated data, the number of individuals detected here initially goes down as the window size increases but then stabilizes and remains almost constant. Notice also the large improvement in the performance when combining the results of all passes with the transitive closure phase[5]. Thus, the results demonstrated under our controlled studies are validated by this set of real-world data.

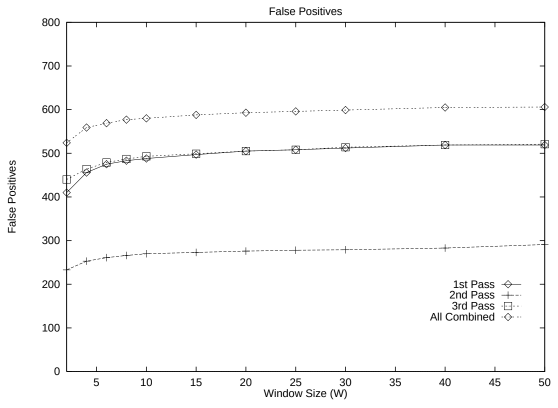
For a window size of 10, the multi-pass process detected 8,125 individuals in the sample. The question we must then answer is whether those 8,125 individuals are closer to the actual number of individuals in the data than OCAR's 8,504 individuals. To answer this question, we looked at the different group of individuals detected by OCAR and our sample results.

We call "possible misses" those groups of individuals that our data cleansing program considered different while OCAR considered similar. We also call "possible false-positives" those groups of individuals that OCAR did not consider similar but where grouped together by our system. Figure 8 shows an example of a "miss" and a "false-positive" under this

[5]Even though we used three passes for the experiments we describe here, two passes using only the first and the third key would have produced almost similar results. The second key pass only marginally improved the results. This significant observation may occur in other real-world data indicating that the multi-pass approach may simply be a "two-pass" approach, significantly reducing the complexity of the process while still achieving accurate results.



(a) Misses with respect to OCAR's results



(b) False Positives with respect to OCAR's results

Figure 9: Accuracy results of sorted-neighborhood method over OCAR's data

definition. Figure 9 depicts the number of possible misses and possible false-positives when comparing our results with OCAR's. As in our previous experiments, the total number of misses goes down as the window size goes up, and drops sharply after the transitive closure phase. The behavior of the false-positives is, as expected, contrary to that of the misses: the number grows with the window size and goes up after the transitive closure phase.

For the multi-pass sorted-neighborhood method to improve on OCAR's results the following two conditions must be met:

- The number of possible misses our data cleansing program correctly did not merge should be larger than the number of "real misses" (those that OCAR correctly merged but our program did not.)
- The number of possible false-positives where records were correctly merged by the multi-pass sorted-neighborhood method should be larger than the real false-positives (cases where our approach incorrectly merged records that OCAR did not).

To study whether our results met the above conditions, we and the OCAR group manually inspected the possible misses and the possible false-positives for the case when the window size was 10. The results of this manual inspection are as follows:

- **Possible Misses**: The multi-pass sorted-neighborhood method failed to detect 96 individuals that OCAR detected. Of these 96 possible misses:
  - 44 (45.8%) were correctly separated by our approach and therefore not real misses. (OCAR's results on these are wrong.)
  - 26 (27.1%) were incorrectly separated by our approach and therefore real misses.
  - 26 (27.1%) were special cases involving "ghost" records or records of payments to outside agencies. We agreed with OCAR to exclude these cases from further consideration.
- **Possible False Positives**: There were 580 instances of the multi-pass sorted-neighborhood method joining records as individuals that OCAR's did not. Of these 580 cases, we manually inspected 225 (38.7%) of them with the following results:
  - 14.0% of them were incorrectly merged by our approach.
  - 86.0% where correctly merged by our approach.

By way of summary, 45.8% of the possible misses are not real misses but correctly classified records, and an estimated 86.0% of the possible false-positives are not real false positives. These results lead OCAR to be confident that the multi-pass sorted-neighborhood method will improve their individual detection procedure.

## 5 Incremental Merge/Purge

All versions of the sorted-neighborhood method we discussed in section 2 started the procedure by first concatenating the input lists of tuples. This concatenation step is unavoidable and presumably acceptable the first time a set of databases is received for processing. However, once the data has been cleansed (via the merge/purge process) and stored for future use, concatenation of this processed data with recently arrived data before re-applying a merge/purge process might not be the best strategy to follow. In particular, in situations where new increments of data are available in short periods of time, concatenating all data before merging and purging could prove prohibitively expensive in both time and space required. In this section we describe an incremental version of the sorted-neighborhood procedure and provide some initial time and accuracy results for statistically generated datasets.

