A Generalization of Band Joins and The Merge/Purge Problem

Mauricio A. Hernández

THESIS PROPOSAL
(CUCS-005-95)

Department of Computer Science
Columbia University
New York, NY 10027

February 1995

Contents

List of Figures

1 Introduction

Merging large databases acquired from different sources with heterogenous representations of information has become an increasingly difficult problem for many organizations. Instances of this problem appearing in the literature have been called the *semantic integration* problem [ACM, 1991] or the *instance identification* problem [Wang and Madnick, 1989]. In this thesis we consider the problem over 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 common that many magazine subscription databases are purchased for the specific purpose of identifying characteristic interests of people for directed marketing purposes. 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.

The problem of merging two relations can be solved by a simple sort followed by a duplicate elimination phase. However, when both relations are heterogeneous, meaning they do not share the same schema, or that the same real-world entities are represented differently in both relations, the problem of merging becomes more difficult. The first issue, where relations have different schema, has been addressed extensively in the literature and is known as the *schema integration* problem [Batini *et al.*, 1986]. This problem is outside the scope of this thesis and is not discussed further. We are primarily interested in the second problem: heterogeneous representations of data and its implication when merging or joining relations.

Another simple way to find duplicates among two relations R and S is to compute the *equijoin* $R \bowtie S$. The naive means of implementing joins is to compute the Cartesian product, a quadratic time process, followed by the selection of relevant tuples. The obvious optimizations as well as parallel variants for join computation are well known: sort-merge and hash partitioning. These strategies assume a total ordering over the domain of the join attributes (an index is thus easily computable) or a "near perfect" hash function that provides the means of inspecting small partitions of tuples when computing the join. In the case we study here, we cannot assume there exists a total ordering, nor a perfect hash distribution that would lead to a completely accurate result, meaning even slight errors in the data imply all possible "matches" of common data about the same entity may not be found. However, the techniques we study and have partially implemented are based upon these two strategies for fast execution, with the particular desire to improve their accuracy.

When these "errors" in the data are not severe, we might ideally expect to find the matching instance of a tuple in R within a "band" of tuples in S. This type of *non-equijoin* joins are called *band-joins* and have been studied by [DeWitt *et al.*, 1991]. A band-join is a join between two relations R and S where the join-predicate, θ, has the form $R.A - c_1 \leq S.B \leq R.A + c_2$, where $R.A$ and $S.B$ represent the join attributes of R and S, respectively, and $c_1 > 0$, $c_2 > 0$ are numeric constants. In this thesis we generalize the definition of band joins by allowing θ to be a more complex function than just a simple arithmetic predicate defined over a totally ordered numeric domain.

For correctness, let R and S be two relations and $Attrs(R)$ and $Attrs(S)$ represent the sets of attributes of each respective relation. We study the efficient execution of queries of the form

$$M \leftarrow \pi_{\mathcal{A}}\left(\sigma_{\theta}\left(R \times S\right)\right)$$

where $\mathcal{A} \subseteq \{Attrs(R) \cup Attrs(S)\}$ and θ is a boolean function $\mathcal{F}(\mathcal{X})$, $\mathcal{X} \leftarrow \{\mathcal{B} \subseteq Attrs(R) \cup \mathcal{C} \subseteq Attrs(S)\}$ ($\mathcal{B}$ and $\mathcal{C}$ are both nonempty). The function $\mathcal{F}$ could be a simple arithmetic predicate or a complex inference procedure defined over a mixture of domains of the chosen attributes.

There are several application problems in which the generalization of band-joins is used. One example is *intersection spatial-joins* [Brinkhoff *et al.*, 1994], where a possible set of spatial objects are first identified and then a more complex geometric filter is applied to determine which objects satisfy the spatial join predicate. Another example is the strategy used in ALEXSYS [Stolfo *et al.*, 1990], an expert system for allocating mortgage pools, where pools that can be successfully allocated can be found "close" to one another if an initial "order" is given to the data. Determining if two or more pools can be allocated to a contract is determined by the rules in the expert system. We will discuss both of these applications as part of the Research Plan in section 6. A third and more common example of an application problem is *merge/purge*. The merge/purge problem is probably the best known example of the instance identification problem and will be the main case study in the proposed thesis.

Merge/purge is ubiquitous in modern commercial organizations, and is typically solved today by expensive *mainframe computing* solutions. Here we consider the opportunity to solve merge/purge on low cost shared-nothing multiprocessor architectures. Such approaches are expected to grow in importance with the coming age of very large network computing architectures where many more distributed databases containing information on a variety of topics will be generally available for public and commercial scrutiny.

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, as detailed in section 3.3.

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 thesis proposal we first discuss two related 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. We then explore the approach of partitioning the data into meaningful clusters and then bringing the matching records on each individual cluster close together by sorting. In this second algorithm we need not compute an entire sort of the full data-set, but rather a number of substantially smaller, independent and concurrent sorts that can be performed more efficiently on reduced datasets. However, we demonstrate that, as one may expect, neither of these basic approaches alone can guarantee high accuracy.

The contributions of this thesis are as follows. We detail a system we have partially 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. Alternative algorithms that were implemented for the fundamental merge process are comparatively evaluated and demonstrate that no single pass over the data using one particular scheme as a key performs as well as computing the transitive closure over several independent runs each using a different key for ordering data. We show, for example, that *multiple passes* followed by the computation of the closure consistently dominates in accuracy for only a modest performance penalty. Finally, we discuss the computational costs of these alternative approaches and demonstrate fully implemented parallel solutions to speed up the process over a serial solution. The moral is simply that several distinct "cheap" passes over the data produces more accurate results than one "expensive" pass over the data. (Our preliminary results detailing this approach has been accepted for publication at the 1995 ACM-SIGMOD Conference [Hernández and Stolfo, 1995].)

This thesis work is far from complete. The experimental results provided in this proposal verifies our core ideas for the solution method we propose. Additional work is needed in the *purge* phase of the merge/purge procedure. Like the merge phase, the implementation of the purge phase requires a large amount of domain-specific knowledge. The particular application we explore in this proposal, duplicate elimination from a list of names, is just one of the many other applications where the use of a general band join process could be of benefit. The design of a tool to elicit this knowledge from a user will be presented at the end of this proposal. Also, to demonstrate the general utility of the methods proposed in this thesis, we will explore two other applications of our facility, one an expert system for financial applications and the other a spatial-join over geometric features.

The rest of this proposal is organized as follows. In the next section, we provide a description of several lines of research that are related to our work. Then, in section 3, we describe the specific details of the merge/purge problem we study, and three solutions based upon sorted-neighborhood searching and clustering. Section 4 presents experimental results for our solution method and section 5 briefly explores parallel processing of these solutions. Finally, we detail in section 6 the work that remains to conclude our thesis.

2 Previous Work

Several lines of work have baring on efficient solutions for the merge/purge problem. The semantic integration problem [Kent, 1991] seeks to identifying a multiplicity of database objects that represent the same or related real-world entity, even though their database representations are different. This problem has been studied by the *heterogenous multi-database* community. The solutions we present for the merge/purge problem are mainly based on sorting the database to bring the "matching" tuples to a close neighborhood in the resulting sorted database. Sorting data is probably the most studied problem in Computer Science and many different algorithms have been presented over the years [Knuth, 1973]. We are however only interested in sorting algorithms in which *duplicates* are removed from the final sorted database. To date, the most relevant work in this area is [Bitton and DeWitt, 1983]. Finally, the proposed solution to the merge/purge problem resembles a *sort-merge join* [Gotlieb, 1975] in which the join condition is a user-defined equivalence function. Of particular relevance to the merge/purge solution proposed here is the work on "band-joins" by [DeWitt *et al.*, 1991]. We briefly describe each one of these lines of work in the following sections.

2.1 Heterogenous Multi-Databases

A *database system* consists of a software component called a database management system and a set of databases it manages. Centralized database systems dominated the research scene during the 1970's. As network technology decentralized most computing facilities, the need for distributed repositories and management of distributed data became necessary. *Distributed database systems* (DDBS) through their distributed database management system, provide consistent access to the data making the partition of the data invisible to end-users. But with the creation of multiple and possibly *heterogenous* DDBS managing a diversity of information sources, the problem of providing consistent access to data managed by different DBMS has become increasingly difficult [Thomas *et al.*, 1990]. Those systems that attempt to provide an unambiguous schema for a collection of heterogeneous databases are known as *Federated database systems* [Sheth and Larson, 1990] or *Heterogeneous database systems*. A more loosely-coupled collection of databases for which no DBMS provides consistent schemas among them are called *Multidatabase Systems* (MDBS) [Elmagarmid and Pu, 1990].

Semantic Heterogeneity has been recognized as a difficult problem in MDBS. Recently, ACM SIGMOD dedicated a special issue to this problem [ACM, 1991]. In that issue, [Kent, 1991] explains how many assumptions in centralized database systems cannot be taken for granted when using a multidatabase system. Some of his examples are:

1. Identity and Naming: The principal assumption on which modeling with databases rests is a one-to-one correspondence between *proxy* objects in the database and the *entity objects* in the real world the proxies are supposed to represent. This assumption works fine on a centralized – one copy of the relation– system. But in a multidatabase

system, the same entity object might be represented differently at several databases. Kent describes the problem as follows:

> "How should we know that two proxies represent the same entity, that $x \equiv y$ even though $x \neq y$? That's the $64,000 question."

2. Constraints: Enforcing the constraint that two employees cannot have the same employee identification number is rather easy and well understood on a centralized system. This is, however, not necessarily true in a multidatabase system, where independent agents can create different identification numbers for the same employee.

3. Certitude: Simple databases look very certain about their information. Asked about the birthday of an employee, the DBS will return one date. On multidatabase systems, because of data entry errors or heterogeneous representations, different strings might be returned for the same query.

Of particular interest for us in this thesis proposal is the identity and naming problem. This problem has been also called the *inter-database identification problem* by [Wang and Madnick, 1989]. [Kent, 1991] proposes the use of *spheres of knowledge* to address this problem. Spheres of knowledge create views of the underlying multidatabases to integrate data from diverse sources and attempt to provide a consistent view of that data to the end-user. The database administrator (DBA) supplies the *meta-data* necessary to integrate conflicting instances into the view, but sometimes the system can only notify the user of a discrepancy when not enough knowledge or information is available for the integration to occur automatically.

The work of [Wang and Madnick, 1989] also proposes a solution to the problem. Their principal contribution is the idea of letting the user write a set of knowledge-based rules that define when two instances from different databases represent the same entity object. The rules are also used to infer new information about separate instances. In this thesis proposal, we take the same approach: we let the user define an *equational theory*, which will be represented in a rule-based language, to identify instances from several databases that are deemed equivalent. [Wang and Madnick, 1989] apply the user-defined rules to the all the input databases to find the desired instances. Since the time complexity of a typical rule-based inference process is at least $O(|WM|^c)$, where c is the maximum number of relations considered on the left-hand side of any rule, and $|WM|$ is the total number of tuples the rule must consider, this idea will only work with small databases, possibly memory-based databases only. In this thesis proposal, we extend this line of work to address the problem of executing this kind of inference with very large databases as input where quadratic time comparison operations are infeasible.

The use of equational theories or knowledge-intensive matching procedures is not a new idea. The first reference to this idea comes from the context of *theorem proving* by *resolution* in [Harrison and Rubin, 1978]. In that paper, Harrison and Rubin generalize the usual unification procedure allowing the specification of "equality predicates". In a more recent example, [Tsur, 1991] introduced this idea in the context of *deductive database*. In particular, he points out that in *Scientific Databases* the data accumulated can contain imprecisions. These uncertainties must be taken into account when data is used for query evaluation and therefore he postulates the need of a domain-specific theory of equality, expressed as a rule set over the stored data. In both of these cases the basic idea is that two complementary first-order literals are unifiable if their constituent terms are provably equivalent even though they are not syntactically equivalent or unifiable.

2.2 Sorting with Duplicate Elimination

Removing duplicates from a sequence of data is an important problem for the database community. Take for example the standard definition of a **relation** under *Relational Databases* [Codd, 1970]. A relation is defined as a *set* of tuples. Tuples, which can be considered a set of $(attr_i, value_i)$ pairs, where $value_i$ is a value from the defined domain of the relation attribute identified by $attr_i$, are therefore unique within a relation. Moreover, in the classic definition of the *project* relational operator, duplicates in the projected relation are expected to be removed from the resulting relation to be valid.

Lower bounds for sorting *multisets* where studied by [Munro and Spira, 1976]. They showed that the multiplicities (i.e., duplicates) of a set can only be obtained by comparisons if the total order is discovered in the process.

Later [Bitton and DeWitt, 1983] studied the problem of duplicate elimination in the context of large data files. They first present the "traditional" algorithm for duplicate elimination consisting of a complete sort of the file followed by a scan to remove duplicates. Our initial approach for solving merge/purge will resemble this "traditional" approach. We will sort the data first, and then search "duplicates" in one scan over the resulting sorted data. The main differences between [Bitton and DeWitt, 1983]'s approach and the approach we propose here are the use of a "window of records" to limit the number of possible duplicates we will consider, and the use of a user-specified, knowledge-based, equality theory to determine if tuples are indeed "duplicates". [Bitton and DeWitt, 1983] then modify the traditional approach to allow duplicate elimination during different stages of the sorting procedure. Through some cost-analysis and models, they show the modified duplicate elimination algorithm to be superior than the "traditional" approach. We will also modify our initial algorithm to allow "duplicate detection" during several phases of the sorting algorithms. A complete description of these algorithms is presented in section 3.1.

2.3 Band-Joins

In the previous subsection we described the merge/purge problem and a solution based upon sorting with duplicate elimination. We can also describe the merge/purge problem as a *Join* operation as follows. Without loss of generality, assume we are interested in identifying "similar" tuples from two relations R and S. Both relations do not need to have the same schema. Let $Attrs(R)$ represent the set of attributes from relation R and $Attrs(S)$ represent the set of attributes from relation S. Depending on the real-world entities represented in the relations, an equivalence function, θ, must be supplied. In the introduction we defined θ as a boolean function a boolean function $\mathcal{F}(\mathcal{X})$, where $\mathcal{X} \leftarrow \{\mathcal{B} \subseteq Attrs(R) \cup \mathcal{C} \subseteq Attrs(S)\}$. The function $\mathcal{F}$ could be a simple arithmetic predicate or a complex inference procedure. Then, to identify similar instances, we execute the query $R \bowtie_\theta S$. The resulting relation will pair every tuple in R with all tuples in S that satisfy θ.

The fast execution of complex select-project-join (S-P-J) queries has received considerable attention from many researchers over the past 20 years. As a result of this effort, a number of algorithms to perform the costly *Join* operation have been proposed (see [Mishra and Eich, 1992]). Currently, three basic algorithms dominate commercial database implementations: *nested-loop-joins* (the "naive" algorithm), *sort-merge-joins*, and *hash-joins* [Bratbergsengen, 1984], with many variants of each.

