An Extensible Framework for Data Cleaning

Helena Galhardas, Daniela Florescu, Dennis Shasha, Eric Simon

s paires es impaires

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

An Extensible Framework for Data Cleaning

Helena Galhardas - Daniela Florescu - Dennis Shasha - Eric Simon

N ° 3742

Juillet 1999

THÈME 3

ISSN 0249-6399

INRIA
ROCQUENCOURT

An Extensible Framework for Data Cleaning

Helena Galhardas* - Daniela Florescu† - Dennis Shasha‡ - Eric Simon§

Thème 3 — Interaction homme-machine,
images, données, connaissances
Projet Caravel

Rapport de recherche n ° 3742 — Juillet 1999 — 44 pages

Abstract: Data integration solutions dealing with large amounts of data have been strongly required in the last few years. Besides the traditional data integration problems (e.g. schema integration, local to global schema mappings), three additional data problems have to be dealt with: (1) the absence of universal keys across different databases that is known as the object identity problem, (2) the existence of keyborad errors in the data, and (3) the presence of inconsistencies in data coming from multiple sources. Dealing with these problems is globally called the data cleaning process. In this work, we propose a framework which offers the fundamental services required by this process: data transformation, duplicate elimination and multi-table matching. These services are implemented using a set of purposely designed macro-operators. Moreover, we propose an SQL extension for specifying each of the macro-operators. One important feature of the framework is the ability of explicitly including the human interaction in the process. The main novelty of the work is that the framework permits the following performance optimizations which are tailored for data cleaning applications: mixed evaluation, neighborhood hash join, decision push-down and short-circuited computation. We measure the benefits of each.

Key-words: data integration, data cleaning, query language, query optimization, approximate join, data transformation

(Résumé : tsvp)

* INRIA Rocquencourt, France Helena.Galhardas@inria.fr
† INRIA Rocquencourt, France Daniela.Florescu@inria.fr
‡ New York University and INRIA Rocquencourt, France shasha@cs.nyu.edu
§ INRIA Rocquencourt, France Eric.Simon@inria.fr

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : 01 39 63 55 11 - International : +33 1 39 63 55 11
Télécopie : (33) 01 39 63 53 30 - International : +33 1 39 63 53 30

Un Environnement Logiciel Extensible pour le Nettoyage de Données

Résumé : La demande en solutions d'intégration de grands volumes de données est fortement présente aujourd'hui. En plus des problèmes traditionnels d'intégration de schéma, trois autres problèmes doivent être pris en compte: (1) l'absence de clés universelles pour différentes sources de données, connu sous le nom de problème d'identification d'objets, (2) l'existence d'erreurs de frappe dans les données et (3) la présence d'incohèrences dans les données qui proviennent de sources différentes. Le traitement de ces problèmes est en général appelé nettoyage de données. Nous proposons un environnement logiciel qui offre les principaux services nécessaires à un processus de nettoyage de données, à savoir: la transformation de données, le dédoublonnage de données et la mise en correspondance entre plusieurs tables, encore appelée jointure approximée. L'implémentation de ces services est supportée par un ensemble de macro-operateurs spécifiables à l'aide d'une extension de SQL. Une caractéristique très importante de l'environnement proposé est l'inclusion explicite de l'utilisateur dans le processus de nettoyage. Une contribution originale de notre travail est un ensemble de techniques d'optimisation adaptées aux applications de nettoyage de données telles que l'évaluation mixte ou le *neighborhood hash join*.

Mots-clé : intégration de données, nettoyage de données, langage et optimisation de requêtes, jointure approximée, transformation de données

1 Introduction

In the last few years, there has been a big demand for data integration solutions that deal with large amounts of data. Besides the problems that have been traditionally addressed in the past, like schema integration, local to global schema mapping and query decomposition, data integration applications have to cope with three additional problems:

1. Data coming from different origins may have been created at different times, by different people and using different conventions. In this context, the question of deciding which data refers to the same real object becomes crucial. A company may have information about its clients stored in several tables, because each client buys different services that are managed by distinct departments. Once it is decided to build a unified repository of all the company clients, the same customer may be referred to in different tables by slightly different but correct names: "John Smith", "Smith John" or "J. Smith". This kind of mismatching is called the *Object Identity problem* in [8][1].
2. There may be *errors* in data. Usually due to mistyping ("John Smith" and "Joh Smith"), errors also result in the object identity problem once data is being integrated, since the same object may be referred to by more than one data record and some may be erroneous. Nevertheless, the algorithms used to detect that fields with data entry errors refer to the same object are not the same as the ones used to recognize that fields written in a different format are correct alternatives.
3. There may be *inconsistencies*. Even though different records may have been recognized to store information about the same object (because the name fields are similar, for example), they may carry contradictory information (for instance, two different birth dates for the same person object).

In this paper, we are interested in the case where the contents of multiple autonomous data sources are combined and reconciliated, with the purpose of building a materialized, error free, unified view of them. Following the data warehouse terminology [3], we shall call *data cleaning* the process of eliminating the three above problems during the construction of the materialized view.

1.1 Statement of the Data Cleaning Problem

To illustrate some of the major steps followed by a data cleaning process, we use a concrete example. In the case illustrated in figure 1, we want to gather into a materialized view, named CLIENT, some information that tells us which clients (identified by an Id, a name, an address, a user name, and a job), have a cellular phone, a phone line or an Internet connection at home. This information is a priori disseminated into the three source tables:

[1] This problem is sometimes called Instance Identification problem, or duplicate elimination or record linkage problem in the case of a single source.