Figure 10 summarizes an incremental Merge/Purge algorithm. The algorithm specifies a loop repeated for each increment of information received by the system. The increment is concatenated to a relation of prime-representatives pre-computed from the previous run of the incremental Merge/Purge algorithm, and any multi-pass sorted-neighborhood method is applied to the resulting relation. Here prime-representatives are a set of records extracted from each cluster of records used to represent the information in its cluster. From the pattern recognition community, we can think of these prime-representatives as analogous to the "cluster centroids" [10] generally used to represent clusters of information, or as the base element of an equivalence class.

Initially, no previous set of prime-representatives exists and the first increment is just the first input relation. The concatenation step has, therefore, no effect. After the execution of the merge/purge procedure, each record from the input relation can be separated into clusters of similar records. The first time the algorithm is used, all records will go into new clusters. Then, starting the second time the algorithm is executed, records will be added to previously existing clusters as well as new clusters.

Of particular importance to the success of this incremental procedure, in terms of accuracy of the results, is the *correct* selection of the prime-representatives of each formed cluster. As with many other phases of the merge/purge procedure, this selection is also a knowledge-intensive operation where the domain of each application will determine what is a good set of prime-representatives. Before describing some strategies for selecting these representatives, note that the description of step 4 of the algorithm in Figure 10 also implies that for some clusters, the best prime-representative is *no* representative at all. For a possible practical example where this strategy is true, consider the OCAR data described in chapter 4. There, clusters containing records dated as more than 10 years old are very unlikely to receive a new record. Such clusters can be removed from further consideration by not selecting a prime-representative for them.

In the case where one or more prime-representatives per cluster are necessary, here are some possible strategies for their selection:

**Definitions:**

| | | |
|---|---|---|
| $R_0$ | : | The initial relation. |
| $\Delta_i$ | : | The $i$-th increment relation. |
| $c_i$ | : | A relation of only "prime representatives" of the clusters identified by the Merge/Purge procedure. |

**Initially:**

```
$\Delta_0 \leftarrow R_0$
$c_0 \leftarrow \emptyset$
$i \leftarrow 0$
```

**Incremental Algorithm:**

```
For every $\Delta_i$ do:
begin
    1. Let $I_i \leftarrow \mathit{CONCATENATE}(c_i, \Delta_i)$.
    2. Apply any Merge/Purge procedure to $I_i$. The result is a cluster
       assignment for every record in $I_i$.
    3. Separate each record in $I_i$ into the clusters assigned by the
       previous step.
    4. For every cluster of records, if necessary, select one or more
       records as prime representatives for the cluster. Call the
       relation formed of all selected prime representatives, $c_{i+1}$.
end.
```

Figure 10: Incremental Merge/Purge Algorithm

- **Random Sample**: Select a sample of records at random from each cluster.
- **N-Latest**: Data is sometimes physically ordered by the time of entry into the relation. In many such cases, the most recent elements entered in the database can be assumed to better represent the cluster (e.g., the OCAR data is such an example). In this strategy, the $N$ latest elements are selected as prime-representatives.
- **Generalization**: Generate the prime-representatives by generalizing the data collected from several positive examples (records) of the concept represented by the cluster. Techniques for generalizing concepts are well known from machine learning[9, 18].
- **Syntactic**: Choose the largest or more complete record.
- **Utility**: Choose the record that matched others more frequently.

In this section we present initial results comparing the time and accuracy performance of incremental Merge/Purge with the basic Merge/Purge algorithm. We selected the N-Latest prime-representative strategy for our experiments for its implementation simplicity. Experiments are underway to test and compare all the above strategies. Results will be described in a future report.

Two important assumptions were made while describing the Incremental Merge/Purge algorithm. First, it was assumed that no data previously used to select each cluster's prime-representative will be deleted (i.e., no *negative deltas*). Second, it was also assumed that no changes in the rule-set will occur after the first increment of data is processed. We now discuss, briefly, the implications of these two assumptions.