More recently, attention has shifted towards the efficient parallel execution of S-P-J queries (e.g., [Schneider and DeWitt, 1989]). Over the past 5 years, researchers have addressed various problems related to the parallel execution of S-P-J queries like, data partition among participating processors [Copeland *et al.*, 1988; Kitsuregawa and Ogawa, 1990; Ghandeharizadeh, 1990], skew handling [Schneider and DeWitt, 1990; Wolf *et al.*, 1991; DeWitt *et al.*, 1992], and load-balancing [Hua and Lee, 1990]. All previous mentioned works assume a homogeneous set of processing sites. Recent work by ourselves [Dewan *et al.*, 1994] and by others [Lu and Tan, 1994] have addressed these problems in the context of load balancing protocols to deal with possible heterogeneity among processing sites. However, almost all work in the parallel execution of S-P-J queries has concentrated on hash-join algorithms, and only one type of join queries, namely, *equijoins*. The hard problem of *non-equijoin* queries has been virtually ignored.

As we mentioned above, merge/purge can be thought as a relational join among two or more tables using a special function that determines equality. In section 3.3 we will show these equality functions can be very complex and may require a knowledge-intensive inference to return an answer. Consider the following very simple example: relations R and S contain laboratory results and we want to find those results for which the "temperature" fields are "about the same" (± 5 degrees). Thus, we can define θ as ($\texttt{S.temp}-5 \leq \texttt{R.temp and R.temp} \leq \texttt{S.temp}+5$). Algorithms for executing this kind of *non-equijoin* predicate have been presented by [DeWitt *et al.*, 1991]. [DeWitt *et al.*, 1991] call joins in which the join-predicate has the form $R.A - c_1 \leq S.B \leq R.A + c_2$, *band-joins*. Their paper presents a new algorithm termed a *partitioned band join* to evaluate these special type of joins. The partitioned band join works by first *range partitioning* one of the relations (R) into partitions such that every partition R_i fits entirely on a memory buffer. The second relation is also range partitioned using the same elements as in R and replicating some tuples among "neighboring" partitions. For example, if value x_{i+1} divides buckets R_i from R_{i+1}, then values within $[x_{i+1} - c_1, x_{i+1} + c_2]$ are put in both partitions, S_i and S_{i+1}, of S. To compute the band join, each partition R_i is brought into memory in turn. For each partition R_i, a *window of pages* of partition S_i is read into memory. The join is then computed among the pages of R_i and S_i currently in memory. If the last tuple in the current window of pages of S_i is used, the next page of S_i

is read into the window and the first page of the window is discarded. Similarly, when the last tuple of R_i is used, the next partition, R_{i+1}, is read into memory. It is assumed that the range of the "band" will never exceed the range of values in the window of pages of S_i.

The initial solution we propose for the merge/purge problem uses a similar idea behind the partitioned band join to identify "matching" tuples. The main differences are:

1. As mentioned above and discussed in section 3.3, we provide the use of a more complex equational theory to determine equivalence among the items being considered. When this equational theory is a simple mathematical predicate like the one discussed above, our solution reduces to DeWitt's partitioned band join.

2. DeWitt's band join assumes that data is easily compared by numeric predicates and the outcome of the comparison is categorical. We cannot assume this to be true in all cases when equivalence is a complex inference process. As we will discuss in section 3.3 and demonstrate through experimental results, the equational theory cannot always capture all tuples that are indeed equivalent, and can in some instances incorrectly identify tuples as equivalent (false positives). Thus, we aim to find algorithms that are not only efficient, but also provides us with *accurate* results.

3. Comparing tuples x, y, and z may result in the determination that $x \equiv y$ and $y \equiv z$, but x's relationship to z remains unknown if x and z do not concurrently occupy the same "band". We therefore provide a *multipass* approach followed by the use of a transitive closure phase to increase accuracy.

4. We propose an approach whereby the original relations are first partitioned into a number of disjoint subsets by a variety of clustering methods (not exclusively by range partitioning) to which the solution methods can be applied as independent parallel processes. Note, each disjoint subset has reduced the complexity simply because the size of the data set is reduced.

Since the sorted-neighborhood method reduces to a partitioned band join, the sorted-neighborhood method is a **generalization of band joins**. Later we discuss the parallelization of this technique and the tradeoffs of the alternative methods of partitioning data that participates in the comparison operations.

3 The Merge/Purge Problem

Here we detail the problem we study using a familiar case of name matching in multiple databases. Since we are faced with the task of merging very large databases, we presume a pure quadratic time process (i.e., comparing each pair of records) is infeasible under strict time constraints and limited capital budgets for hardware. For pedagogical reasons we assume that each record of the database represents information about "employees" and thus contains fields for a social security number, a name, and an address and other significant

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

information that may be utilized in determining equivalence. Numerous errors in the contents of the records are possible, and frequently encountered. For example, names are routinely misspelled, parts are missing, salutations are at times included as well as nicknames in the same field. In addition, our employees may move or marry thus increasing the variability of their associated records. Table 1 displays records with such errors that may commonly be found in mailing lists for junk mail, for example. (Indeed, poor implementations of the merge/purge task by commercial organizations typically lead to several pieces of the same junk mail being mailed at obviously greater expense to the same household, as nearly everyone has experienced.)

There are two fundamental problems with performing merge/purge. First, the size of the data sets involved may be so large that only a relatively small portion of the total available data can reside in main memory at any point in time. Thus, the total database primarily resides on external store and any algorithm employed must be efficient, requiring as few passes over the data set as possible.

Second, the incoming new data is corrupted, either purposefully or accidentally, and thus the identification of matching data requires complex tests to identify matching data. The inference that two data items represent the same domain entity may depend upon considerable statistical, logical and empirical knowledge of the task domain. "Faulty" inferences can be in some cases worse than missing some matching data. The "accuracy" of the result (maximizing the number of correct matches while minimizing the number of false positives) is therefore of paramount importance.[1] It is common that much of the engineering of a merge/purge process is devoted to experiment and comparative evaluation of the accuracy of the overall process, and in particular alternative criteria for matching records.

[1]In credit scoring applications, numerous complaints from many individuals who have been incorrectly identified as other persons with similar identities but with poor credit histories have lead to Congressional hearings forcing credit bureaus to significantly re-architect their credit reporting systems.

3.1 The Sorted Neighborhood Method

We consider two approaches to obtaining efficient execution of any solution: partition the data to reduce the combinatorics of matching large data sets, and utilize parallel processing. We require a means of effectively partitioning the data set in such a way as to restrict our attention to a number of small sets of candidates for matching. Consequently, we can process the candidate sets in parallel. Furthermore, if the candidate sets can be restricted to a very small subset of the data, quadratic time algorithms applied to each candidate set may indeed be feasible in the allotted time frame for processing, leading to perhaps better accuracy of the merge task. The rest of this section will concentrate on algorithms for partitioning the data set. The parallel execution of the algorithms will be discussed later in section 5.

One obvious method for bringing matching records close together is sorting the records over the most important discriminating key attribute of the data. After the sort, the comparison of records is then restricted to a small neighborhood within the sorted list. We call this method *the sorted-neighborhood method*. The effectiveness of this approach is based on the quality of the chosen keys used in the sort. Poorly chosen keys will result in a poor quality merge, i.e., data that should be merged will be spread out far apart after the sort and hence will not be discovered. Keys should be chosen so that the attributes with the most discriminatory power should be the principal field inspected during the sort. This means that similar and matching records should have nearly equal key values. However, since we assume the data is corrupted and keys are extracted directly from the data, then the keys will also be corrupted. Thus, we may expect that a substantial number of matching records will not be caught. Our experimental results, presented in section 4, demonstrate this to be the case.

The sorted-neighborhood method resembles a merge-sort in which we are interested in removing duplicates. Two approaches for duplicate elimination using merge-sort were described in [Bitton and DeWitt, 1983]. The first approach, called the “traditional” approach (which we will call the “naive” approach), is the following: first, the file is sorted using an external merge-sort algorithm. Then duplicate records are removed in one sequential scan of the sorted database.

To understand the second approach for duplicate elimination, we must first remember how an external merge-sort works. An external merge-sort is a two phase process: a sorting phase and a merging phase. During the sorting phase, the database is read in pieces that fit into memory. Each piece is sorted in memory using a fast algorithm like, for e.g., quicksort, and then stored on disk as a *sorted run*. During the merging phase, the first pages of n sorted runs are read into memory and an *n-way* merge of their tuples is performed. The next page of a sorted run is read into memory every time its last tuple is consumed. The sorted result is stored in disk as one more “sorted run”. This procedure is repeated until there is only one sorted run remaining: the resulting sorted file. The second approach for duplicate elimination takes advantage of the fact that duplicate records will come together during the sort and the merge phases and thus can be eliminated at each phase. During the sorting phase, after a piece of the input database is sorted in memory, duplicates can be removed as

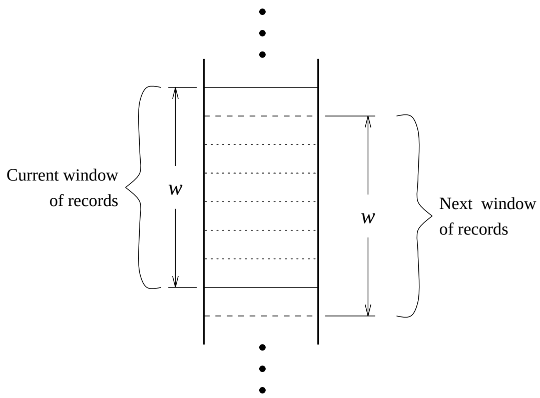


Figure 1: Window Scan during the Merge Phase

the sorted run is written to disk. Then, duplicates occurring in different sorted runs can be eliminated during the n-way merge phase.

Although sorting the data may not be the dominant cost of merge/purge, we consider here an alternative to sorting based upon first partitioning the dataset into independent clusters using a key extracted from the data. Observe that we do not need a completely sorted database, but rather we desire a means of partitioning the data into independent subsets of data in such a fashion that we are assured as much as possible that matching records appear in each cluster. We then apply the sorted-neighborhood method to each individual cluster independently and in parallel ideally as a main-memory based process. We call this approach the *clustering method*.

In the rest of this section we describe in detail the three approaches of the sorted-neighborhood method we have introduced. We first describe the "naive" version of the sorted-neighborhood method followed by a description of its "duplicate elimination" counterpart. We then describe how data could be clustered before the application of any of the two sorted-neighborhood versions.

3.1.1 The Naive 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 may be viewed as knowledge intensive and the effectiveness of the sorted-neighborhood method highly depends on it. We discuss the effect of the choice of the key in section 3.4.

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.

Note, however, that for very large databases the dominant cost could be disk I/O, i.e., 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 [Nyberg *et al.*, 1994], 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, the last pass may indeed be the dominant cost. Later we consider the means of improving this phase by processing "parallel windows" in the sorted list.

We note with interest that the sorts of optimizations detailed in the AlphaSort paper [Nyberg *et al.*, 1994] may of course be fruitfully applied here. In this proposal we are more concerned with alternative process architectures that lead to higher accuracies in the computed results while also reducing time complexity. Thus, in this proposal we consider alternative metrics for the purposes of merge/purge to include how *accurately* can you merge/purge for a fixed dollar and given time constraint, rather than the specific cost- and time-based metrics proposed in [Nyberg *et al.*, 1994].

3.1.2 The Duplicate Elimination Sorted-Neighborhood Method

Similar to the "naive" algorithm, given a collection of two or more databases, we first concatenate them into one sequential list of N records. The "duplicate elimination" sorted-neighborhood method for solving the merge/purge problem works as follows (see figure 2):

1. **Create Keys**: Compute a key for each record in the list by extracting relevant fields or portions of fields.

2. **Sort Data Eliminating Duplicates** : Merge-Sort the records in the data list using the key of step 1 and dividing the sorted output into two lists. In the first list, deposit all records for which duplicate keys are detected. All other records participate in the merge-sort and end in the final sorted list which we will call the "no-duplicates sorted

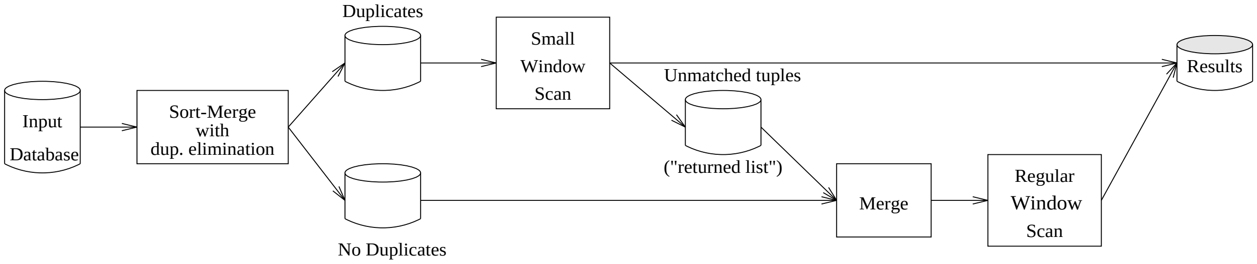


Figure 2: Duplicate Elimination Sorted Neighborhood Method

list". More formally, if $\mathcal{I}$ is the *multiset* of all tuples in the input database, and $\mathcal{D}$ is the multiset of all tuples put in the "duplicates list", and $\mathcal{N}$ is the multiset of all tuples put in the "no-duplicates list", then $\mathcal{I} = \mathcal{D} \cup \mathcal{N}$ and $\mathcal{D} \cap \mathcal{N} = \emptyset$.

3. **Sort the duplicate list** : Notice that even though the "duplicates list" is generated incrementally during the merge-sort, it is not completely sorted. Since duplicates of a record can be found during the sort and the merge phases of the merge-sort, these duplicates may not be in order. Thus, before proceeding to the next step, we sort the duplicate list using the key.

4. **First Window Scan**: Move a "small" window through the sequential list of duplicate records limiting the comparisons to those records in the window *having the same key.* Let ψ be the size of the small window used in this step. Every record that is about to enter this window either has the same key as all other records already in the window or its key is different than the key of all records in the window. If the key of the new record is equal to the key of the records in the window, then this new record is compared with the $\psi - 1$ previous records in the window to find "matching" records. As with the **Merge** phase in the naive method, "matching" records are determined by the equational theory. The first record in the window slides out of the window. On the other hand, if the key of the current record is not equal to the key of the records already in the window, then the following steps are followed:

   (a) Append to the "returned list" of records, those records in the window that where not "matched" with any other record.

   (b) Also append to the "returned list" the record that was matched the most (and at least once) with other records in the window. This record will become the "prime representative" of its key in the following step.

   (c) The window is moved $\psi - 1$ positions making the new record the first one in the window.