GSM-CLIENT, HOME-CLIENT, and INTERNET-CLIENT. This example requires a data cleaning process since there is no common client key in the three source tables and some data entry errors may have been introduced when filling in the fields of each table.

The first step is to determine the client records (each taken from a different source table) that correspond to the same client in the real life (object identity problem). Henceforth, such records are called *matching records*. One may express that records are matching records if their name and address values are equal, or if their name values have a high degree of similarity using some notion of similarity. A more complex criteria could specify that two records are matching records if their name values are similar and one record comes from the GSM-CLIENT table while the other comes from the HOME-CLIENT table. Before comparing records, some conversions of individual table fields may be needed since data formats may vary. For instance, names can be normalized using the format *<lower_Case(name)>*.

Once matching records are found, a second important step is to inspect the matching records and determine which ones are valid or not. This is necessary because a single tuple, say of HOME-CLIENT, may a priori match with several records in GSM-CLIENT or INTERNET-CLIENT due to the approximation performed by the matching criteria. One possibility is to ask the user to inspect all matching records, possibly entailing the examination of a huge number of records. Another possibility is to keep the matching records that have the highest similarity value(e.g., wrt a threshold), provided that such value is kept with each matching record.

The last phase is to integrate the remaining matching records and generate tuples for the materialized view. Several possible rules for integration can be envisioned. For instance, the name value of CLIENT records can be taken from the GSM-CLIENT matching records while the address value is taken from the HOME-CLIENT matching records. The job value of CLIENT records can be obtained by concatenating the two job values of matching records from GSM-CLIENT and HOME-CLIENT. These choices can be specified in advance or interactively performed by the user during the cleaning process.

As a general goal, the cleaning process must assure a good quality of the data generated in the materialized view, in the natural sense: (i) client records that correspond to the same client, and only those, should be found as matching records, and (ii) integrated matching records should be error-free. To satisfy this quality goal, the cleaning process must have two major features. First, it should be iterative and interactive. Indeed, during a first iteration, matching records are found based on initial matching criteria. After inspection of the result, errors may be found (e.g., some matching records should not match). As a consequence, new matching criteria are defined and a new matching phase is started again. Similarly, the integration phase can also be iterated to refine the integration criteria after manually controlling the first generated results. The involvement of a human being (usually, an expert) in the different stages of the cleaning process is necessary because many errors and inconsistencies cannot be resolved automatically. As a second feature, the cleaning process must be expressive and extensible. It should be possible to express a rich variety of criteria for matching records, eliminating erroneous matches, and integrate remaining matching records. Furthermore, it should be easy to incorporate domain specific data normalization,

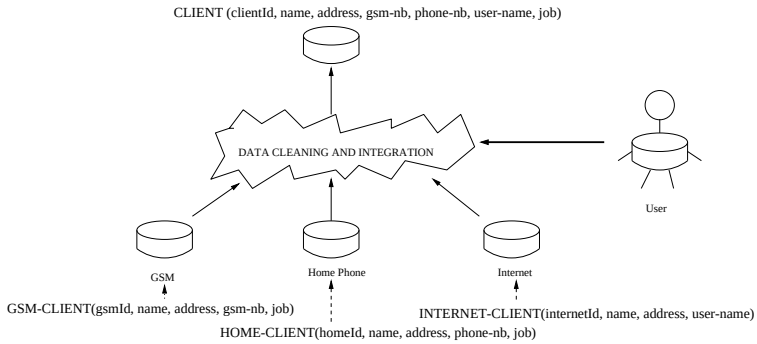

Figure 1: Integrated view of Telecom customers

matching, and integration algorithms in the cleaning process. For instance, the algorithms used in practice to compare person names or addresses may not be the same. Extensibility allows both to be invoked during the cleaning process.

However, specifying and implementing a cleaning process with good quality result raises three main difficulties. First, the volume of data involved in a cleaning process is usually large, which makes the matching phase quite expensive. Optimization techniques are needed to keep this cost reasonably acceptable, knowing that this phase can be iterated over several times, while at the same time avoiding to miss some pertinent matching records (i.e., some comparisons). Second, because humans are an expensive resource, it is important to automate as much work as possible before the user interferes in the cleaning process, and give to the user the maximal amount of information so that well founded decisions can be taken (e.g., to eliminate erroneous matching records). Typically, one wants to minimize the amount of data that needs to be inspected manually. Third, due to the iterative nature of the cleaning process, it is essential to be able to easily modify the whole specification and implementation of the cleaning process. Note that the two first points merely reflect the tradeoff that exists between quality and efficiency.

1.2 Existing Solutions for Data Cleaning

A first obvious solution is to write a dedicated application program (e.g., a C program), that implements the whole data cleaning process after extracting the source data from some databases and storing them into files. Such a program could call external procedures to compare record fields during the matching phase. However, this solution is not appropriate for the following reasons. First, this solution is not flexible enough to support the iterative process of data cleaning explained earlier since modifying parts of the process would mean rewriting large amounts of code. Second, this solution does not take advantage of the data processing capabilities offered by current database systems to implement some costly data

intensive operations (e.g., joins) during the matching phase. Last, optimizations are left to the programmer and hand-wired in the code.