Removing records already clustered could split some clusters. If a removed record was responsible for merging two clusters, the original two clusters so merged will become separated. Two new prime-representatives must be computed before the next increment of data arrives for processing. The procedure to follow in case of deletions is the following:

1. Delay all deletions until after step 3 of the Incremental Algorithm in Figure 10.
2. Perform all deletions. Remember cluster IDs of all clusters affected.
3. Re-compute the closure in all clusters affected, splitting existing clusters as necessary.

Then, the Incremental Algorithm resumes at step 4 by recomputing a new prime-representative for all clusters, including the new one formed after the deletions.

Changes to the data are a little more difficult. Changes could be treated as a deletion followed by an insertion. However, it is often the case (in particular, if it is a human making the change) that the new record should belong to the same cluster as the removed one. Here a user-set parameter should determine how and in what circumstances changes to data should be treated as a deletion followed by an insertion (to be evaluated in the next increment evaluation) or just a direct change into an existing cluster.

Changes of the rule-base defining the equational theory are even more difficult to correctly incorporate into the Incremental Algorithm. Minor changes to the rule-base (for example, small changes to some thresholds defining equality over two fields, deletion of rules that have rarely fired) are expected to have little impact on the contents of the formed clusters. Nonetheless, depending on the data or if major changes are made to the rule-base, a large number of current clusters could be erroneous. Unfortunately, the only solution to this problem is to run a Merge/Purge procedure once again using all available data. On the other hand, depending on the application, a slight number of inconsistencies might be acceptable therefore avoiding the need to run the entire procedure. Here, once again, the decision is highly application dependent and requires human intervention to resolve.

## 5.1 Initial experimental results on the Incremental Algorithm

We conducted a number of experiments to test the incremental Merge/ Purge algorithm. In these experiments we were interested in studying the time performance of the different stages of the algorithm and the effect on the accuracy of the results.

To this end, we started with the OCAR sample described in section 4.1 (128,439 records) and divided it into five (5) parts, $\Delta_0, \Delta_1, \Delta_2, \Delta_3, \Delta_4$, with $25,000$, $25,000$, $25,000$, $25,000$ and $28,439$ records, respectively. The incremental Merge/Purge algorithm was implement as a UNIX shell script which concatenated and fed the proper parts to the basic multi-pass sorted-neighborhood method. An *AWK* script combined with a C program was used to implement the prime-representative selection part of the algorithm. The only strategy tested was the N-latest strategy, where $N = 1$ (i.e., only the latest record in a cluster was