5. **Merge**: Merge the "returned list" of records with the records in the "no-duplicates sorted list". Since both list are already sorted, a simple 2-way merge is sufficient.

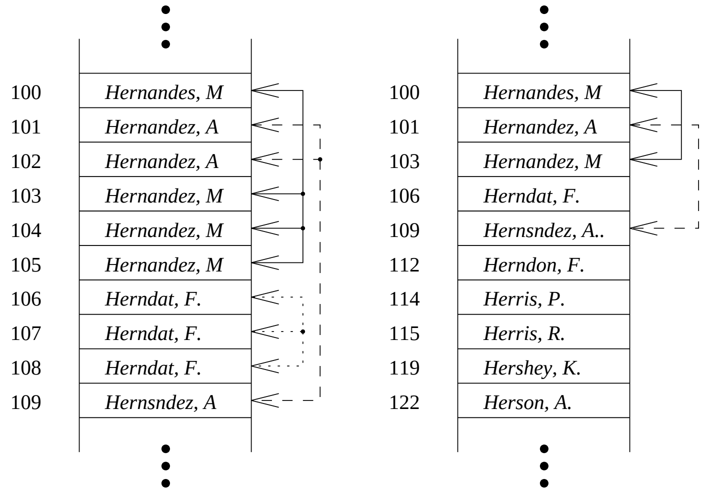


Figure 3: Example of the Duplicate Elimination Sorted Neighborhood Method

However, an extra bit-field is added to the resulting sorted data to indicate whether a record came from the "returned list" or the "no-duplicates" list.

6. **Second Window Scan**: Move a fixed sized window through the sequential list of records produced in the previous step 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. Notice that if the record entering the window came from the "returned list" (detected using the extra bit-field), then it should only be compared with records that *did not* come from the "returned list". The reason for this is that those records were already compared (via the equational theory) during the **first window scan** step of this procedure and were found to be "non-matching". On the other hand a record coming from the "no-duplicates" list should be compared with all $w-1$ previous records for it has not been compared to any record before. Similarly to the naive sorted-neighborhood method, the first record in the window slides out when a new record enters the window.

Figure 3 shows a simple example of how the "duplicate elimination" version of the sorted-neighborhood method works. The left-hand side of the figure shows a sequence of tuples after being sorted by the name field. The arrows to the right of the sequence show the tuples that should be merged. If this sequence of tuples were sorted with the algorithm described on step 2 of the "duplicate elimination" algorithm, tuples $\{101, 102, 103, 104, 105, 106, 107, 108\}$ would end up in the "duplicates list" while tuples {100,109} would end in the "no-duplicates list". After the **first window scan** (step 4), assuming a perfect implementation of the equational theory, pairs $(101, 102)$, $(103, 104)$, $(103, 105)$, $(106, 107)$, and $(106, 108)$ would be detected as "similar" and reported as merged. At the same time, a representative of each key on the "duplicates list" would be put in the "returned list". In the case of this example tuples $101, 103, 106$ are placed in the "returned list". The right-hand side of figure 3 shows the sorted sequence of tuples after the tuples in the "returned list" are merged with the tuples in "no-duplicates" list (step 5). Now the **second window scan** is applied to this sequence, detecting the two pair of tuples that should be merged not captured during the **first window scan**, namely, pairs $(100, 103)$ and $(101, 109)$. Notice how the size of the window needed to detect the similar records on the **second window scan** is smaller than the window needed if we use the "naive" sorted-neighborhood method on the original sorted sequence of tuples. In particular, to merge tuples 101 and 109 using the "naive" version, we would need to slide a 9 record wide window down the sorted sequence. To merge the same tuples using the "duplicate elimination" version of the algorithm, a window of size 4 is needed over the sorted sequence in the right-hand side of figure 3.

When this procedure is executed serially, initially sorting and removing the duplicates is an $O(N \log N)$ operation and the first window scan is $O(\psi f N)$, where f is the ratio of records with duplicates in the input database, $0 < f < 1$. During this step, a number of tuples in the "duplicates list" are put into the "returned list". Assuming g is the ratio of tuples in the "duplicates list" put into the "returned list", $0 < g < 1$, then the cost of the merge step is $O(gfN + (1-f)N)$, where $(1-f)N$ is the size of the "no-duplicates list". The cost of the second window scan is then $O(wgfN + w(1-f)N)$.

Using this simple time analysis, we can try to predict under what conditions will the "duplicate elimination" version of the sorted-neighborhood method be better than its "naive" counterpart. We can approximate the time to execute the "naive" version as follows:

$$T_{naive} = c_s N \log N + c_{ws} w N$$

where c_s and c_{ws} are the computational constants for the sort and window scan phases respectively. Similarly, the time to execute the "duplicates elimination" version is:

$$T_{de} = c_s N \log N + c_s f N \log f N + c_{ws} \psi f N + c_m (gfN + (1-f)N) + c_{ws} w (gfN + (1-f)N)$$

where, c_s and c_{ws} have the same meaning as in the "naive" formula, and c_m represents the constant for the merge in step 5 of the "duplicate elimination" algorithm. Solving the equation $T_{naive} > T_{de}$, we found

$$w > \left(\frac{f}{f - gf}\right)\psi + \left(\frac{c_s}{c_{ws}}\right) f \log f N + \left(\frac{c_m}{c_{ws}}\right)\left(\frac{1 - f + gf}{f - gf}\right)$$

We expect $c_{ws} > c_s > c_m$. Thus we can assume $\frac{c_s}{c_{ws}} < 1$ and $\frac{c_m}{c_{ws}} < 1$. The factor $\left(\frac{f}{f-gf}\right)$ is close to 1 if g is closer to 0 than it is to 1. In fact, as $g \to 1$, meaning that most of the work in the **first window scan** phase was useless since most of the tuples from the "duplicate list" were put into the "returned list", this factor tends to infinity. In the same vein, the factor also tends to infinity as $f \to 0$, meaning the number of duplicates in the data is small. Thus, the "duplicate elimination" should work fine if there is a considerable number of duplicates in the data. We expect $g \leq \frac{1}{2}$ and $f > \frac{1}{4}$. Thus, we can assume this factor to be close to 1. The last factor, $\left(\frac{1-f+gf}{f-gf}\right)$, behaves similarly to the previous one as $g \to 1$ and $f \to 0$. But under the assumption $g \leq \frac{1}{2}$, $f > \frac{1}{4}$, the value of this factor is also close to 1 (or at most a small integer). Taken all together, we can then say that the "duplicate elimination" version of the sorted-neighborhood method should do better in time than the "naive" version when:

$$w > \psi + C$$

where $C > 0$ is a small integer number. We show this model to be close to reality with some experiments in section 4.4.

3.2 Clustering the data first

Given a group of two or more databases, we first concatenate them into one sequential list of N records. The clustering method can be summarized as the following two phase process:

1. **Cluster Data**: We scan the records in sequence and for each record we extract an n-attribute key and map it into an n-dimensional cluster space. For instance, the first three letters of the last name could be mapped into a 3D cluster space from our employee database example.

2. **Sorted-Neighborhood Method**: We now apply the sorted-neighborhood method independently on each cluster. We do not need, however, to recompute a key (step 1 of the sorted-neighborhood method). We can use the key extracted above for sorting.

When this procedure is executed serially, the cluster data phase is an $O(N)$ operation, and assuming we partition the data into C equal sized clusters, the sorted-neighborhood phase is $O(N \log \frac{N}{C})$.

Clustering data as described above raises the issue of how well partitioned the data is after clustering. We use an approach that closely resembles the multidimensional partitioning strategy of [Ghandeharizadeh, 1990]. If the data from which the n-attribute key is extracted is distributed uniformly over its domain, then we can expect all clusters to have approximately the same number of records in them. But real-world data is very unlikely to be uniformly distributed, i.e. skew elements and other hot spots will be prevalent, and thus, we must expect to compute very large clusters and some empty clusters.

Sometimes the distribution of some fields in the data is known, or can be computed as the data is inserted into the database. For instance, consider a database containing a field for names. We can find lists of person names from which we can compute, say, the distribution of the first three letters of every name[2]. If we do not have access to such a list, we can randomly sample the name field of our database tables to have an approximation of

[2] That is, we have a cluster space of $27 \times 27 \times 27$ bins (26 letters plus the space).

the distribution of the first three letters. This information can be gathered off-line before applying the clustering method.

Now let us assume we want to divide our data into C clusters using a key extracted from a particular field. Given a frequency distribution histogram with B bins for that field ($C \leq B$), we want to divide those B bins (each bin represents a particular range of the field domain) into C subranges. Let b_i be the normalized frequency for bin i of the histogram ($\sum_{i=1}^{B} b_i = 1$). Then for each of the C subranges we must expect the sum of the frequencies over the subrange to be close to $\frac{1}{C}$ (e.g., if bins s to e, $1 \leq s \leq e \leq B$, are assigned to one cluster then we expect $\sum_{i=s}^{e} b_i \simeq \frac{1}{C}$). Each subrange will become one of our clusters and, given a record, we extract the key from the selected field, and map the key into the corresponding subrange of the histogram. The complexity of this mapping is, at worst, $\log B$.

3.2.1 Alternative Clustering Strategies

In the previous section we discussed a clustering strategy based on partitioning by the range values of a key. In general, this technique can be applied to several attributes of a relation and is known as *multidimensional partition strategy* (e.g., [Ghandeharizadeh, 1990]). There are, however, other partitioning strategies we might want to provide in a generic merge/purge facility we plan to implement, namely, constant partitioning, uniform partitioning, hash partitioning, and classification.

Constant partitioning is a special case of range partitioning and is used when the range of values of an attribute is a small number of constant values. For example, in a relation containing information about persons physical characteristics, we might want to partition the data by gender or color of the hair.

Uniform partitioning divides the input relation into equal-sized fragments regardless of the value of any attribute. This might be useful when, for example, the equational theory can be applied to a random set of tuples, or when the input relation is already sorted. In section 6 we will discuss the implementation of ALEXSYS, an expert system for mortgage allocation, using our generic merge/purge facility. The implementation of ALEXSYS would benefit if an option for constant partitioning is given.

Hash partitioning uses a hash function to divide the input into P partitions or *buckets*. This partitioning scheme could be useful in cases where the input relation has a key attribute which is known not to be noisy.

Discovery algorithms have been described by [Frawley *et al.*, 1992] as procedures to extract knowledge from data. Two processes are involved in these procedures: interesting patterns must be identified, and a meaningful description of each pattern must be provided. Discovery algorithms can start their identification process by classifying records into classes (or clusters) that reflect patterns inherent in the data[3]. The creation of clusters can involve traditional cluster analysis methods [Dubes and Jain, 1976] or more recent *conceptual clustering* methods which, as the former, use attribute similarity to form clusters, but also take into consideration background knowledge, such as knowledge about likely cluster shapes.

[3]These processes are sometimes referred to as unsupervised and supervised learning.

Some recent examples of this approach include the two Bayesian classifiers in [Cheeseman *et al.*, 1988] and [Anderson and Matessa, 1990]. The output of these classifications algorithms can be thought as a set of selection predicates that divide the data into disjoint clusters.

We can generalize some of these clustering approaches using the following model. Let $d(t_1, t_2)$ define the distance between two tuples t_1, t_2 as follows:

$$d(t_1, t_2) = \mid f(t_1.x) - f(t_2.x) \mid$$

where f is some function defined on the domain of attribute x of the underlying relation. The behavior of this clustering scheme is determined by the choice of the function f and a threshold θ, such that $d(t_1, t_2) \leq \theta$. If f is a hash function, and $\theta = 0$, then this reduces to simple hash partitioning. If f is the identity function and $\theta = 0$, then this reduces to restricting by constant values. If f is the identity function and $\theta = c$, for some non-zero constant c, then this reduces to range partitioning (under appropriate definition of the "end points" t_i). Finally, when f is the "classifier" and $\theta = 0$, this reduces to conceptual classification.

3.3 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 later. 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 merge/purge 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 solving merge/purge efficiently.

As an example, here is a simplified rule in English that exemplifies one axiom of our equational theory relevant to merge/purge applied 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 4 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.

For the purpose of experimental study, we wrote an OPS5 [Forgy, 1981] 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[4].

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 actual rule program uses a couple more rules to initialize the system, detect termination, and computes the transitive closure of the results for the purposes we explain in the next section.

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 joined 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

[4] At the time the system was built, the public domain OPS5 compiler was simply too slow for our experimental purposes. Another compiler, the OPS5C compiler [Miranker *et al.*, 1990] was not available to us in time for these studies. The OPS5C compiler produces code that is reportedly many times faster than previous compilers. We captured this speed advantage for our study here by hand recoding our rules in C.

to see if a merge can be done. Otherwise, in the third stage, more precise "edit-distance" functions are used over some fields as a last attempt for merging a pair of records. Table 1 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.

3.4 Computing the transitive closure over the results of independent runs

The effectiveness of the sorted-neighborhood method highly depends on the key selected to sort the records. A key is defined to be a sequence of a subset of attributes, or substrings within the attributes, chosen from the record. For example, we may choose a key as the last name of the employee record, followed by the first non blank character of the first name sub-field followed by the first six digits of the social security field, and so forth.

In general, no single key will be sufficient to catch all matching records. Attributes that appear first in the key have a higher priority than those appearing after them. 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.

It is clear that the utility of this approach is therefore driven by the nature and occurrences of the errors appearing in the data. Once again, 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 prior to running a merge/purge 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.

4 Experimental Results

4.1 Generating the databases

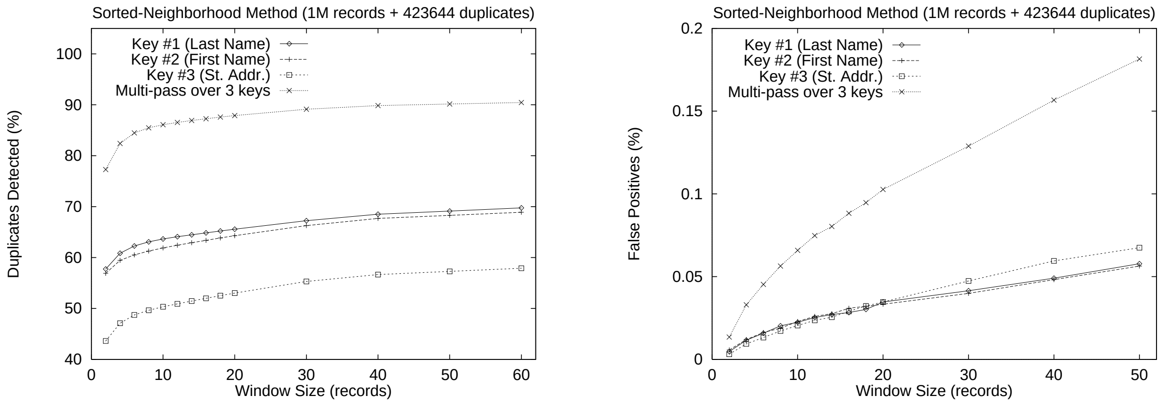
All databases used to test the sorted-neighborhood method and the clustering method were generated automatically by a database generator that allows us to perform controlled studies and to establish the accuracy of the solution method. This database generator provides a large number of parameters including, 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. 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. The cities, states, and zip codes (all from the U.S.A) come from publicly available lists.