The Squirrel system [37] provides an automatic way to generate integration mediators from a declarative specification in a language called ISL. Using ISL, it is possible to specify a rich variety of matching criteria to produce matching records from several input tables. However, the semantics enforced by the system is limited so that a given record in a table can only match with exactly one record in another table. Hence, when a record is matching with a second record, a conflict is raised and the user is called to resolve it. This is not suitable both for performance reasons and because the user has to take decisions locally without having a global view of all possible conflicts.

A related work in [4] proposes a formalism based on annotated non-recursive Datalog for specifying inter-schema correspondences (conversion, matching and reconciliation) between relational tables to integrate. Each correspondence is specified by a set of conditions under which it is applicable and a program that implements the correspondence. Unlike Squirrel, a tuple in a table can match with several records in another table. However, no means are provided to support the analysis of the matching records (as in the second step of our example), before the generation of the materialized view. This may sacrifice the quality of the cleaning process because either the user may have to inspect too many matching records, or as a shortcut all matching records can be kept and consequently errors can be introduced in the view.

The WHIRL system[7, 8] provides a powerful way to generate approximate matching records using SQL queries enhanced with a special join operator that uses a similarity comparison function based on the vector-space model commonly adopted in statistical information retrieval. Matching records are sorted according to their similarity value. However, as it is targeted to integrate data coming from different Web sites, WHIRL does not provide any means to support the subsequent phases required by a data cleaning process, such as the second and third steps of our example. Furthermore, the proposed solution does not work when duplicates exist in the source tables. Last, the process is by nature not interactive since all decisions are left to the vector-space metrics.

In [13], a method, called Multi-Pass Neighborhood Method, is proposed to detect the maximum number of exact or approximate duplicate records (due to errors or to the lack of field standardization) in the smallest amount of time. The method[2] consists of repeating the two following steps after having concatenated all the records to be cleaned into a single file: 1) choose a key (consisting of one or several attributes, or substrings within the attributes) for each record and sort records accordingly; 2) compare those records that are close to each other within a fixed, usually small, sized window. The criteria for comparing records in order to find out duplicated ones is defined through a set of matching rules encoded in a proprietary programming language (for instance, "C") [14]. Each execution of the two previous steps (each time with a different key) produces a set of pairs of matching records. A final transitive closure is then applied to those pairs of records, yielding a union of all pairs generated by all

[2] which is now available as the Informix DataCleanser data blade [10].

independent executions, plus all those pairs that can be inferred by transitivity of equality. The main drawback of the method is that, besides being a knowledge and time intensive task (since it implies several passes on the data), finding the suitable key for putting together similar records may be quite hard, or even impossible. If the field values to be compared by the matching criteria are not standardizable (e.g. client names or European addresses), they may be written in slightly different and yet similar ways in distinct records which will not be put close to each other by any sorting key. In an extreme case, those records eventually become neighbors after many passes of execution using different keys, but this is not guaranteed to happen in a reasonable period of time. As another drawback, the support for cleaning rules offered in [13], but also in [23], [16] and some commercial data cleaning tools, allow matching rules to be applied only to pairs of neighbor records in the same file. Therefore, no distinction of sources is done since the records are concatenated in a single file before being compared, and the schema of the individual source tuples is supposed to be the same.

Other relevant related work has been done. The YAT [6] and the TranScm [22] systems as well as commercial tools as Oracle's SQL*Loader utility [24] propose solutions for data conversion. The prototype described in [29] proposes a Name-Address standardizer that aims at solving the object identity problem for a specific domain. The SERF system [5] offers a set of reusable and extensible primitives to deal with schema evolution and transform data accordingly, but do not consider the above mentioned data problems. Some work has been done focusing on data fusioning as [25] and [9]. And finally, a description of some commercial tools for data cleaning can be found in [26], [36] and [17].

1.3 Proposed Solution

In this paper we propose a new framework, in which a data cleaning application is modeled as a directed acyclic flow of transformations applied to the source data. The framework consists of a free and open system platform that offers three main services. A *Data Transformation* service is offered to perform data schema conversion and standardization of attribute values. The *Multi-Table Matching* service, also called *Approximate Join*, enables to produce a single unified table from a set of input tables after solving the object identity problem. This service consists of a sequence of matching and merging phases. Finally, the *Duplicate Elimination* service enables to remove exact and approximate duplicates from a single data source through a sequence of matching, clustering, and merging phases. A notable feature of our framework is that the user can be explicitly involved in the Multi-Table Matching and Duplicate-Elimination services, for eliminating errors and inconsistencies in the matching records, and merging matching records. A second feature is the extensibility of the framework, which is a crucial requirement because it is likely that our framework will not cover all possible data cleaning applications. Each service is supported by a few macro-operators, shared by all services, that can be used to specify the behavior of the service for a given application (e.g., how duplicates should be eliminated). Our framework can be customized to the needs of a particular cleaning application in the following ways: *(i)* the proposed macro-operators can

invoke external domain specific functions (for normalization and matching, for instance) that can be added to a pre-defined library of functions; *(ii)* operators can be combined with SQL statements in order to obtain complex data cleaning programs, *(iii)* the set of algorithms proposed for certain operators (as clustering and decision) can be extended as needed. In addition to the proposition of an open and extensible framework, this paper makes two technical contributions:

1. The SQL-like command language we propose for expressing each of our macro-operators is both expressive and flexible. In particular, our language is more expressive than the WHIRL SQL-like language in the sense that duplicates are allowed in the sources, richer similarity functions can be used, and more operators can be expressed, e.g., to support the merging of matching records. Finally, the semantics of our macro-operator for matching is more expressive than the windowing approach of [13]. Indeed, with our matching operator, each record is compared with every other record through a Cartesian product-based algorithm. As a result, every set of matching records that can be produced by the windowing approach is computable by our operator whereas the reverse is not true. The flexibility of our language is achieved by the declarativeness of our macro-operators, itself inherited from the SQL-like flavor of the language which is not the case with the special operators offered by commercial tools such as Integrity [35]. This property guarantees easy deployment and maintenance of data cleaning programs.

2. We provide several techniques to make the execution of our macro-operators efficient. First, the execution of our operators is partially supported by both a regular SQL query engine and an external home-grown execution engine. This dual architecture enables us to exploit the computing capabilities offered by relational database systems, as well as particular optimization techniques which are tailored to the specificities of our operators, such as the support for similarity functions. Second, we present optimization techniques that render our Cartesian product-based semantics of the matching macro-operator still feasible, even for large amounts of data.

This paper does not address all the issues involved in data cleaning. For instance, we do not provide any mechanism to help the user discovering the integration and cleaning criteria which are most appropriate for a certain application. This relies mainly on the knowledge and experience accumulated by experts working in the application domain, and dealing daily with anomalies in their data. Also, we do not cover algorithms for the incremental maintenance of materialized views when data sources are updated.

Apart from this introduction, the paper is organized as follows. Section 2 presents our general framework and describe our macro-operators. Section 3 describes our novel optimizations techniques and motivate their design by means of examples. Some experimental results that demonstrate the value of our approach are presented in the fourth section. Finally, Section 5 concludes.

2 Framework

In this section we first give an overview of the data cleaning process enabled by our framework through its functional decomposition into services. Then, we describe the semantics and syntax of the macro-operators that support those services. Each macro-operator is implemented by a physical operator. Some physical operators are already known as *map* for supporting the mapping macro-operator. Finally, we describe the metadata required to document a data cleaning and integration process so that back tracing is possible.

2.1 Overview of Framework's Services

In our framework, a data cleaning process accepts as input a set of possibly erroneous and inconsistent base tables and returns a consistent error-free set of tables composing the integrated view. A basic assumption of our work is that tuples within a single table are uniquely identified by a key attribute. Furthermore, we consider that the integrated view of a join between a table R and a table S models a 1:1 relationship, i.e. in the resulting view, each tuple from R will match with at most one tuple from S, and symmetrically, each tuple from S can match with at most one tuple from R[3]. The complete transformation from the input data to the output view is decomposed into a sequence of steps, each of which corresponds to the use of a service provided by our framework.

Before describing in detail the framework, we would like to emphasize once again that its purpose is to cover the more frequent requirements of data cleaning and integration applications. Nevertheless, we believe that it is able to encompass a large family of such applications due to its extensibility. We now successively describe each one of the services provided by our framework.

1. The *Data Transformation* service applies external functions to either transform the values of certain attributes, for the purpose of simplification and standardization, or convert data from one schema to another so that further comparison is possible. This service is supported by a mapping macro-operator, which takes n tables as input and produces m tables as output.

2. The *Multi-Table Matching or Approximate Join* service performs three different tasks on a set of input tables. The first task is to produce a set of matching records. This task is supported by a matching macro-operator, which performs a generalized join operation over n input tables. In the case of two tables, it associates a similarity value between 0 and 1 to every pair of tuples in the Cartesian product of the two tables. Then follows the task that analyzes the matching records and finds out which are the valid matches. This task is supported by a decision macro-operator. Finally, the last task is to generate an error-free table with unique records. This is supported by a mapping macro-operator as the one used by the Data Transformation service.

[3]Being able to treat 1:n relationships requires a non-trivial extension of the algorithm presented in this paper, and we will concentrate our attention on this important case in future work.

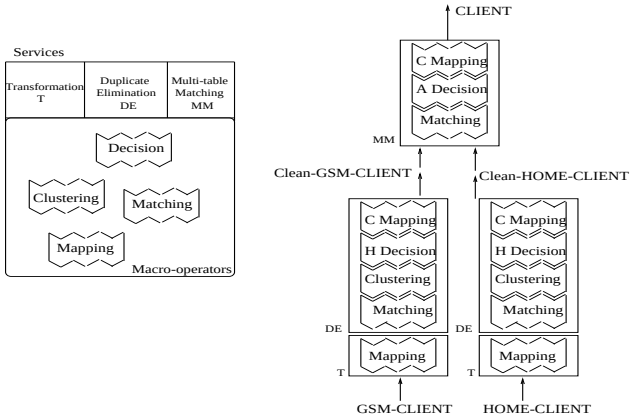

Figure 2: Framework and a data cleaning process for the Telecom example

3. The *Duplicate-Elimination* service performs four distinct tasks on a given input table. The first one is also supported by the matching macro-operator that is applied to one input table and aims at finding exact or approximate duplicate records. The next task is to group matching records into clusters of potentially similar objects. This task is supported by a clustering macro-operator, which takes as input a table and produces as output a set of clusters. The third task is performed by the decision operator using a different algorithm that analyzes the clusters and finds out which are the unique records. And finally the mapping operator is also applied in order to construct integrated and cleaned tuples.