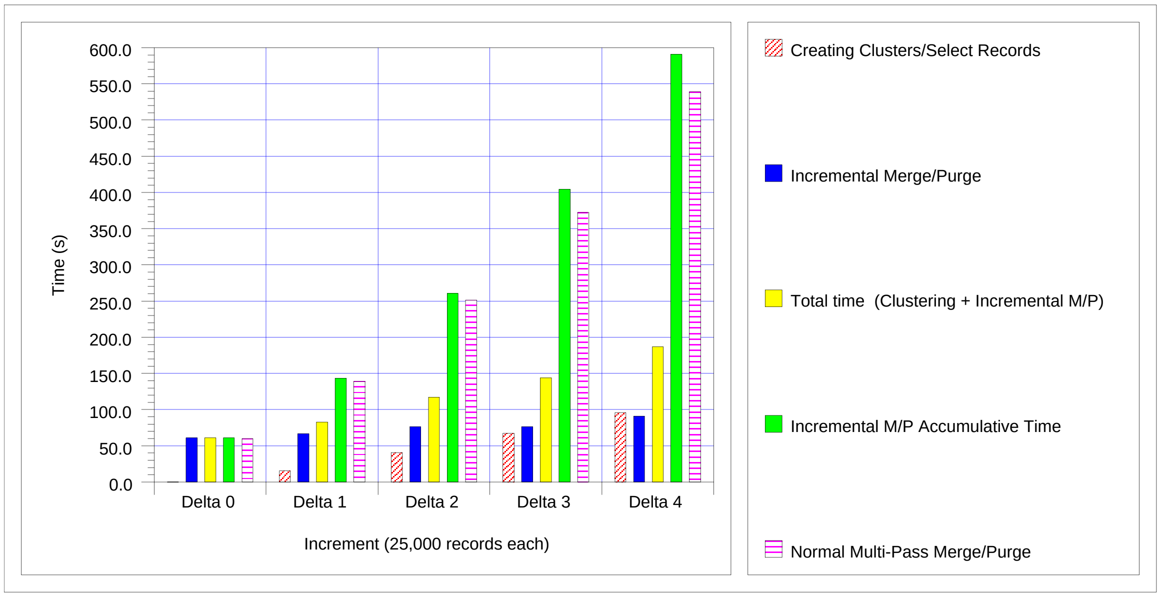


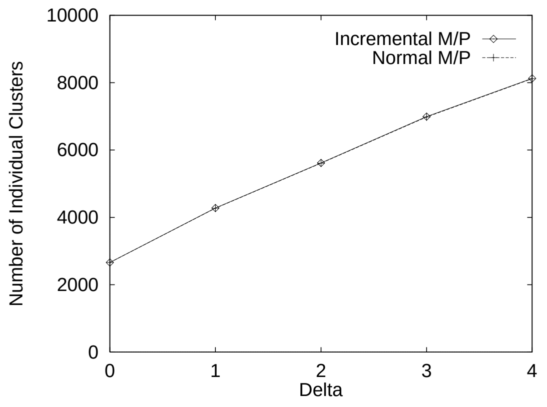
Figure 11: Incremental vs. Normal Multi-Pass Merge/Purge Times

used as prime-representative).

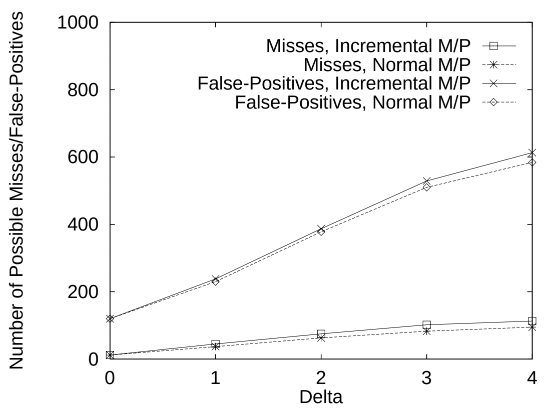
Figure 11 shows the time results for the five-part incremental Merge/Purge procedure in contrast to the normal (non-incremental) Merge/Purge. These results were obtained with a three-pass basic multi-pass approach using the keys described in section 4.2, a window-size of 10 records, and using a Sun 5 Workstation running Solaris 2.3.

The results in Figure 11 are divided by deltas. Five bars, each representing the actual time for a particular measured phase, are present for each division in Figure 11. The first bar corresponds to the time taken to collect the prime-representatives of the previous run of the multi-pass approach (note this bar is 0 for the first delta). The second bar represents the time for executing the multi-pass approach over the concatenation of the current delta with the prime-representatives records. The total time for the incremental Merge/Purge process is the addition of these two times and is represented by the third bar. The fourth bar shows the accumulated total time after each incremental Merge/Purge procedure. Finally, the last bar shows the time for a normal Merge/Purge procedure running over a databases composed of the concatenation of all deltas, up to and including the current one.

Notice that for every case after the first delta, the total time for the incremental Merge/Purge process is considerably less than the time for the normal process. For all cases tested in this



(a) Clusters formed



(b) Possible Misses and False-Positives

Figure 12: Accuracy of the Incremental M/P procedure

experiment, the cumulative time for the incremental Merge/Purge process was larger than the total time for the normal Merge/Purge. This is due to the large time cost of clustering and selecting the prime-representative records for each cluster. In the current implementation, the entire dataset (the concatenation of all deltas, up to an including the current one) is sorted to find the clusters and all records in the cluster are considered when selecting the prime-representatives. This is clearly not the optimal solution for the clustering of records and the selection of prime-representatives. A better implementation could incrementally select prime-representatives based on the previously computed one. The current implementation, nonetheless, gives a "worst-case" execution time for this phase. Any optimization will only decrease the total incremental Merge/Purge time.

Finally, Figure 12 compares the accuracy results of the incremental Merge/Purge procedure with the normal procedure. The total number of individuals (clusters) detected, the number of possible misses and the number of possible false-positives went up with the use of the incremental Merge/Purge procedure. Nonetheless, the increase of all measures is almost negligible and arguably acceptable given the remarkable reduction of time provided by the incremental procedure.

## 6 Conclusion

The sorted-neighborhood method is expensive due to the sorting phase, as well as the need to search in large windows for high accuracy. Alternative methods based on data clustering modestly improves the process in time as reported elsewhere. However, neither achieves high accuracy without inspecting large neighborhoods of records. Of particular interest is that performing the data cleansing process multiple times over small windows, followed by the computation of the transitive closure, dominates in accuracy for either method. While multiple passes with small windows increases the number of successful matches, small windows also favor decreases in false positives, leading to high overall accuracy of the merge phase. An alternative view is that a single pass approach would be far slower to achieve a comparable accuracy as a multi-pass approach.