The errors introduced in the duplicate records range from small typographical changes, to complete change of last names and addresses. When setting the parameters for the kind of typographical errors, we used known frequencies from studies in spelling correction algorithms [Pollock and Zamora, 1987; Church and Gale, 1991; Kukich, 1992]. For this study, the generator selected from 10% to 50% of the generated records for duplication with errors, where the error was controlled according to published statistics found for common real world datasets.

4.2 Pre-processing the generated database

Pre-processing and conditioning the records in the database prior to the merge/purge operation might increase the chance of finding two duplicate records [Pu, 1991]. For example, names like `Joseph` and `Giuseppe` match in only three characters, but are the same name in two different languages, English and Italian. A nicknames database or name equivalence database is used to assign a common name to records containing identified nicknames. (Alternatively the nicknames database can be included as one of the merged databases from the start, permitting two records with alternative names to be found by the transitive closure step.)

Since misspellings are introduced by the database generator, we explored the possibility of improving the results by running a spelling correction program over some fields. Spelling correction algorithms have received a large amount of attention for decades [Kukich, 1992]. Most of the spelling correction algorithms we considered use a corpus of correctly spelled words from which the correct spelling is selected. Since we only have a corpus for the names of the cities in the U.S.A. (18670 different names), we only attempted correcting the spelling of the city field. We chose the algorithm described by Bickel in [Bickel, 1987] for its simplicity and speed. Although not shown in the results presented in this proposal, the use of spell corrector over the city field improved the percent of correctly found duplicated records by only 1.5% - 2.0%. Most of the effort in matching resides in the equational theory rule base.



(a) Percent of correctly detected duplicated pairs

(b) Percent of incorrectly detected duplicated pairs

Figure 4: Accuracy results for a 1,000,000 records database

4.3 Initial 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 4(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 4(a) shows our results when the 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. Table 2 shows a sample of actual record-pairs that represent the same real-world entity, but were not detected by the *multi-pass* approach. In most of these cases the records were not deemed "similar" because the

SSN	Name	Street Address	City, State, Zip
None	Brottier Grawey	PO gBox 2761	Clovis NM 88102
None	B baermel	O Box 2761	Clovis NM 88102
None	Chipman H Jianqi	953 Backencamp Rd 2h2	Ukiah OR 97880
None	Jianqi H casperston	923 ackencamp Rd 2h2	Ukiqah OR 97880
None	Bruin U Hochstetlfer	330 Hennrich Lane 0m4	San German PR 00753
None	Bruin U jamamn	330 Hennrich Lane 0m4	Ssan German PR 00753
None	Leitem Z Jeffords	988 Benaz St 0x6	Culebra PR 00645
None	keite Z vasriya	988 Bnez Street 0x6	Cukebra PR 00645
None	Avra N Dadas	PO Box 3965	Hillsboro NM 88042
None	Bavra N ada	381 Ladeau St 4q6	nillsboro NM 88042
152014425	Bahadir T Bihsya	220 Jubin 8s3	Toledo OH 43619
152014423	Bishya T ulik	318 Arpin St 1p2	Toledo OH 43619

Table 2: Example of record-pairs missed

error introduced into the key fields (first name, last name, and street address) will place the records too far apart in the sorted databases to ever be considered during window-scanning phase. The only case where this observation is not true is the third example in table 2. Here our equational theory did not make the decision of marking the records similar even though the first names and addresses are the same. Our equational theory needed at least one more small hint to actually make the positive decision, for example, a valid social security number or some similarity in the last name. This last decision is application dependent. Relaxing the equational theory to admit this particular record as "similar" would probably increase the number of false-positives detected.

As mentioned above, our equational theory is not completely trustworthy. It can mark two records as similar when they are not the same real-world entity (false-positives). Figure 4(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

SSN	Name	Street Address	City, State, Zip
274158217	Frankie Y Gittler	PO Box 3628	Gresham, OR 97080
267415817	Erland W Giudici	PO Box 2664	Walton, OR 97490
738424074	Langham Inukai	552 Boecke St 3o8	San Juan, PR 00926
384250742	Drobnik B Cuschera	238 Boedecker St 3p0	Sweetwater, OK 73666
760652621	Arseneau N Brought	949 Corson Ave 5l5	Blanco, NM 87412
765625631	Bogner A Kuxhausen	212 Corson Road 0o3	Raton, NM 87740
283135241	Ballard Anspach	PO Box 105	Custer City, OK 73639
128317524	Lemmo L Lesway	PO Box 1057	Konawa, OK 74849
257657453	Dialout I Alvarado	PO Box 243	Cave Junction, OR 97523
586574530	Cordelie T Latrina	PO Box 2437	Robertsville, OH 44670
330258754	Ballarte Fortner	PO Box 2084	Albuquerque, NM 87185
338025275	Benham Y Jobes	PO Box 2083	Cayey, PR 00737

Table 3: False-positives reported by the rule program

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!

Table 3 shows a sample of record-pairs marked as "similar" by the equational theory even though they do not represent the same real-world entity. In all these cases, distances computed were very close to the thresholds used by the equational theory in discriminating similar data. We also notice that our equational theory does not have rules to handle the special case of P.O. Boxes. For example, in the last case in table 3, our equational theory infers that the addresses are "close" based only on the small edit-distance between "`PO Box 2084`" and "`PO Box 2083`". A better equational theory would consider other parts of the address (e.g., the city name) if it detects a P.O. Box that appears as the "street address".

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 solutions to compute transitive closure exist [Agrawal and Jagadish, 1988]. 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 address this issue in section 5.

4.4 The Duplicate Elimination Method

We ran a series of experiments to evaluate the performance of the "duplicate elimination" sorted-neighborhood method and compared it to the performance of the "naive" counterpart. For the three experiments reported in this section, we started with the same database of 100,000 records and then allowed the generator to select 35% of the tuples for duplication with modifications. In each experiment, the maximum number of times a selected record can be duplicated varied (5, 10, and 15 respectively). Three runs of each sorted-neighborhood method (naive and duplicate elimination) were executed, each run used a different key. The results of the three independent runs were then processed with a transitive closure phase to improve the accuracy of the results. The size of the window for both the "naive" and the "duplicate elimination" methods varied in the range $[2, 10]$. The size of the "small" window for the special **first window scan** phase of the "duplicate elimination" method is $\psi = 3$. The results of the experiments are shown in figures 5, 6, and 7. *Key 0* is the last name followed by the first name, *Key 1* is the first name followed by the last name, and *Key 2* is the street address followed by the last name field.

First, notice that the accuracy of the "duplicate elimination" method closely follows the accuracy of the "naive" method. In fact, although hard to see from the figures, in all cases tested in this series of experiments, the accuracy of the "duplicate elimination" method edged slightly higher than the accuracy of its counterpart. The reason for this small, but consistent better performance of the "duplicate elimination" method can be understood by looking at

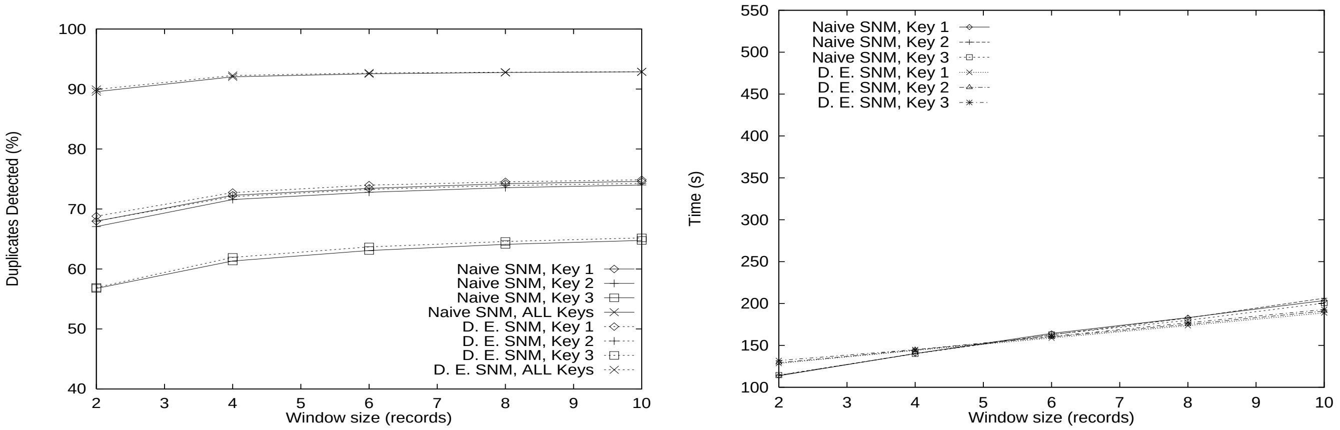


Figure 5: Duplicate Elimination vs. Naive (100,000 records, max 5 dups/record)

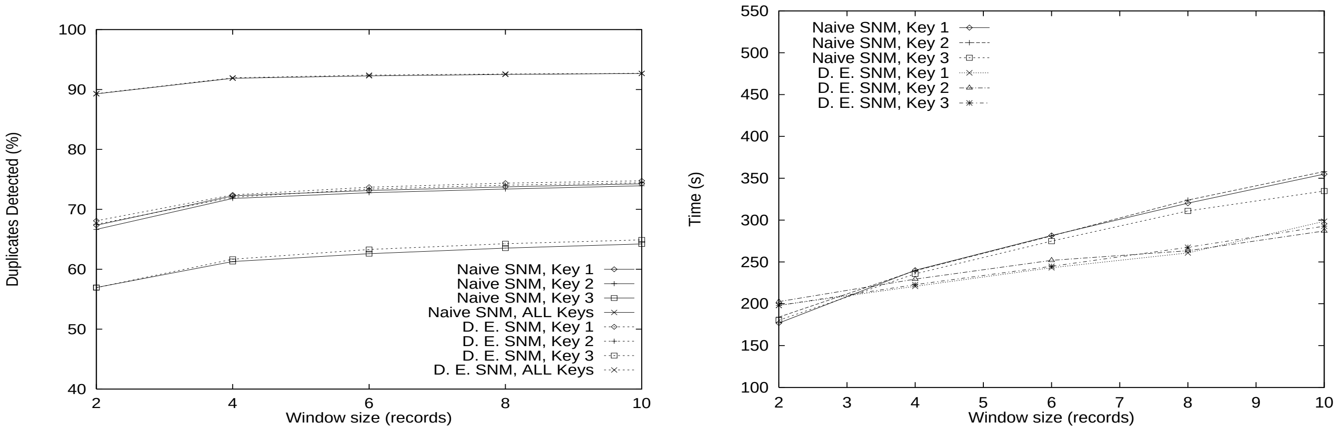


Figure 6: Duplicate Elimination vs. Naive (100,000 records, max 10 dups/record)

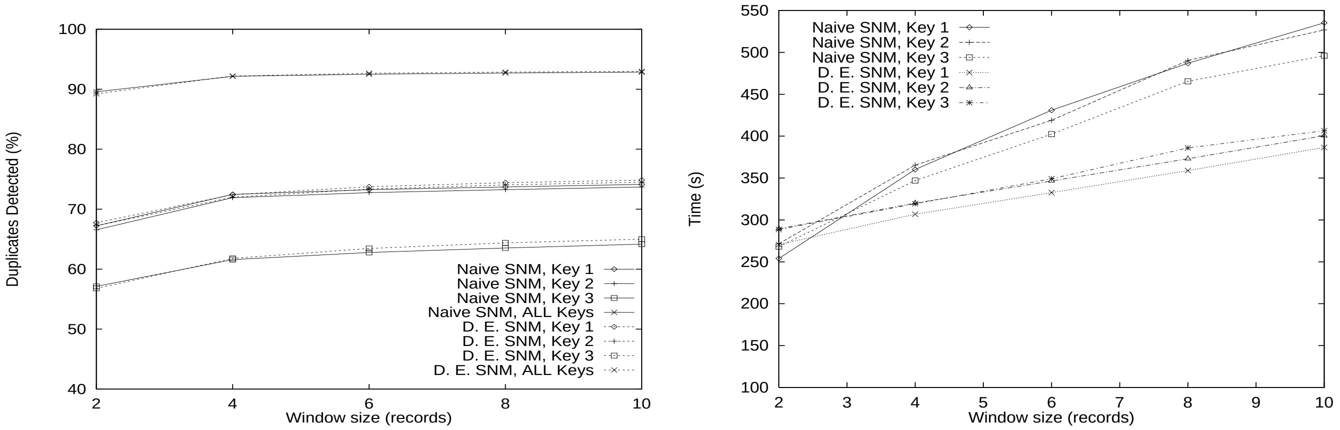


Figure 7: Duplicate Elimination vs. Naive (100,000 records, max 15 dups/record)

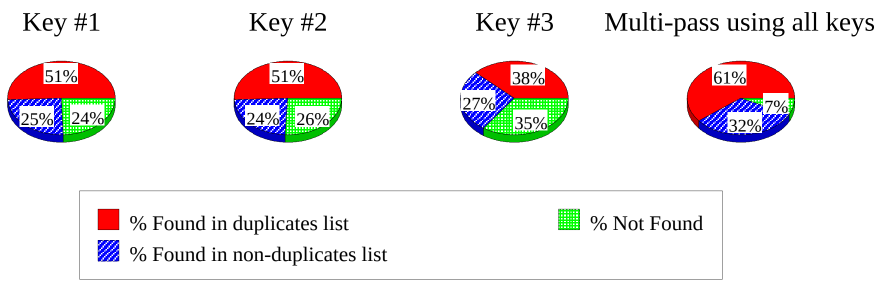


Figure 8: Percent of records identified as "similar" by the "duplicate elimination" algorithm

figure 3 again. Notice that the **first window scan** finds a good number of duplicates, thus removing a considerable number of records from the "middle" of the window used during the **second window scan** phase. Thus, records that should "match" but fell outside the window with the "naive" method, are closer together after removing duplicates and therefore could be "matched" with a window of the same size as the one used unsuccessfully in the "naive" method.

An important question is how many of the tuples identified as "similar" are detected by the **first window scan** phase (window scan of a small window over the duplicates) and how many are detected by the **second window scan** phase (window scan over the "returned" and "no-duplicates" list of sorted records)? The pie charts in figure 8 show the percent of records identified as "similar" at each phase by the "duplicate elimination" method for the database with 5 maximum duplicates per selected record[5]. In all cases, from 58% ($\frac{38}{38+27} \times 100$) to almost 66% of the total number of records detected as "similar", were detected during the **first window scan** phase. Thus, a large part of the "matching" work is being done during the **first window scan**, which uses a smaller window than the one used during the second **second window scan**.