Figure 2 shows the three services and the macro-operators that support them on the left side. The right side of the figure depicts the successive phases of a partial cleaning process for the Telecom example and highlights the use of the macro-operators in each phase. Note that some are noted as "automatic" (*A Decision*) while others are noted as "human" (*H Decision*), meaning that the user is interactively involved in the operation. *C mapping* stands for construction mapping that corresponds to the instantiation of the mapping operator for constructing integrated tuples. Source tables GSM-CLIENT and HOME-CLIENT are in a same relational format and fields have to be appropriately normalized through a transformation process. Then, a duplicate-elimination step is applied to both GSM-CLIENT and HOME-CLIENT if they have approximate duplicate records. And, finally a multi-table matching step is applied to the result of these two steps.

2.2 Macro-Operators

In this section we present the syntax and semantics of the macro-operators provided by our framework. Each operator corresponds to an SQL-like command. To make the presentation more illustrative, we present each one of the tasks that constitutes the services introduced before, and show how it is supported by a macro-operator.

2.2.1 Data Mapping

An example of a command requesting a data transformation is the following:

```
CREATE MAPPING MP1
SELECT g.gsmId, lowerName, street, number, pob, city, g.gsm-phone, newjob
FROM GSM-CLIENT g
LET lowerName = lowerCase(g.name)
    [street, number, pob, city] = extractAddressComponents(g.address)
    newjob = IF (g.job != null) THEN RETURN g.job ELSE RETURN "unknown"
```

The string MP1 that follows the keywords CREATE MAPPING indicates the name of the output table. This convention extends to all macro-operators. In this example the client names of the GSM-CLIENT table are converted to lower case, address fields are decomposed into four fields (street, number, post office box and city) and null values of the job field are replaced by the string "unknown". The specification of the data transformations applied to each tuple of the GSM-CLIENT table is done using the LET clause. This clause consists of a sequence of variable declarations of the form: <*var_name*> = <*expression*>. Variable declarations define expressions, used to compute the value of the corresponding variables, that are evaluated for each element in the Cartesian product of the tables in the FROM clause. Expressions include calls to external functions, which can be either given by the user or predefined in the system platform. Moreover, an *if-then-else* construction enables to express complex expressions as shown in the example. Expressions can refer to any variable defined in the LET clause, or to some attributes of the input tables. Finally, the SELECT clause can project the value of any attribute of a relation appearing in the FROM clause, or variable specified in the LET clause.

More generally, a mapping operator can create multiple tables from the same set of operand tables. In this case, multiple SELECT clauses can be specified, and each of them explicitly states the output tables, using an INTO < *table* > additional specification. Let us imagine that we want to produce two tables as output of the previous mapping: one of the normalized GSM clients called NORM-GSM-CLIENT and a second one JOBS that stores the correspondences between existing jobs and client identifiers. Both tables were created before. In this case, MP1 does not represent anymore the output table, since the output is explicitly specified by the INTO clauses. This is represented by:

```
CREATE MAPPING MP1
SELECT g.gsmId, lowerName, street, number, pob, city, g.gsm-phone, newjob INTO NORM-GSM-CLIENT
```

```
SELECT g.gsmId, newjob INTO JOBS
FROM GSM-CLIENT g
LET lowerName = lowerCase(g.name)
    [street, number, pob, city] = extractAddressComponents(g.address)
    newjob = IF (g.job != null) THEN RETURN g.job ELSE RETURN "unknown"
```

2.2.2 Data Matching

A matching operator is specified by means of a specific MATCH command, of which an example is given below.

```
CREATE MATCH M1
FROM GSM-CLIENT g, HOME-CLIENT h
LET sim1 = nameSimF(g.name, h.name)
    sim2 = addressSimF(g.address, h.address)
    SIMILARITY = IF (sim1 > 0.8 and sim2 > 0.9) THEN RETURN min(sim1, sim2)
                 ELSE IF (sim1 > 0.8) THEN RETURN sim2
                 ELSE IF (sim2 > 0.9) THEN RETURN sim1
                 ELSE RETURN 0;
WHERE h.phone-nb LIKE '01%'
THRESHOLD SIMILARITY >= 0.6;
```

Given a set of input relations specified in the FROM clause, the MATCH command produces as result a relation whose schema includes a key attribute for each input relation, and an extra attribute named *similarity* whose domain is a real number between 0 and 1. In the example, the projected keys are automatically named *key1* and *key2* corresponding respectively to *gsmId* and *homeId*.

An important point to note about the matching operator is that the computation of the *similarity* variable in the LET clause is mandatory. In our example the similarity is calculated using a complex formula, based on the value of the variable *sim1* (that stores the similarity between the client names) and the value of the variable *sim2* (that stores the similarity between the corresponding addresses). Functions *nameSimF()* and *addressSimF()* used for this purpose are external functions that compute the closeness between names and addresses, respectively. A class of functions commonly used for comparing strings are edit distance functions. They compute and return the distance between two strings. Let us thus consider that the relationship between similarity and edit distance is given by $similarity = 1 - distance/1000$ where 1000 is an arbitrary constant defined for convenience. A simple example is the Hamming distance function [] that aligns the two input strings and returns the number of characters that are different among them. Two more complex edit distance functions are the Smith-Waterman algorithm [33] and the Ukkonen algorithm [34]. Such functions are written in a programming language, and they are either included in the framework's pre-defined library of functions, or are added by the user.