The results we demonstrate for statistically generated databases provide the means of quantifying the accuracy of the alternative methods. In real-world data we have no comparable means of rigorously evaluating these results. Nevertheless, the application of our program over real-world data provided by the State of Washington Child Welfare Department has validated our claims of improved accuracy of the multi-pass method based upon "eye-balling" a significant sample of data. Thus, what the controlled empirical studies have shown indicates that improved accuracy will be exhibited for real world data with the same sorts of errors and complexity of matching as described in this paper.

Finally, the results reported here form the basis of a *DataBlade* Module available from Informix Software as the *DataCleanser DataBlade*. The technology is broadly applicable; after all, real world data is dirty.

## 7 Acknowledgments

We are grateful to Timothy Clark, Computer Information Consultant for OCAR, for the help provided obtaining the results in section 4. Thanks also to Dr. Diana English, OCAR's Chair, for allowing the use of their database in our work.

## References

[1] ACM. SIGMOD record, December 1991.

[2] R. Agrawal and H. V. Jagadish. Multiprocessor Transitive Closure Algorithms. In *Proc. Int'l Symp. on Databases in Parallel and Distributed Systems*, pages 56–66, December 1988.

[3] C. Batini, M. Lenzerini, and S. Navathe. A Comparative Analysis of Methodologies for Database Schema Integration. *ACM Computing Surverys*, 18(4):323–364, December 1986.

[4] D. Bitton and D. J. DeWitt. Duplicate Record Elimination in Large Data Files. *ACM Transactions on Database Systems*, 8(2):255–265, June 1983.

[5] B. P. Buckles and F. E. Petry. A fuzzy representation of data for relational databases. *Fuzzy Sets and Systems*, 7:213–226, 1982. Generally regarded as the paper that originated Fuzzy Databases.

[6] J. P. Buckley. A Hierarchical Clustering Strategy for Very Large Fuzzy Databases. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, pages 3573–3578, 1995.

[7] K. W. Church and W. A. Gale. Probability Scoring for Spelling Correction. *Statistics and Computing*, 1:93–103, 1991.

[8] T. K. Clark. Analyzing Foster Childrens' Foster Home Payments Database. In KDD Nuggets 95:7 (http://info.gte.com/˜kdd/nuggets/95/), Piatetsky-Shapiro, ed., 1995.

[9] T. Dietterich and R. Michalski. A Comparative Review of Selected Methods for Learning from Examples. In R. Michalski, J. Carbonell, and T. Mitchell, editors, *Machine Learning*, volume 1, pages 41–81. Morgan Kaufmann Publishers, Inc., 1983.

[10] R. Dubes and A. Jain. Clustering Techniques: The User's Dilema. *Pattern Recognition*, 8:247–260, 1976.

[11] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. From Data Mining to Knowledge Discovery in Databases. *AI Magazine*, 17(3), Fall 1996.

[12] I. Fellegi and A. Sunter. A Theory for Record Linkage. *American Statistical Association Journal*, pages 1183–1210, December 1969.

[13] C. L. Forgy. OPS5 User's Manual. Technical Report CMU-CS-81-135, Carnegie Mellon University, July 1981.

[14] R. George, F. E. Petry, B. P. Buckles, and R. Srikanth. Fuzzy Database Systems – Challenges and Opportunities of a New Era. *International Journal of Intelligent Systems*, 11:649–659, 1996.

[15] S. Ghandeharizadeh. *Physical Database Design in Multiprocessor Database Systems*. PhD thesis, Department of Computer Science, University of Wisconsin - Madison, 1990.

[16] M. Hernández and S. Stolfo. The Merge/Purge Problem for Large Databases. In *Proceedings of the 1995 ACM-SIGMOD Conference*, May 1995.

[17] K. Kukich. Techniques for Automatically Correcting Words in Text. *ACM Computing Surveys*, 24(4):377–439, 1992.

[18] M. Lebowitz. Not the Path to Perdition: The Utility of Similarity-Based Learning. In *Proceedings of 5th National Conference on Artificial Intelligence*, pages 533–537, 1986.