However, even though a large part of the work is being done during the **first window scan**, the duplicate elimination algorithm merges some of the tuples in the "duplicate list" with those in the "no-duplicate list" to perform the "second window scan" phase. In fact, the duplicate elimination algorithm contains more phases than the naive version of the sorted-neighborhood method. Thus, the next question we must answer is whether or not the time performance of the duplicate elimination algorithm is better than that of the naive algorithm.

The right-hand side of figures 5, 6, and 7 show the total time to run each algorithm over different keys for the three experimental databases described before. In all cases, notice that the naive version of the sorted-neighborhood method did better than the duplicate elimination version for small window sizes. Nevertheless, as the size of the window grows (and, thus, the complexity of the window scan phase of the naive approach and the complexity of the **second window scan** phase of the duplicate elimination approach grows as well), the

[5]The charts of the other two experiments were similar and are not presented in this proposal.

duplicate elimination version starts doing better than the naive version. Notice also that, as the number of possible duplicates per record increases, so does the total execution time difference between the duplicate elimination and naive versions. This difference is explained by recalling that a large portion of the duplicates will be matched and removed from further consideration during the **first window scan** of the duplicate elimination version which always uses a constant sized window ψ ($\psi = 3$ on these experiments). After duplicates are eliminated, we are left with a database whose size does not change much for each experiment. Thus, the time to apply the **second window scan** to this database is virtually the same for each experiment (under the same window size). Leaving the **second window scan** window size fixed, the increase in time of the duplicate elimination version is driven by the increased number of duplicates being considered by the small window in the **first window scan**. On the other hand, the increase in time in the naive version is driven by the size of the window, w, and the total size of the input database. It is therefore expected that the duplicate elimination version of the sorted-neighborhood method would have a better time performance than the "naive" version when $w > \psi$. It is also expected that the performance of the two versions should be about the same when $w \approx \psi$. Figures 5-7 clearly show these relations to be true.

From these results we must conclude that the duplicate elimination version of the sorted-neighborhood method is the version of choice for databases where the number of possible duplicates per record is known to be large. In fact, carefully looking at figure 6, we notice that, for example, the time for $w = 6$ for the naive version is almost the same as the time for $w = 10$ for the duplicate elimination version. That is, for the same amount of time, a larger window can be used with the duplicate elimination version providing a larger accuracy than the naive version. On the other hand, when the number of duplicates per records is small, it almost makes no difference which version of the sorted-neighborhood method is used if the window size, w is relatively large ($w > \psi$).

4.5 The Clustering Method

To test the clustering method, we created a database with 250,000 records of which 35% of the records were selected to add a maximum of 5 duplicates per selected record. The resulting 468,730 records database was analyzed using the naive sorted-neighborhood method and the clustering method. We used the same three keys used above for the sorted-neighborhood method and ran three independent runs, one for each key. Then the transitive closure over the results of all independent runs was computed. Each run of the clustering method initially divided the data into 32 clusters. These number of clusters was chosen to match the "fan-out" of the merge-sort algorithm of the "naive" method. This way we guarantee each cluster fits in memory. The results presented in this section were obtained on a Sun's Sparc 5, running SunOS 5.3.

Figure 9 compares the timing results of the "naive" and clustering versions of the sorted-neighborhood method. As expected, the total time to execute each independent run (and, thus, the *multi-pass* approach), is lower when we partitioned the data first into independent

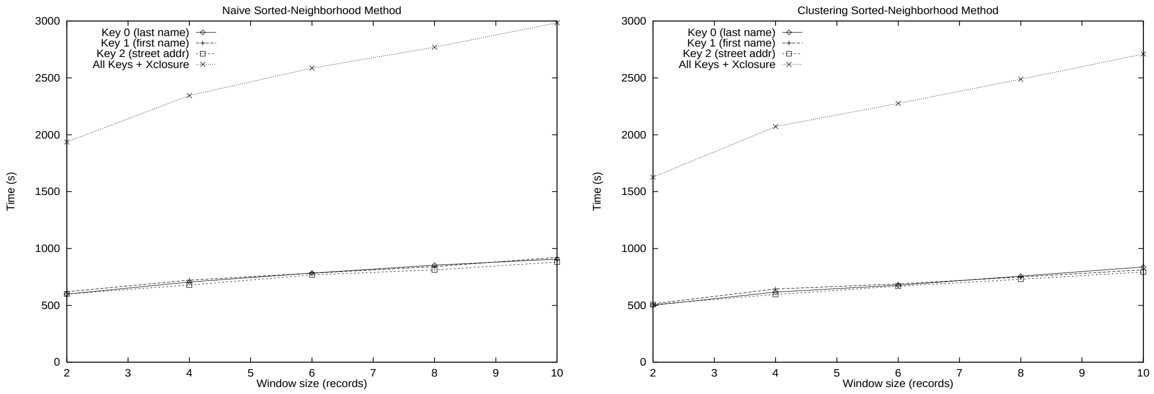


Figure 9: Time Results for the Sorted-Neighborhood and Clustering Methods on 1 processor

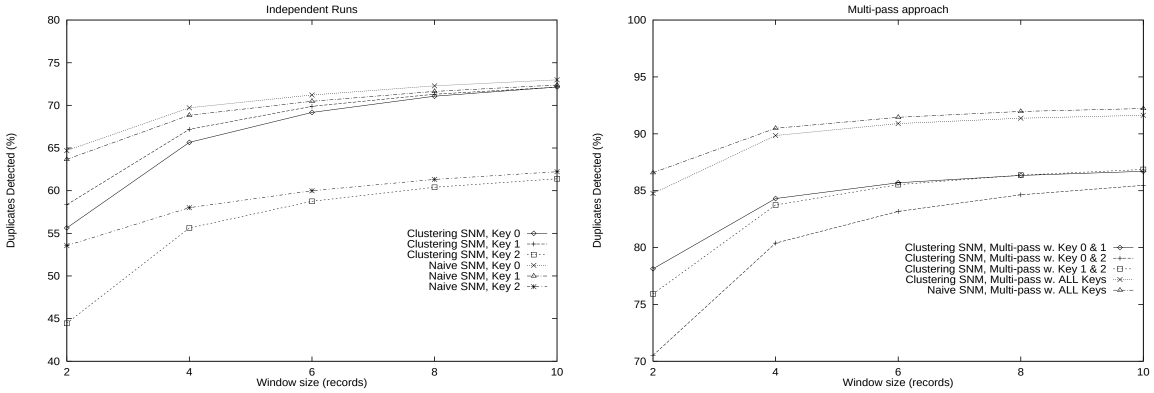


Figure 10: Accuracy of the Clustering vs. Naive Sorted-Neighborhood methods

Window	Run 0: Last Name			Run 1: First Name			Run 2: Street Addr.			Multi-pass
Size	Sort	Merge	Closure	Sort	Merge	Closure	Sort	Merge	Closure	Closure
2	296	261	41	308	270	40	303	259	36	117
4	298	326	79	314	330	76	291	332	54	239
6	306	392	84	306	391	84	308	389	69	252
8	306	459	87	300	455	85	292	448	71	263
10	297	520	89	305	528	87	294	513	72	276

Table 4: Naive Sorted Neighborhood Method Times (250,000 records, 1 SUN4 site)

Window	Run 0: Last Name			Run 1: First Name			Run 2: Street Addr.			Multi-pass
Size	Cluster	Merge	Closure	Cluster	Merge	Closure	Cluster	Merge	Closure	Closure
2	184	280	35	194	281	37	196	279	30	105
4	194	353	69	193	378	71	198	348	48	215
6	187	411	77	198	409	80	199	405	63	241
8	194	481	81	194	472	81	196	467	66	252
10	192	561	83	197	531	83	200	523	69	266

Table 5: Clustering Method Times (250,000 records, 1 SUN4 site)

clusters. Tables 4 and 5 show the actual timing results for each phase of both methods. Compare, in particular, the reduction in time of the clustering phase with respect to the sorting phase of the naive sorted-neighborhood method.

The graphs in figure 10 show the accuracy results of both methods for this experiment[6]. The left-hand side graph shows the accuracy obtained by each independent run of both methods. In all cases the accuracy of the naive sorted-neighborhood edged higher than the accuracy of the clustering method. The principal reason for this is the size of the key used on each run. Even though the field used to produce the key for each independent run was the same under each method, the size of the key was not. As explained in section 3.2, the clustering method uses the fixed-sized key extracted during its clustering phase to later sort each cluster independently. On the other hand, the naive sorted-neighborhood method used the complete length of the strings in the key field, making the size of the key used variable. Records that should be merged are expected to end closer together in a sorted list the larger the size of the key used. Since, for the case of this experiment, the average size of the key used by the naive method is larger that the one used by the clustering method, we must expect the naive method to produce more accurate results.

The right-hand side graph in figure 10 shows how the accuracy improves after the *multi-pass* approach is applied to the independent runs. As we saw in the previous section, when we applied the closure to all pairs found to be similar with the three independent runs, the accuracy jumped to over 90% for $w > 4$. In this graph we also present the accuracy of the clustering method when the multi-pass approach is applied to only two independent runs instead of three. For this particular experiment, under any of the three possible combinations for a two run multi-pass approach, the performance remained near but under 85%.

4.6 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* (for the moment 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),

$$T_{multipass} = c_{sort} rN \log N + c_{wscan} rwN + T_{closure_{mp}}$$

[6]Note the different y scales of both graphs.

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 approach 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.

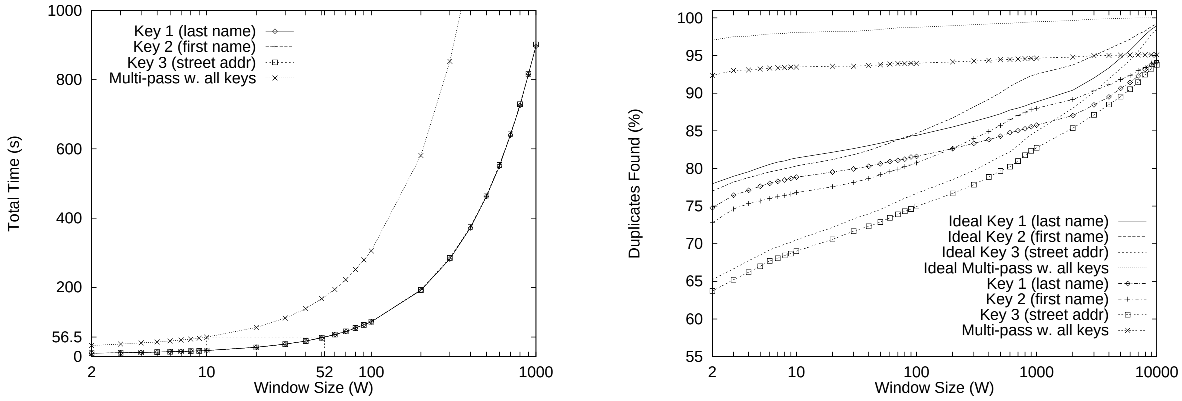
$$cN \log N + \alpha cWN + T_{closure_{sp}} > crN \log N + \alpha crwN + T_{closure_{mp}}$$

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 the particular case were $w = 10$, we have $\alpha \simeq 6$, $c \simeq 1.2 \times 10^{-5}$, $T_{closure_{sp}} = 1.2s$, and $T_{closure_{mp}} = 7$. Thus, the *multi-pass* approach dominates the single sort approach for these datasets when $W > 41$.

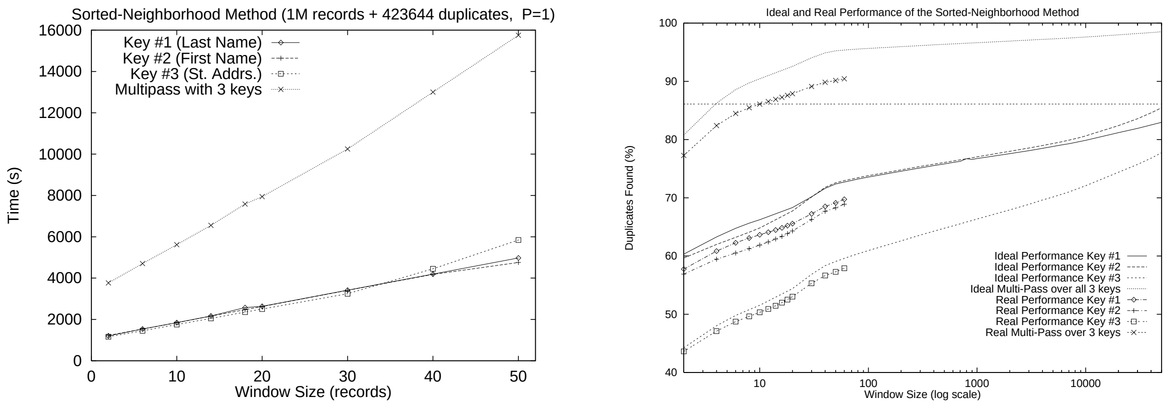
Figure 11(a) shows the time required to run each independent run of the sorted-neighborhood method on one processor, and the total time required for the *multi-pass* approach while figure 11(b) shows the accuracy of each independent run as well as the accuracy of the *multi-pass* approach (please note the logarithm scale). For $w = 10$, figure 11(a) shows that the *multi-pass* approach needed $56.3s$ to produce an accuracy rate of 93.4% (figure 11(b)). Looking now at the times for each single-pass run, their total time is close to $56s$ for $W = 52$, slightly higher than estimated with the above model. But the accuracy of all single-pass runs in figure 11(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 11(a)) their execution time are over 4,800 seconds (80 minutes).



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

(b) Ideal vs. Real Accuracy of each run

Figure 11: Time and Accuracy for a Small Database



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

(b) Ideal vs. Real Accuracy of each run

Figure 12: Time and Accuracy for the 1,000,000 records database

Figure 12(a) shows the time required to run each independent run of the sorted-neighborhood method on one processor, and the total time required for the *multi-pass* approach, in the case of the database we used in section 4.3 (1423644 records). As illustrated in figure 4(a), the *multi-pass* approach produced an accuracy rate of 86.1% using a window size of $w = 10$. Using figure 9 to extrapolate the time performance of a single-pass run, we notice that it is similar to the time performance of the *multi-pass* approach with a small window, $w = 10$, when $W \simeq 56$. But similarly to the previous case with a smaller database, the accuracy of all single-pass runs in figure 4(a), at $W = 50$, are from 57% to 68%, well below the 86.1% accuracy level of the *multi-pass* approach. To study how large the window size W must be for **one** of the single-pass runs to achieve the same accuracy level of the *multi-pass* approach we replaced the rule based equational theory with a stub that quickly tells us if two records within the window are actually equal according to the database generator (thus we study the "ideal" performance). The results, depicted in figure 12(b)[7], show that any single-pass run would need a window size larger than $W = 50000$ to achieve the same accuracy level as the *multi-pass* approach using $w = 10$. Thus, the *multi-pass* approach achieves dramatic improvement in time and accuracy over a single pass approach.