The THRESHOLD clause specifies an optional acceptance or matching similarity threshold value. In the presence of such a clause, the tuples in the Cartesian product whose

similarity value is below the threshold value will be discarded from the result. The semantics of the THRESHOLD clause is similar to the SQL WHERE clause except that it imposes a condition on the value of an internal variable rather than on an input table attribute value.

Finally, a simple example of a matching operator for finding duplicates in table GSM-CLIENT is the following:

```
CREATE MATCH M2
FROM GSM-CLIENT g1, GSM-CLIENT g2
LET SIMILARITY = simf(g1.name, g2.name)
WHERE g1.gsmId != g2.gsmId
THRESHOLD SIMILARITY > 0.5
```

where *simf* stands for a similarity function in general and the WHERE clause avoids a match of each record with itself.

To summarize, the semantics of the matching operator is that of a Cartesian product with external function calls. The tables involved in the Cartesian product are the ones specified in the FROM clause and the external functions are called in the LET clause.

2.2.3 Data Clustering for Duplicate Elimination

The clustering phase is only needed when removing duplicates. In the case of the approximate joins, the validation of the matching information is performed directly on the result of the matching phase. In order to perform duplicate elimination, tuples that have been considered similar in the matching phase have to be grouped into sets, each of which may represent a unique object in the real world [15]. The clustering command enables to call the clustering algorithms: by transitive closure and by diameter, provided by the framework function library or others supplied by the user.

- *by transitive closure:* a similarity value above the threshold value specified in the matching operator induces a symmetric relationship between tuples in the input table. The transitive closure of this relationship fragments the table content into disjoint partitions. Each partition will give birth to a cluster; clusters produced in this manner can be either treated automatically, or by a human.
- *by diameter:* the clustering by diameter will return all sets of tuples S which satisfy the following two conditions: (a) for each pair (t_1, t_2) of tuples in S, the similarity value between t_1 and t_2 is above the threshold specified in the matching operator and (b) S is a maximal set with the property (a). As it can be immediately observed, clustering by diameter will not fragment the set of tuples into disjoint partitions. Instead, it will produce potentially overlapping sets of tuples with high degree of connectivity. Therefore, this kind of clustering is targeted only for human consumption.

In both cases, for each constructed cluster a unique key is generated. The result of this operation is a binary table $(clusterKey, tupleKey)$ which associates tuples in the input table to clusters. The command used to specify the clustering by diameter is:

```
CREATE CLUSTER C1
FROM M2
BY diameter
```

The name of the function implementing the clustering algorithm to be invoked is specified in the BY clause (if a clustering by transitive closure was required, the keyword BY would be followed by the name *transitiveClosure*). In both cases, a global cluster similarity value, identified by keyword GSIMILARITY, may be associated to each constructed cluster, representing the global similarity of the cluster and computed from the similarities of its members using an aggregate function (e.g. min, max, avg) or a combination of aggregate functions. Introducing the cluster similarity could modify C1 as follows:

```
CREATE CLUSTER C2
FROM M2
BY diameter
LET GSIMILARITY = min(SIMILARITY)
THRESHOLD GSIMILARITY > 0.7
```

where the LET clause specifies the value of each cluster similarity as the minimum of all similarities among members of that cluster. Each returned tuple contains then an additional field which is the cluster similarity. In this case the presence of the THRESHOLD clause indicates that an additional filtering is performed, and all the clusters whose global similarity is *not* above 0.7 are not included in the result. The semantics of the THRESHOLD clause is analogous to an SQL HAVING clause.

2.2.4 Data Decision

The data decision judges the correctness of the matching information that has been produced in the matching phase. The algorithms to be used in this operation vary according to the type of application (i.e. duplicate elimination and approximate joins) and they can be written by the user or may be chosen among the ones provided by the framework's library of functions. Both in duplicate elimination and approximate join, the decision can be done in three ways: (a) automatically, (b) manually or (c) semi-automatically.

Decision for duplicate elimination

- **Automatic** decision of the duplicate tuples can only be taken after a clustering phase by transitive closure, since the clustering by diameter does not induce a disjoint partitioning of the set of tuples of the relation. In this case, we use the framework's library **cluster cohesion** algorithm, which declares that all the matching tuples that belong to a cluster are duplicates.
- **Semi-automatic** decision of duplicate tuples performs as in the previous case, with an additional THRESHOLD limit that distinguishes the clusters which are automatically detected as sets of duplicates (by cluster cohesion). The clusters whose global similarity is

under the given threshold will be analyzed manually and the decision is taken appropriately. As in the previous case, the semi-automatic decision is only possible if the clustering method applied previously was transitive closure. In order to be able to apply the threshold filter, a global similarity of the cluster has to be defined[4]. This decisional mode is expressed in our language as:

```
CREATE DECISION D1
FROM C1
BY clusterCohesion
THRESHOLD GSIMILARITY > 0.9
```

- **Manual** decision of duplicate tuples can be applied after a clustering phase by either transitive closure, or by diameter. In this case, a human has to inspect each cluster and extract what she believes are unique tuples. However, the clustering phase is still necessary in order to minimize the user's effort, and to guarantee that her decisions are taken based on a maximal knowledge of potential duplicates.