[19] A. Monge and C. Elkan. An Efficient Domain-independent Algorithm for Detecting Approximate Duplicate Database Records. In *Proceedings of the 1997 SIGMOD Workshop on Research Issues on DMKD*, pages 23–29, 1997.

[20] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. AlphaSort: A RISC Machine Sort. In *Proceedings of the 1994 ACM-SIGMOD Conference*, pages 233–242, 1994.

[21] J. J. Pollock and A. Zamora. Automatic spelling correction in scientific and scholarly text. *ACM Computing Surveys*, 27(4):358–368, 1987.

[22] T. Senator, H. Goldberg, J. Wooton, A. Cottini, A. Umar, C. Klinger, W. Llamas, M. Marrone, and R. Wong. The FinCEN Artificial Intelligence System: Identifying Potential Money Laundering from Reports of Large Cash Transactions. In *Proceedings of the 7th Conference on Innovative Applications of AI*, August 1995.

[23] Y. R. Wang and S. E. Madnick. The Inter-Database Instance Identification Problem in Integrating Autonomous Systems. In *Proceedings of the Sixth International Conference on Data Engineering*, February 1989.

## A OPS5 version of the equational theory

```
/*
 * rule_program ( number of tuples, first tuple, window size )
 *   Compare all tuples inside a window.  If a match is found,
 *   call merge_tuples().
 */
void
rule_program(int ntuples, int start, int wsize)
{
  register int i, j;
  register WindowEntry *person1, *person2;
  boolean similar_ssns, similar_names, similar_addrs;
  boolean similar_city, similar_state, similar_zip;
  boolean very_similar_addres, very_close_aptm,
          very_close_stnum, not_close;

  /* For all tuples under consideration */
  for (j = start;  j < ntuples;  j++) {
    /* person2 points to the j-th tuple */
    person2 = &tuples[j];
    /* For all other tuples inside the window (wsize-1 tuples
     * before the j-th tuple).
     */
    for (i = j − 1;  i > j−wsize && i ≥ 0;  i−−) {
      /* person1 points to the i-th tuple */
      person1 = &tuples[i];

      /* Compare person1 with person2 */

      /* RULE:  find-similar-ssns    */
      similar_ssns = same_ssn_p(person1→ssn,person2→ssn,3);

      /* RULE: compare-names   */
      similar_names =
        compare_names (
             person1→name, person2→name,
             person1→fname, person1→minit, person1→lname,
             person2→fname, person2→minit, person2→lname,
             person1→fname_init, person2→fname_init
             );

      /* RULE: same-ssn-and-name */
      if (similar_ssns && similar_names) {
        merge_tuples(person1, person2);
        continue;
      }

      /* RULE: compare-addresses */
      similar_addrs = compare_addresses(person1→stname,
                                     person2→stname);

      /* Compare other fields of the address */
      similar_city = same_city(person1→city, person2→city);
      similar_zip =  same_zipcode(person1→zipcode,
                               person2→zipcode);
      similar_state =
               (strcmp(person1→state, person2→state) == 0);

      /* RULEs: closer-addresses-use-zips and
       *         closer-address-use-states
       */
      very_similar_addrs =
        (similar_addrs && similar_city &&
         (similar_state || similar_zip));

      /* RULEs:  same-ssn-and-address and
       *          same-name-and-address
       */
      if ((similar_ssns || similar_names) &&
          very_similar_addrs) {
        merge_tuples(person1, person2);
        continue;
      }

      not_close = close_but_not_much(person1→stname,
                                 person2→stname);

      if (person1→stnum && person2→stnum)
        very_close_stnum = very_close_num(person1→stnum,
                                   person2→stnum);
      else
        very_close_stnum = FALSE;

      if (person1→aptm && person2→aptm)
        very_close_aptm = very_close_str(person1→aptm,
                                   person2→aptm);
      else
        very_close_aptm = FALSE;

      /* RULEs: compare-addresses-use-numbers-state,
       *         compare-addresses-use-numbers-zipcode, and
       *         same-address-except-city
       */
      if ((very_close_stnum && not_close && very_close_aptm
           && similar_city &&
          (similar_state || similar_zip) && !similar_addrs) ||
          (similar_addrs && very_close_stnum &&
           very_close_aptm && similar_zip)) {
        very_similar_addrs = TRUE;

        /* RULEs: same-ssn-and-address and
         *         same-name-and-address (again) */
        if (similar_ssns || similar_names) {
          merge_tuples(person1, person2);
          continue;
          }
      }

      /* RULE: very-close-ssn-close-address */
      if (similar_addrs && similar_ssns && !similar_names)
        if (same_ssn_p (person1→ssn, person2→ssn, 2)) {
          merge_tuples(person1, person2);
          continue;
        }


      /* RULE: hard-case-1 */
      if (similar_ssns && very_similar_addrs && similar_zip &&
          same_name_or_initial(person1→fname,person2→fname)) {
        merge_tuples(person1, person2);
        continue;
      }

    }
  }
}
```