Let us now consider the issue when the process is I/O bound rather than a compute-bound main-memory process. We consider three cases. In the first case, the sorted-neighborhood method, one pass is needed to create keys, $logN$ passes[8]. to globally sort the entire database, and one final pass for the window scanning phase. Thus, approximately $2 + logN$ passes are necessary. In the second case, the clustering method, one pass is needed to assign the records to clusters followed by another pass where each individual cluster is independently processed by a main-memory sort and a window scanning phase. The clustering method, with approximately only 2 passes, would dominate the global sorted-neighborhood method. Nevertheless, notice that the actual difference in time, shown in figure 9, is small for the case we considered. This is mainly due to the fact that the window-scanning phase is, for the case of our equational-theory, much more expensive than the sorting or clustering phase and thus any time advantage gained by first clustering and then sorting becomes small with respect to the overall time.

The third case, the *multi-pass* approach, would seem to be the worse of the lot. The total number of passes will be a multiple of the number of passes required for the method we chose to do each pass. For instance, if we use the clustering method for 3 passes, we should expect at least 6 passes over the dataset (for each key, one pass to cluster and another pass to window scan each cluster), while if we use the sorted-neighborhood method, we should expect $6 + 3logN$ passes (3 separate sorts). Clearly then the *multi-pass* approach would be the worst performer in time over the less expensive clustering method. Figure 9 clearly shows this increase in time.

However, notice the large difference in accuracy in figure 10. Clearly the *multi-pass*

[7]The "real" performance lines in figure 12(b) are those of figure 4(a). We included them here to give a sense on how effective our rule-based equational theory is when compared with the ideal case.

[8]In our experiments we used merge sort, as well as its parallel variant, which used a 16-way merge algorithm to merge the sorted runs.

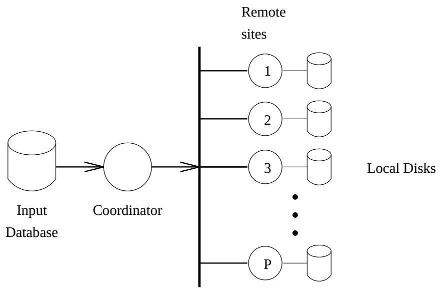


Figure 13: General shared-nothing architecture

approach has a larger accuracy than any of the two single-pass approaches. Thus, in a serial environment, the user must weight this trade-off between execution time and accuracy.

In the next section we explore parallel variants of the three basic techniques discussed here to show that with suitable 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.

5 Parallel implementation

With the use of a centralized parallel or distributed shared-nothing multiprocessor computer we seek to achieve a linear speedup over a serial computer. We briefly sketch the means of achieving this goal in this section.

The general architecture of the system used to implement the experiments detailed in this section is shown if figure 13. A processor is selected as the coordinator processor in charge of dividing the work and synchronizing each phase of the merge/purge process. We assume, without lost of generality, that the results of the merging phase will be collected at this coordinator site.

5.1 Single and Multi-pass sorted-neighborhood method

The parallel implementation of the sorted-neighborhood method is as follows. Let N be the number of records in the database, P be the number of processors in our multiprocessor environment, and w be the size (in number of records) of the merge phase window.

For the sort phase, the coordinator processor (CP) fragments the the input databases in a round-robin fashion among all P sites. Each site then sorts its local fragment in parallel. Then the CP does a P-way join, reading a block at a time (as needed) from each of the P sites.

Conceptually, for the merge phase, we should start by partitioning the input database into P fragments and assign each fragment to a different processor. The fragment assigned to processor i should replicate the last $w-1$ records from the fragment assigned to site $i-1$,

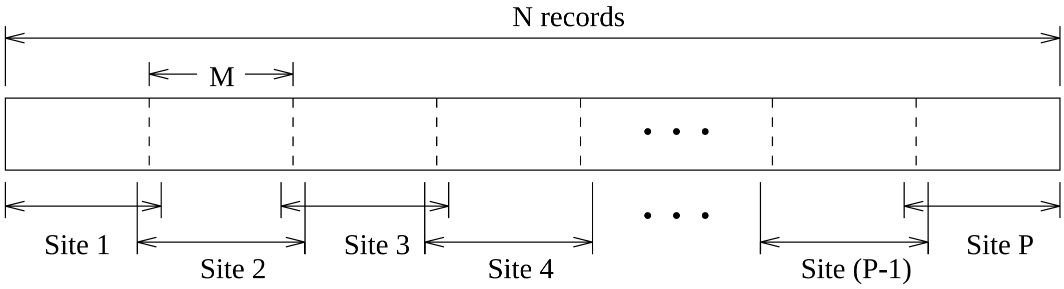


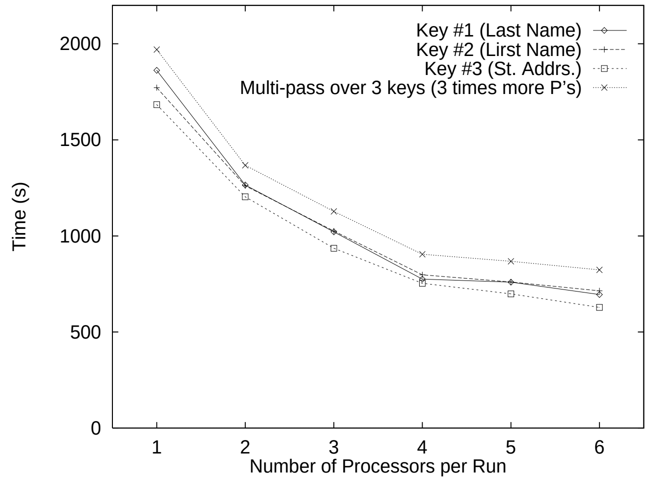
Figure 14: Partition of a sorted database with "bands" of replicated tuples.

for $1 < i \leq P$. Similarly, for $1 \leq i < P$, the fragment at site i should replicate the first $w-1$ records from the fragment at site $i+1$. These small "bands" of replicated records are needed to make the fragmentation of the database invisible when the window scanning process is applied in parallel to each fragment. This fragmentation strategy is depicted in figure 14.

For concreteness, we divide the database as follows. Let M be the number of records that fit in memory at each of the P sites. The CP reads a block of M records and send them to site 1 which stores them in memory and starts applying the window scanning procedure. The CP stores the last $w-1$ of the block sent to site 1 and reads $M-(w-1)$ records from disk, for a total of M records which are then sent to site 2. This algorithm is repeated for each site in a round-robin order until there are no more records available at the input database. Notice that the total number of replicated records is larger for this approach than the method described in the beginning of this paragraph. Nevertheless, with this approach the amount of time a processor is idle is reduced, and each site does not need to write the received blocks into disk since its processing is in memory. In fact, as each site receives a block of records, it applies the window scanning procedure to the records, sends the resulting pairs (a pair of tuple id's) back to the CP, discards the current block, and waits for the CP to send another block.

The total number of replicated records can be estimated as follows: there are, at most, $w-1$ replicated records on each block and an approximate total of $\frac{N}{M-(w-1)}$ blocks sent. Thus, the number of replicated records sent is $\frac{N(w-1)}{M-(w-1)}$ and the ratio of replicated records to the total number of records sent is $\frac{\frac{N(w-1)}{M-(w-1)}}{N+\frac{N(w-1)}{M-(w-1)}} = \frac{(w-1)}{M}$. Therefore, the window size must be small with respect to the size of the block of records for the effect of the replication to be negligible. In our system, typical values of M and w are $8K$ and 10 records, respectively.

We implemented this method on an HP cluster consisting of 8 HP9000 processors interconnected by a FDDI token-ring network. Figure 15 shows the total time taken for each of the three independent runs from figure 4(a) as the number of processors increases. The window size for all these runs was 10 records. The figure also shows the time it will take the sorted-neighborhood method to execute all three independent runs over three times the number of processors and then the computation of the transitive closure of the results. Since



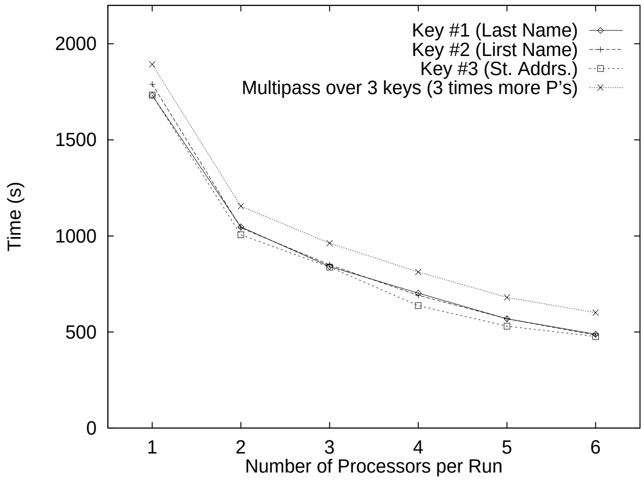


Figure 15: Time Results for the Sorted-neighborhood and Clustering Methods

we do not have enough processors to actually run all sorted-neighborhood runs concurrently, we must estimate this time using the results of each independent run. We ran all independent runs in turn and stored the results on disk. We then computed the transitive closure over the results stored on disk and measured the time for this operation. The total time, if we run all runs concurrently, is approximately the maximum time taken by any independent run plus the time to compute the closure. Notice that the speed-ups obtained as the number of processors grows are sublinear. The obvious overhead is paid in the process of reading and sending data to all processors.

5.2 Single and Multi-pass clustering method

The parallel implementation of the clustering method works as follows. Let N be the number of records in the database, P the number of processors and C the number of clusters we want to form per processor. Given a frequency distribution histogram, we divide its range into $C \times P$ subranges as described in section 3.2. Each processor is assigned C of those subranges. To cluster the data, the coordinator processor reads the database and sends each record to the appropriate processor. Each processor saves the received records in the proper local cluster. (Notice that we may precompute the cluster assignment of each record for the alternative keys on the *multi-pass* approach in only a single pass over the data.) Once the coordinator finishes reading and clustering the data among the processors, all processors sort and apply the window scanning method to their local clusters. As in the sorted-neighborhood method, resulting merge-pairs are reported to the coordinator process. Alternatively, to scale the process up, multiple coordinators can be used to cluster the data in parallel, followed by a final "cluster merging phase".

Load balancing of the operation becomes an issue when we use more than one processor and the histogram method does a bad job of partitioning the data. Our program attempts

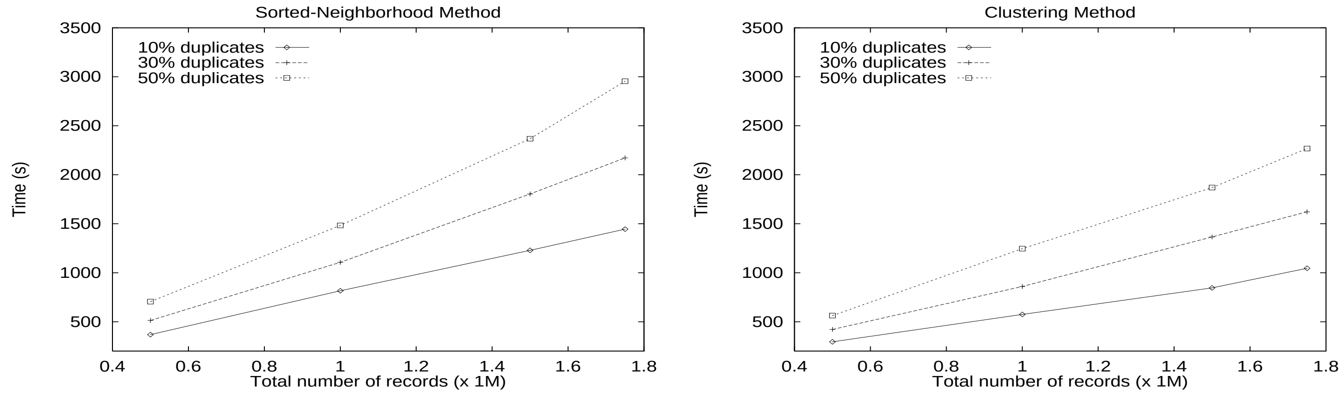


Figure 16: Time performance of the sorted-neighborhood and clustering methods for different size databases. 4 processors/run.

to do an initial static load balancing. The coordinator processor keeps track of how many records it sent to each processor (and cluster) and therefore it knows, at the end of the clustering stage, how balanced the partition is. It then redistributes the clusters among processors using a *longest processing time first* [Graham, 1969] strategy. That is, move the largest job in an overloaded processor to the most underloaded processor, and repeat until a "well" balanced load is obtained. In [Dewan *et al.*, 1994] we detailed the load balancing algorithm in the context of parallel database joins and how it can deal with skewed data distributions in a heterogenous processing environment.

The time results for the clustering method are depicted in figure 15. These results are for the same database used to obtain the timing results for the sorted-neighborhood method, a window size of 10 records, and 100 clusters per processor. Comparing the results in figure 15 we note that the clustering method is, as expected, a faster parallel process than the sorted-neighborhood method.

In all cases the *multi-pass* approach remained about 20% slower than all other single passes. However, its accuracy was consistently 50% better (compare figures 4a and 15). An alternative interpretation would be to consider how slow the single passes would be in comparison to the multi-pass for the fixed accuracy level of the *multi-pass* approach. In section 4.6 we estimated for these data sets that a single pass would require a window size of 50,000. Unfortunately, the amount of real time to verify this was far too much to justify locking up the computing resources we had available for this study.

Suffice it to say, that for very large databases of say 1 billion records, a 50% increase in accuracy for a modest 20% decrease in performance translates to a huge number of found duplicates with obvious advantages in business operation.

5.3 Scaling Up

Finally, we demonstrate that the sorted-neighborhood and clustering methods scale well as the size of the database increases. Due to the limitations of our available disk space, we

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 6: Database sizes

could only grow our databases to about 3,000,000 records. We again ran three independent runs of the sorted-neighborhood method (and the clustering method), each with a different key, and then computed the transitive closure of the results. We did this for the 12 databases in table 4 and ran all the experiments assigning 4 processors to each independent run. (We started with 4 "no-duplicate databases" and for each we scaled up by creating duplicates for 10%, 30%, and 50% of the records, for a total of 12 distinct databases of varying complexity.) The results are shown in figure 16. As expected, the time increases linearly as the size of the databases increase independent of the duplication factor.