Decision for multi-table matching

- **Automatic** decision of the final pairs of matching tuples may be taken as follows. Given the result of the match phase, which computes the set of potentially matching tuples, as well as their similarity, the automatic decision phase chooses all the matching pairs (r,s) such that there is no other matching pair of the form (r',s) or (r,s') such that $similarity(r', s) > similarity(r, s)$ or $similarity(r, s') > similarity(r, s)$. We will call such a pair (r,s) a *best match* [32]. This semantics can be implemented by the **best match first** algorithm[5] that is provided as default by the framework:

1. Let L= the list of matching pairs of tuples, *sorted* in descending order of similarity value
2. Iterate over the list L
   2.1. let (r,s) be the current pair from L
   2.2. declare automatically that the tuples r and s *do* join
   2.3. remove from the rest of the list L all pairs referring to either r or s
   (which have already been declared to match)

We will refer to the step 2.3. in this algorithm as the **decision purge**. There are two exceptional situations for this algorithm where the user may want to be called. The first one arises when there are at least two pairs of matching records (r, s) and (r, s') with equal similarity - we call them *ties*. The second case is highlighted by the following example: (r, s, sim_1), (r, s', sim_2) and (r', s', sim_3) and $sim_1 > sim_2 > sim_3$. In these situations, the decision purge is not performed as described. Those tuples are shown to the user and only afterwards a decision purging can be executed. If the best match first algorithm is used, the syntax of the decision operation is the following:

[4]Recall that global similarities values are optional for clusters.

[5]Remark that this algorithm is only valid if we assume that the matches between R and S are 1:1. Extensions for 1:m and m:n are presented in section 3

```
CREATE DECISION D2
FROM M1
BY bestMatchFirst
```

where *bestMatchFirst* in the BY clause stands for the name of the function implementing this algorithm. If the user wants to provide another automatic decision algorithm, she replaces *bestMatchFirst* by the name of the new function to be invoked.

- **Semi-automatic** decision follows a similar approach, with the main difference that the decision is most likely to be done automatically when the similarity of the considered pair of matching tuples is above a certain specified threshold *decision threshold*, while as soon as the similarity value falls under this threshold, the pairs of matching tuples have to wait for manual validation. This would be translated in a simple modification of step 2.2. in the best match first algorithm; the automatic decision is replaced by a request for user validation when the similarity is inferior to the decision threshold. In both cases, once a pair of matched tuples is declared to be valid, a decision purge is performed. This decisional mode is expressed in our language as:

```
CREATE DECISION D3
FROM M1
BY bestMatchFirst
THRESHOLD SIMILARITY > 0.9
```

- **Manual** decision is a simple version of the semi-automatic case, for which *all* the potentially matching tuples are validated by human interaction. Again, a decision purge is needed after each positive decision. The previous decision operation would then be transformed into the following:

```
CREATE DECISION D4
FROM M1
BY userCall
```

We notice that the threshold value refers to slightly different notions in the decision phase, depending on whether we are talking about a duplicate elimination or a 1:1 approximate join. In the first case, the threshold value refers to a global similarity value of a cluster and helps distinguishing the *clusters* to be treated automatically from the clusters to be analyzed by the user, while in the second case it refers to a simple similarity value and helps distinguishing the *pairs of matching tuples* with probability of being treated automatically from pairs that must be analyzed by the user.

In all the previous discussion, while talking about an approximate join between two tables R and S, the result of the decision phase will only contain tuples (r, s) which have been decided to match. Very often in the data integration process, for example in the case of data consolidation, we are interested in outer-join semantics, i.e. we are interested in keeping in

the integrated view not only pairs of tuples who *do* match, but also the tuples from R (or from S, or from both) which do *not* match with any other tuple in the other relation. Such a semantics can be easily obtained by applying a projection followed by a difference and a union, after the decision of the real matches is taken. This is expressible in native SQL, and this is why we ignore it in the process.

Finally, we would like to remark that data clustering and decision are fundamental bricks in our framework since they introduce the notion of human interaction for deciding on tuples that are not automatically chosen. Thus, we would like to summarize the possible ways to combine these two operations using the algorithms previously described. As it was already pointed out, for the case of duplicate removal, *automatic decision by cluster cohesion* always follows a *clustering operation by transitive closure*. *Clustering by diameter* is only performed for display purposes and is always followed by *manual decision*. The underlying re-organization only intends to help the user on deciding. Finally, *decision by best match first* is applied to 1:1 approximate joins and is performed directly on matching results.