## B C version of the equational theory

```
/*
 * rule_program ( number of tuples, first tuple, window size )
 *   Compare all tuples inside a window.  If a match is found,
 *   call merge_tuples().
 */
void
rule_program(int ntuples, int start, int wsize)
{
  register int i, j;
  register WindowEntry *person1, *person2;
  boolean similar_ssns, similar_names, similar_addrs;
  boolean similar_city, similar_state, similar_zip;
  boolean very_similar_addres, very_close_aptm,
          very_close_stnum, not_close;

  /* For all tuples under consideration */
  for (j = start;  j < ntuples;  j++) {
    /* person2 points to the j-th tuple */
    person2 = &tuples[j];
    /* For all other tuples inside the window (wsize-1 tuples
     * before the j-th tuple).
     */
    for (i = j − 1;  i > j−wsize && i ≥ 0;  i−−) {
      /* person1 points to the i-th tuple */
      person1 = &tuples[i];

      /* Compare person1 with person2 */

      /* RULE:  find-similar-ssns    */
      similar_ssns = same_ssn_p(person1→ssn,person2→ssn,3);

      /* RULE: compare-names    */
      similar_names =
        compare_names (
             person1→name, person2→name,
             person1→fname, person1→minit, person1→lname,
             person2→fname, person2→minit, person2→lname,
             person1→fname_init, person2→fname_init
             );

      /* RULE: same-ssn-and-name */
      if (similar_ssns && similar_names) {
        merge_tuples(person1, person2);
        continue;
      }

      /* RULE: compare-addresses */
      similar_addrs = compare_addresses(person1→stname,
                                   person2→stname);

      /* Compare other fields of the address */
      similar_city = same_city(person1→city, person2→city);
      similar_zip =  same_zipcode(person1→zipcode,
                            person2→zipcode);
      similar_state =
               (strcmp(person1→state, person2→state) == 0);

      /* RULEs: closer-addresses-use-zips and
       *          closer-address-use-states
       */
      very_similar_addrs =
        (similar_addrs && similar_city &&
         (similar_state || similar_zip));

      /* RULEs:  same-ssn-and-address and
       *           same-name-and-address
       */
      if ((similar_ssns || similar_names) &&
          very_similar_addrs) {
        merge_tuples(person1, person2);
        continue;
      }

      not_close = close_but_not_much(person1→stname,
                                person2→stname);

      if (person1→stnum && person2→stnum)
        very_close_stnum = very_close_num(person1→stnum,
                                    person2→stnum);
      else
        very_close_stnum = FALSE;

      if (person1→aptm && person2→aptm)
        very_close_aptm = very_close_str(person1→aptm,
                                   person2→aptm);
      else
        very_close_aptm = FALSE;

      /* RULEs: compare-addresses-use-numbers-state,
       *          compare-addresses-use-numbers-zipcode, and
       *          same-address-except-city
       */
      if ((very_close_stnum && not_close && very_close_aptm
           && similar_city &&
          (similar_state || similar_zip) && !similar_addrs) ||
          (similar_addrs && very_close_stnum &&
           very_close_aptm && similar_zip)) {
        very_similar_addrs = TRUE;

        /* RULEs: same-ssn-and-address and
         *          same-name-and-address (again) */
        if (similar_ssns || similar_names) {
          merge_tuples(person1, person2);
          continue;
          }
      }

      /* RULE: very-close-ssn-close-address */
      if (similar_addrs && similar_ssns && !similar_names)
        if (same_ssn_p (person1→ssn, person2→ssn, 2)) {
          merge_tuples(person1, person2);
          continue;
        }


      /* RULE: hard-case-1 */
      if (similar_ssns && very_similar_addrs && similar_zip &&
          same_name_or_initial(person1→fname,person2→fname)) {
        merge_tuples(person1, person2);
        continue;
      }

    }
  }
}
```