Using the graphs in figure 16 we can estimate how much time it will take to process 1 billion records using both methods. We assume the time will keep growing linearly as the size of the database increases. For the sorted-neighborhood method, let us consider the last point of the 30% graph. Here, a database with 2,639,892 records was processed in 2172 seconds (including all the I/O time). Thus, given a database with 1,000,000,000 records, we will need approximately $1 \times 10^9 \times \frac{2172}{2639892}$ s $= 8.2276 \times 10^5$ s $\simeq$ 10 days. Doing the same analysis with the clustering method, we first notice that a database of size 2,639,892 records was processed in 1621 seconds. Thus, given a database with 1,000,000,000 records, we will need approximately $1 \times 10^9 \times \frac{1621}{2639892}$ s $= 6.1404 \times 10^5$ s $\simeq$ 7 days. Of course, doubling the speed of the workstations and utilizing the various RAID-based striping optimizations to double disk I/O speeds discussed in [Nyberg *et al.*, 1994] and elsewhere (which is certainly possible today since the HP processors and disks used here are slow compared to, for example, *Alpha* workstations with modern RAID-disk technology) would produce a total time that is at least half the estimated time, i.e. within 3-4 days.

6 Research Plan

In this thesis we propose to study the efficient implementation of band joins where the join conditions are user-defined predicates. To date, we have studied some algorithms and implemented some solutions for the merge/purge problem. There are, however, several open ends we would like to address as part of this thesis. This section provides a brief overview of the immediate future direction we pursue. We start, first, by describing the status of our current implementation and results. We then describe some ideas about the *purge* phase of the merge/purge procedure. Because of the many differences among application domains over which a merge/purge process can be used, a tool is needed to guide the user in defining all phases of the process. We provide some details of a tool we plan to build to elicit the domain knowledge from the user prior to the application of the merge/purge facility. Finally, to further demonstrate the utility of the techniques presented in this proposal, we plan to implement at least two more applications from different domains. The first application, ALEXSYS, is an expert system for allocating a mortgage pool into contracts. The second application uses merge/purge to implement a well know spatial-databases algorithm.

6.1 The Sorted-Neighborhood Implementation and Results

A large portion of the software needed to perform the sorted-neighborhood method is implemented. The modules implementing the serial versions of the sorted-neighborhood method need no major changes at this point. The modules implementing the parallel version, however, are several revisions behind the modules used for the serial version. Moreover, there is no parallel version for the duplicate elimination version of the sorted-neighborhood method yet. Work is underway to update the parallel version.

This thesis proposal presented a variety of results that were gathered over the past year. Notice, however, that these results came from a variety of architectures (e.g., Sun's IPX running SunOS 4.1.3, Sun's Sparc 5 running SunOS 5.3 – Solaris –, and HP9000/735's running HP-UX 9.01) and, thus, time performance of some experiments are not directly comparable to others. Notice also that the sizes of the databases used during the experiments changed depending on the architecture used. A set of results gathered over only one architecture is needed and will probably replace the results presented in this proposal.

6.2 The Purge Phase

In this proposal we have discussed a particular instance of the "instance identification" problem. We then addressed the most time consuming part of the merge/purge procedure, namely, the merge phase. We have completely ignored, until now, the *purge* phase.

The purge phase is very application dependent. For the type of application illustrated throughout this proposal, there could be many things the user of a merge/purge engine would like to do with those tuples identified as "similar". A possible use of the purge phase is duplicate elimination from the input datasets. However, the user might want to perform an analysis of the "duplicates" before removing all but one. For example, in the case of a mailing list, the user may want to remove all but the most recent instance. Another user might want to merge the information in some duplicates and remove only duplicates that contain redundant information. Moreover, some users might not want to eliminate duplicates but rather "infer" new information form the pieces of information marked as "similar" or "related" during the merge phase. In general, the uses of the purge phase can go from simple duplicate elimination to complex operations like those used for knowledge discovery in databases.

In section 4.3, we showed that the equational theory can falsely identify data items as "similar" (false positives). Table 3 showed some examples of false positives for the example application under study in this proposal. A study of the reasons why the equational theory produced these false positives revealed that in most of the cases, various of the distance functions used to determine equality among several fields were at their thresholds. This suggests that when distance functions are returning borderline values, we must mark the resulting pairs as "similar but possible false positive". Then, a more complex and expensive analysis of these possible false positive pairs can be executed during the purge phase.

As part of this thesis work, we plan to address the problem discussed here and provide mechanisms for the user to define the purge phase.

6.3 A Generic User Interface for Merge/Purge

Many phases of the merge/phase procedure are application dependent. For the simple example we studied in this proposal involving lists of persons, to execute the merge phase we must decide which keys to use to sort the data, provide an equational theory which could be expressed in a ruled-based form, provide functions that implement the distance functions with the proper thresholds if they are necessary, decide the size of the windows, and the number of independent runs if the multi-pass approach is used. Then, once the merge phase is defined, the user must decide what to do with the "merged" tuples during the purge phase. We discussed some example uses for the purge phase in the previous section. Needless to say, setting up every part of the merge/purge procedure without any help from an interface could be a major burden to the possible users of merge/purge facility.

As part of this thesis work, we plan to construct and demonstrate a generic graphical user interface for a merge/purge engine. This user interface must, at least, help the user to:

1. Define the different keys to be used at each independent run of the sorted-neighborhood method, and provide a comparative evaluation of the utility of alternative key structures.

2. Define the "clustering" strategy to be used, if any. This clustering strategy could be based on uniform partitioning, constant partitioning, range restrictions, hash partitioning, or classification.

3. Define an equational theory.

   (a) Use a simple rule-based language to specify the equational theory (probably a language resembling OPS5).

   (b) Provide a compiler for the rule-based language. Here we could use one of the many public-domain OPS5 compilers available. However we probably will have to change them to output code that is optimized for the window-scanning phase of the sorted-neighborhood method.

(c) Provide a library of distance functions similar to those used by us in this proposal. We should, of course, allow the user to add their own distance functions to the library.

4. Test the equational theory. The interface must provide means of extracting samples from the input databases to for testing purposes. The use of small databases permits the quick prototyping of the equational theory. The user can quickly learn the effects of adding/removing a rule to/from the equational theory, as well as changing a threshold in any distance function, to the accuracy of the obtained results. As in some expert system shells, a *explanation* facility must also be provided that would provide the user with the reason behind the decision of merging (or not merging) a pair of records.

5. Configure the sorted-merged neighborhood method to run over the available hardware. This is of particular importance when the system is run in a parallel environment.

The use of tools to elicit knowledge from the user resembles the procedure of eliciting knowledge from an expert in the process of building an expert system [Hayes *et al.*, 1983]. In fact, there has been some recent tools that, confronted with the same problem as us –a variety of domains from which applications can be chosen–, have created a knowledge acquisition tool to help the user define parameters and rules (e.g., [Solotorevsky *et al.*, 1994]).

Toolkits for building user interfaces are widely available [comp.graphics FAQ, 1994]. At the moment of writing, no decision had been made regarding which toolkit to use.

6.4 Other Applications of Merge/Purge

For this thesis proposal we implemented and studied one instance of the merge/purge problem: identifying duplicates in a database of names. Once we implement the interface described in the previous section, it would be interesting to test it under other domains where our solutions to the merge/purge problem can be beneficial. In particular, we must look for applications where unification of related entities is not well defined and might require an intensive inference process. To date, we have identified two such instances. We describe them here briefly.

6.4.1 ALEXSYS: The Mortgage Pool Allocator Expert System

ALEXSYS (AL*locator* EX*pert* SYS*tem*) is a rule-based expert system for allocating mortgage-backed securities. This allocation is a combinatorial optimization problem whose complexity is known to be NP-complete. ALEXSYS, described elsewhere [Stolfo *et al.*, 1991], has become a benchmark in the rule-base processing community [Neiman, 1994] and has also recently been used as a test case in [Dewan, 1994].

The ALEXSYS rule program works by combining sets of pools into *sell contracts* whose sizes are controlled by Public Securities Administration (PSA) regulations. Ideally, we would like to apply the PSA rules over the entire set of pools and extract the best allocations. Nevertheless, since the number of pools can be quite large, the time required makes this approach infeasible. ALEXSYS uses a set of heuristics to overcome this optimization complexity. For example, ALEXSYS first tries to allocate pools with the highest profitability into contracts. Then, it allocates single pools that can almost exactly fit into a contract. A complete list of these heuristics and the reasoning behind them can be found in [Stolfo *et al.*, 1990] and are outside the scope of this proposal. For now it suffices to mention that the practical idea behind the use of heuristics is to reduce the number of pools the rules must consider at each step of its inference process. But when the number of pools is very large, even the use of heuristics is not enough to reduce the complexity of matching the rules to the set of pools. In this case, data partitioning techniques had been used to initially divide the data into independent pieces. The inference rules are then independently applied to each distinct subset of pools.

The above procedure is analogous to the sorted-neighborhood method presented on this thesis proposal. The initial partition of the data into independent pieces is analogous to our initial sort phase while the inference phase where pools are allocated can be implemented using the window scan procedure where the PSA rules are the equational theory. In more details, our implementation of ALEXSYS via the sorted-neighborhood method will work as follows:

1. Sort the initial data by the size of the pools. By considering pools in this order, we are implementing the first heuristic mentioned above.

2. Use the window scan phase over the sorted data with a relatively "large" window. Use the ALEXSYS rules to determine which pools can be allocated into contracts. Pools allocated to contracts are purged from the list. In this case one or more pools will be deemed "equivalent" to a "good-million" contract allocation.

3. At the end of the previous step, a sorted list of still unallocated pools will remain. We apply step 2, again, to the remaining data in an attempt to fill more contracts. This procedure is repeated until either there is a pass in which no new allocation is produced, or a time deadline is reached.

6.4.2 A Spatial Join Application

Spatial Joins is one of the most frequently used operations in a *spatial* database system [Brinkhoff *et al.*, 1994]. Given a relation A whose i-th column is a spatial attribute and a relation B whose j-th column is also a spatial attribute, then $A \bowtie_\theta B$ is a spatial join if θ is a spatial predicate involving the spatial attributes of A and B [Günther, 1993]. Consider a database containing geometric information taken from a map of New York City. A spatial join would be used to answer a query like "Is Central Park <u>inside</u> Manhattan?". This particular type of spatial join is called an *intersection join.*

In [Brinkhoff *et al.*, 1994], a procedure to process intersection joins over relations with spatial attributes is described. We note analogies between their solution, which they call the *multi-step procedure of spatial joins*, and our sorted-neighborhood method. The multi-step procedure starts by applying *spatial access methods* which traditionally use an object bounding box locations as a geometric key. The key is used to select a set of possible candidates objects that satisfy the spatial join predicate. This is analogous to our initial sort with a key. The selected candidates are then examined closely using a *geometric filter* that actually determines which objects satisfy the spatial predicate. This second step is analogous to the merge phase of the sorted-neighborhood method where the geometric filter is analogous to our equational theory.

We plan to implement a simple spatial join using the user interface described in the previous section that would follow the multi-step procedure. Consider a simple image containing only a large number of squares and circles. For every circle in the image, we want to find all rectangles that are "near" that circle. Traditional solutions to this problem would cluster the objects around some centroids. But in this case, we would need each circle to become the centroid of a cluster. Alternatively, we can use the sorted-neighborhood method as follows:

1. **First pass**: Sort all objects by their x position. Use a window scan procedure to only consider a small set of objects and find all rectangles that are "near" a circle.

2. **Second pass**: Sort all objects by their y position and again use the window scan procedure to find rectangles "near" the circle.

3. Use a "transitive closure" step to union the results obtained by the two passes.

7 Contributions

We claim the following contributions from this work once completed as a thesis:

1. An implementation of a general-purpose merge/purge facility, including a parallel implementation targeted to a variety of parallel computing architectures.

2. A user interface as described in section 6.3 to help users define the domain-dependent components of the merge/purge process.

3. The sorted-neighborhood method with user-defined equational theory as a generalization of band joins. We will provide time and accuracy results of all our different approaches to the sorted-neighborhood method.

4. The multi-pass approach of the sorted neighborhood method. We use a transitive closure phase after several independent runs of the single-pass sorted-neighborhood method to obtain more accurate results in less time than it would take any single-pass run to reach the same level of accuracy.

5. Comparative evaluations of a range of examples in different domains to demonstrate its general utility.

8 Conclusion

The sorted-neighborhood method described in this proposal is expensive due to the sorting phase, as well as the need to search in large windows for high accuracy. An alternative method based on data clustering modestly improves the process in time. However, neither achieves high accuracy without inspecting large neighborhoods of records. Of particular interest is that performing the merge/purge 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.

In this proposal we have only addressed the most time consuming and important part of the merge/purge problem, namely, the merge phase. We have not addressed the *purge* phase. In many applications the purge phase requires complex functions to extract or "deduce" relevant information from merged records, including various statistical measures. The rule base comes in handy here as well. The consequent of the rules can be programmed to specify selective extraction, purging, and even deduction of information, i.e. "data-directed" projections, selections and deductions can be specified in the rule sets when matching records are found.

9 Acknowledgments

We thank Dan Schutzer of Citicorp for valuable discussions and support of this work. We also thank the support from AT&T's Cooperative Research Fellowship Program, the New York State Science and Technology Foundation through the Center for Advanced Technology in Telecommunications at Polytechnic University, and NSF grant IRI-94-13847.

References

[ACM, 1991] ACM. SIGMOD record, December 1991.

[Agrawal and Jagadish, 1988] 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.

[Anderson and Matessa, 1990] J. Anderson and M. Matessa. A Rational Analysis of Categorization. In *Machine Learning: Proceedings of the Seventh Int'l Conference*, pages 76–84, 1990.

[Batini *et al.*, 1986] 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.

[Bickel, 1987] M. A. Bickel. Automatic Correction to Misspelled Names: a Fourth-generation Language Approach. *Communications of the ACM*, 30(3):224–228, 1987.

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

[Bratbergsengen, 1984] K. Bratbergsengen. Hashing methods and relational algebra operators. In *Proceedings of the 1984 VLDB Conference*, August 1984.

[Brinkhoff *et al.*, 1994] T. Brinkhoff, H. Kriegel, R. Schneider, and S. Bernhard. Multi-Step Processing of Spatial Joins. In *Proceedings of the 1994 ACM-SIGMOD Conference*, pages 197–208, May 1994.

[Cheeseman *et al.*, 1988] P. Cheeseman, J. Kelly, M. Self, J. Sutz, W. Taylor, and D. Freeman. AUTOCLASS: A Bayesian Classification System. In *Proceedings of the Fifth Int'l Conference on Machine Learning*, pages 54–64, 1988.

[Church and Gale, 1991] K. W. Church and W. A. Gale. Probability scoring for spelling correction. *Statistics and Computing*, 1:93–103, 1991.

[Codd, 1970] E. Codd. A relational model for large shared data banks. *Communications of the ACM*, 13(6), June 1970.

[comp.graphics FAQ, 1994] comp.graphics FAQ. Computer Graphics Resource Listing. In http://www.cis.ohio-state.edu/hypertext/faq/usenet/graphics/resources-list/, 1994. Part 6 of 6, Subject 20: User Interface Builders.

[Copeland *et al.*, 1988] G. Copeland, W. Alexander, E. Boughter, and T. Keller. Data placement in bubba. In *Proceedings of the 1988 ACM-SIGMOD Conference*, pages 99–108, 1988.

[Dewan *et al.*, 1994] H. M. Dewan, M. A. Hernández, K. Mok, and S. Stolfo. Predictive Load Balancing of Parallel Hash-Joins over Heterogeneous Processors in the Presence of Data Skew. In *Proc. 3rd Int'l Conf. on Parallel and Distributed Information Systems*, pages 40–49, September 1994.

[Dewan, 1994] H. Dewan. *Runtime Reorganization of Parallel and Distributed Expert Database Systems*. PhD thesis, Columbia University, May 1994.

[DeWitt *et al.*, 1991] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. An Evaluation of Non-Equijoin Algorithms. In *Proc. 17th Int'l. Conf. on Very Large Databases*, pages 443–452, Barcelona, Spain, 1991.

[DeWitt *et al.*, 1992] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *Proceedings of the 18th VLDB Conference*, pages 27–39, 1992.

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

[Elmagarmid and Pu, 1990] A. Elmagarmid and C. Pu. Guest Editors' Introduction to the Special Issue on Heterogeneous Databases. *Computing Surveys*, 22(3):175–178, September 1990.

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

[Frawley *et al.*, 1992] W. Frawley, G. Piatetsky, and C. Matheus. Knowledge Discovery in Databases: An Overview. *AI Magazine*, pages 57–70, Fall 1992.

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

[Gotlieb, 1975] L. Gotlieb. Computing joins of relations. In *Proceedings of the 1975 ACM SIGMOD Conference*, 1975.

[Graham, 1969] R. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal of Computing*, 17:416–429, 1969.

[Günther, 1993] O. Günther. Efficient Computations of Spatial Joins. In *Proceedings of the 9th Int'l Conf. on Data Engineering*, pages 50–69, 1993.

[Harrison and Rubin, 1978] M. C. Harrison and N. Rubin. Another generalization of resolution. *Journal of the ACM*, 25(3), July 1978.

[Hayes *et al.*, 1983] F. Hayes, D. Waterman, and D. Lenat. *Building Expert Systems*. Addison-Wesley Publishing Company, Inc., 1983.

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

[Hua and Lee, 1990] K. A. Hua and C. Lee. An adaptive data placement scheme for parallel database computer systems. In *Proceedings of the 16th VLDB Conference*, pages 493–506, 1990.

[Kent, 1991] W. Kent. The Breakdown of the Information Model in Multi-Database Systems. *SIGMOD Record*, 20(4):10–15, December 1991.

[Kitsuregawa and Ogawa, 1990] M. Kitsuregawa and Y. Ogawa. Bucket spreading parallel hash: A new, robust, parallel hash join method for data skew in the super database computer (sdc). In *Proceedings of the 16th VLDB Conference*, pages 210–221, 1990.

[Knuth, 1973] D. Knuth. *The Art of Computer Programming: Sorting and Searching (Volume 3)*. Addison-Wesley, 1973.

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

[Lu and Tan, 1994] H. Lu and K. Tan. Load-balanced join processing in shared-nothing systems. *Journal of Parallel and Distributed Computing*, 23(3):382–398, December 1994.

[Miranker *et al.*, 1990] D. P. Miranker, B. Lofaso, G. Farmer, A. Chandra, and D. Brant. On a TREAT-based Production System Compiler. In *Proc. 10th Int'l Conf. on Expert Systems*, pages 617–630, 1990.

[Mishra and Eich, 1992] P. Mishra and M. Eich. Join processing in relational databases. *ACM Computing Surveys*, 24(1):63–113, March 1992.

[Munro and Spira, 1976] I. Munro and P. Spira. Sorting and Searching in Multisets. *SIAM Journal of Computing*, 5(1):1–8, March 1976.

[Neiman, 1994] D. Neiman. Issues in the Design and Control of Parallel Rule-firing Production Systems. *Journal of Parallel and Distributed Computing*, 23(3), December 1994.

[Nyberg *et al.*, 1994] 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.

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

[Pu, 1991] C. Pu. Key Equivalence in Heterogenous Databases. In *Proceedings of the First InternationFirts Internatinal Workshop on Interoperability in Multidatabase Systems*, pages 314–316, April 1991.

[Schneider and DeWitt, 1989] D. A. Schneider and D. J. DeWitt. A performance evaluation of four parallel join algorithm in a shared-nothing multiprocessor environment. In *Proceedings of the 1989 ACM-SIGMOD Conference*, pages 110–121, 1989.

[Schneider and DeWitt, 1990] D. Schneider and D. J. DeWitt. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. In *Proceedings of the 16th VLDB Conference*, pages 469–480, 1990.

[Sheth and Larson, 1990] A. Sheth and J. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22(3), September 1990.

[Solotorevsky *et al.*, 1994] G. Solotorevsky, E. Gudes, and A. Meisels. RAPS: A Rule-Based Languange for Specifying Resource Allocation and Time-Tabling Problems. *IEEE Transactions on Knowledge and Data Engineering*, 6(5):681–697, October 1994.

[Stolfo *et al.*, 1990] S. Stolfo, L. Woodbury, J. Glazier, and P. Chan. The ALEXSYS Mortgage Pool Allocation Expert System: A Case Study of Speeding up Rule-based Systems. In *AI and Business Workshop, AAAI-90*, 1990.

[Stolfo *et al.*, 1991] S. J. Stolfo, O. Wolfson, P. K. Chan, H. M. Dewan, L. Woodbury, J. S. Glazier, and D. A. Ohsie. PARULEL: Parallel rule processing using meta-rules for redaction. *Journal of Parallel and Distributed Computing (JPDC)*, 13(4):366–382, 1991.

[Thomas *et al.*, 1990] G. Thomas, G. Thompson, C. Chung, E. Barkmeyer, F. Carter, M. Templeton, S. Fox, and B. Hartman. Heterogenous Distributed Database Systems for Production Use. *ACM Computing Surveys*, 22(3):237–266, September 1990.

[Tsur, 1991] S. Tsur. PODS invited talk: Deductive databases in action. In *Proc. of the 1991 ACM-PODS: Symposium on the Principles of Database Systems*, 1991.

[Wang and Madnick, 1989] 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.

[Wolf *et al.*, 1991] J. L. Wolf, Dias. D. M., P. S. Yu, and J. Turek. Comparative performance of parallel join algorithms. In *Proceedings of the 7th Int'l Conference on Data Engineering*, pages 78–88, 1991.

A OPS5 version of the equational theory

```
;; Rules for the Merge/Purge Procedure
;;      by: Mauricio A. Hernandez
;;          Computer Science Department
;;          Columbia University
;;
(literalize   window   size max-windows top pid outpipe)

(literalize          goal    name)
(literalize        merged    id1 id2)
(literalize   similar-ssns    id1 id2)
(literalize   similar-addrs    id1 id2)
(literalize very-similar-addrs  id1 id2)
(literalize   similar-names    id1 id2)

(literalize   person  id ruid ssn name addr fname minit
                      lname stnum stname aptm city state
                      zipcode status)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; The rules in this section of the program will be the one
;; perfroming the match between records and finding
;; candidates to be merged.

(p find-similar-ssns
   (goal ^name initial-matches)
   (person ^status active ^id <id_1> ^ssn { <s1> > 0 })
   (person ^status active ^id  { <id_2> > <id_1> }
          ^ssn { <s2> > 0 (same_ssn_p  <s1> 4) } )
   - (similar-ssns ^id1 <id_1> ^id2 <id_2>)
-->
   (make similar-ssns ^id1 <id_1> ^id2 <id_2>))

(p compare-names
   (goal ^name initial-matches)
   (person ^status active ^id <id_1> ^name <name1>)
   (person ^status active ^id { <id_2> > <id_1> }
          ^name (compare_names <name1>))
   - (similar-names ^id1 <id_1> ^id2 <id_2>)
-->
   (make similar-names ^id1 <id_1> ^id2 <id_2>))

(p compare-addresses
   (goal ^name initial-matches)
   (person ^status active ^id <id_1> ^stnum <num1>
          ^stname <addr1>)
   (person ^status active ^id { <id_2> > <id_1> }
          ^stnum <num2>)
   (person ^status active ^id <id_2>
          ^stname (compare_addresses
                      <addr1> <num2> <num1>))
   - (similar-addrs ^id1 <id_1> ^id2 <id_2>)
-->
   (make similar-addrs ^id1 <id_1> ^id2 <id_2>))

(p closer-addresses-use-zips
   (goal ^name initial-matches)
   (similar-addrs ^id <id_1> ^id2 <id_2>)
   (person ^status active ^id <id_1> ^city <c>
          ^zipcode <z>)
   (person ^status active ^id <id_2>
          ^city    (same_city <c>)
          ^zipcode (same_zipcode <z>))
   - (very-similar-addrs ^id1 <id_1> ^id2 <id_2>)
-->
   (make very-similar-addrs ^id1 <id_1> ^id2 <id_2>))

(p closer-addresses-use-states
   (goal ^name initial-matches)
   (similar-addrs ^id <id_1> ^id2 <id_2>)
   - (very-similar-addrs ^id1 <id_1> ^id2 <id_2>)
   (person ^status active ^id <id_1> ^city <c> ^state <s>)
   (person ^status active ^id <id_2> ^city (same_city <c>)
          ^state <s>)
-->
   (make very-similar-addrs ^id1 <id_1> ^id2 <id_2>))

(p change-context-1
   (goal ^name initial-matches)
-->
   (modify 1 ^name second-matches))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;      Context:  Second Matches
;;

;; For those records that have very-close SSNs, use thier names
;; to determine if they are the same person.  If so, declare the
;; records "MERGED''.

(p compare-addresses-use-numbers-state
   (goal ^name second-matches)
   (person ^status active ^id <id_1>
          ^stnum { <num> > 0 }
          ^stname <addr> ^aptm { <a> <> 0 }
          ^city <c> ^state <s>)
   (person ^status active ^id { <id_2> > <id_1> }
          ^stnum (very_close_num <num>)
          ^stname (close_but_not_much <addr>)
          ^aptm { <> 0 (very_close_str <a>) }
          ^city (same_city <c>) ^state <s>)

   - (similar-addrs ^id1 <id_1> ^id2 <id_2>)
   - (very-similar-addrs ^id1 <id_1> ^id2 <id_2>)
-->
   (make similar-addrs ^id1 <id_1> ^id2 <id_2>)
   (make very-similar-addrs ^id1 <id_1>
                       ^id2 <id_2>))
```

```
(p compare-addresses-use-numbers-zipcode
   (goal ^name second-matches)
   (person ^status active ^id <id_1>
          ^stnum { <num> > 0 }
          ^stname <addr> ^aptm  { <a> <> 0 }
          ^city <c> ^zipcode <z>)
   (person ^status active ^id { <id_2> > <id_1> }
          ^stnum (very_close_num <num>)
          ^stname (close_but_not_much <addr>)
          ^aptm { <> 0 (very_close_str <a>) }
          ^city (same_city <c>)
          ^zipcode (same_zipcode <z>))

   - (similar-addrs ^id1 <id_1> ^id2 <id_2>)
   - (very-similar-addrs ^id1 <id_1> ^id2 <id_2>)
-->
   (make similar-addrs ^id1 <id_1> ^id2 <id_2>)
   (make very-similar-addrs ^id1 <id_1>
                          ^id2 <id_2>))

(p same-same-same-address-except-city
   (goal ^name second-matches)
   (similar-addrs ^id1 <id_1> ^id2 <id_2>)
   - (very-similar-addrs ^id1 <id_1> ^id2 <id_2>)
   - (merged ^id1 <id_1> ^id2 <id_2>)
   (person ^status active ^id <id_1>
          ^stnum { <num> > 0 }
          ^aptm { <a> <> 0 }
          ^zipcode <z>)
   (person ^status active ^id <id_2>
          ^stnum (very_close_num <num>)
          ^aptm { <> 0  (very_close_str <a>) }
          ^zipcode (same_zipcode <z>))
-->
   (make very-similar-addrs ^id1 <id_1>
                          ^id2 <id_2>))

(p same-ssn-and-name
   (goal ^name second-matches)
   { (similar-ssns  ^id1 <id_1> ^id2 <id_2>) <ssns>  }
   { (similar-names ^id1 <id_1> ^id2 <id_2>) <names> }
   - (merged ^id1 <id_1> ^id2 <id_2>)
-->
   (remove <ssns> <names>)
   (make merged ^id1 <id_1> ^id2 <id_2>))

(p same-ssn-and-address
   (goal ^name second-matches)
   { (similar-ssns  ^id1 <id_1> ^id2 <id_2>)     <ssns> }
   { (very-similar-addrs ^id1 <id_1> ^id2 <id_2>) <vsa> }
   - (merged ^id1 <id_1> ^id2 <id_2>)
-->
   (remove <ssns> <vsa>)
   (make merged ^id1 <id_1> ^id2 <id_2>))

(p same-name-and-address
   (goal ^name second-matches)
   { (similar-names  ^id1 <id_1> ^id2 <id_2>)   <names> }
   { (very-similar-addrs ^id1 <id_1> ^id2 <id_2>) <vsa> }
   - (merged ^id1 <id_1> ^id2 <id_2>)
-->
   (remove <names> <vsa>)
   (make merged ^id1 <id_1> ^id2 <id_2>))

(p change-context-2
   (goal ^name second-matches)
-->
   (modify 1 ^name third-matches))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;        Context:  Third Matches
;;

(p very-close-ssn-close-address
   (goal ^name third-matches)
   (similar-addrs ^id1 <id_1> ^id2 <id_2>)
   (similar-ssns  ^id1 <id_1> ^id2 <id_2>)
   - (similar-names ^id1 <id_1> ^id2 <id_2>)
   (person ^status active ^id <id_1> ^ssn <s1>)
   (person ^status active ^id <id_2> ^ssn (same_ssn_p <s1> 2))
   - (merged ^id1 <id_1> ^id2 <id_2>)
-->
   (make merged ^id1 <id_1> ^id2 <id_2>))


(p hard-case-1
   (goal ^name third-matches)
   { (very-similar-addrs ^id1 <id_1> ^id2 <id_2>) <sa> }
   (person ^status active ^id <id_1>
          ^fname <fn>
          ^stname <addr1>
          ^zipcode <z>)
   (person ^status active ^id <id_2>
          ^fname (same_name_or_initial <fn>)
          ^stname <addr1> ^zipcode <z>)
   - (merged ^id1 <id_1> ^id2 <id_2>)
-->
   (remove <sa>)
   (make merged ^id1 <id_1> ^id2 <id_2>))

(p goto-report
   (goal ^name third-matches)
-->
   (modify 1 ^name do-report))
```

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;
      }

    }
  }
}
```