2.2.5 Construction Mapping

After an approximate join operation, a construction operation needs to be specified in order to automatically build tuples in materialized tables that constitute the target integrated view. This is assured by the following mapping operation that constructs automatically CLIENT tuples from each matching tuple obtained as result of decision D2:

```
CREATE MAPPING MP2
SELECT key, name, address, d2.key1.gsm-nb, d2.key2.phone-nb, job INTO CLIENT
FROM D2 d2
LET key = keyCreation(d2.key1, d2.key2)
    sim1 = simf(d2.key1.name, d2.key2.name)
    sim2 = simf(d2.key1.street, d2.key2.street)
    SIMILARITY = min(sim1, sim2)
    name = IF (sim1 > 0.9) THEN
                RETURN d2.key2.name
    street = IF (sim2 > 0.9) THEN
                RETURN d2.key1.street
    number = first(d2.key1.number, d2.key2.number)
    pob = first(d2.key1.pob, d2.key2.pob)
    city = first(d2.key1.city, d2.key2.city)
    address = concat (number, street, pob, city)
    job = concat(d2.key1.job, d2.key2.job)
THRESHOLD SIMILARITY > 0.9
```

where D2 attributes are accessed by an implicit scan of the base tables using their keys. For example, the expression *d2.key1.name* represents the access to the attribute *name* of the GSM-CLIENT table. The correlation between the attribute *key1* and its original relation GSM-CLIENT is obtained from the metadata information stored during the entire process of data transformation.

Key attributes are generated using a unique identifier generation function; the value for the name field is obtained from the GSM-CLIENT database; the address is concatenated from the partial available information on street, number, POB and city; and job is obtained from the concatenation of the corresponding base fields. The function that concatenates the address components must be able to deal with null arguments (e.g. number or POB). The external *first* function returns the first argument which has a non-null value.

We note that such a construction operation is meant to be applied only after an approximate join operation, and not after an operation of duplicate removal. The construction of the result in the latter case can only be done manually in the current status of our framework. In fact, it seems unreasonable to choose *automatically* values for the attributes of the tuple which is meant to represent a entire cluster of duplicates in the resulting duplicate-free view. Since there is no way to distinguish tuples within a cluster, the only way in which the construction could be done is by choosing random values, and this procedure makes no sense. On the contrary, it makes absolute sense to construct automatically the tuples in the integrated view after an approximate join. The difference is that in this case, tuples from *different* sources, with different levels of trustworthiness, are merged in order to produce a unique resulting tuple. However, even for the case of approximate join it is possible to request human interaction in the extreme cases where the level of inconsistencies between certain fields is too high. As in the previous phases, a degree of consistency between the tuples that have to be merged can be calculated within the LET clause. Then, user interaction can be requested if this level is under a certain threshold. For example, the construction operator given above specifies that if the similarity of the names and addresses of a client in the two databases is under 0.9, the user has be called to choose the good name and address values.

2.3 Metadata Requirements

One of the important requirements for a data cleaning application is the possibility of explaining the data transformation process. The work published in [28] and [31] shows the fundamental role played by metadata to describe multiple integration efforts. Metadata describes attributes and the changes they undergo during integration. Reference [30] explains the use of metadata applied to the relational model where it takes the form of *meta-attributes* attached to relations and is stored in a *metadata repository* that may also be called a metadatabase or dictionary. The big advantages of such additional information are the incremental support of the integration process, the possibility of information reuse by exploiting common metadata, and the possibility of tracing the integration process [1].

In data conversions, the transformed attributes are enhanced with meta-attributes that describe their meaning and representation (e.g. units specification, data format). In general, the metadata information for each phase of a cleaning and integration process encloses the source and target schema description and a library of the transformation functions applied (for example: normalization functions for mapping operations and similarity functions for matching). To support the traceability of the process, we need also to store tuple-level

metadata. This may include the date when the information was produced, the person producing it, etc. Each attribute value must be documented with the identification of the operation that produced it. Finally, standard attribute-value pairs may be produced during the cleaning and integration and need to be also stored as reference metadata in the metadatabase.

3 Optimization Techniques

For data quality reasons, our framework is Cartesian product-based since the matching operator involves the comparison of all tuples from the source tables. We believe this is the only way to guarantee that all potential errors and matches are captured[6]. However, while doing so, a performance penalty is incurred since the Cartesian product-based semantics enhanced with external function calls is evaluated by DBMS engines through a *nested loop join* algorithm with external function calls. Matching is thus the most expensive operation of the framework, once a considerable amount of data is involved and taking into account the iterative behavior required by a data cleaning process.

The optimization of the matching operator will then be the main focus of the techniques shortly described below and presented with detail in the following sub-sections.

1 **Mixed Evaluation:** A part of the match operation is executed inside the RDBMS and another part is executed outside the RDBMS. For example, Cartesian products calling external functions are executed completely outside the RDMBS by functions that manipulate data files, because current relational databases perform this operation too slowly. This implies the need for *dumping* operations that create files from tuples and *loading* that create tables from files.

2 **Operation re-ordering:** Early evaluation of selective operations have been traditionally used to reduce the tuple cardinality. In this context, early selection can also be used to reduce the number of tuples that have to be handled during a matching. Thus, we are able to lower the cost of the underlying nested loop join with external function calls.

3 **Neighborhood Hash Join:** The nested loop evaluation can be converted into a more efficient hash based algorithm. This is the case when we have some additional knowledge of the semantics of the similarity function, provided as metadata attached to the function signature. For example, if the edit distance between two strings s_1 and s_2 is at least $\mid length(s_1) - length(s_2) \mid$, strings that differ by more than k in length need not be compared, where k is the maximum acceptable distance. This means we hash by overlapping distance lengths and compare only certain length groups.

4 **Decision Push-Down:** The similarity threshold value specified in the decision phase may be moved down in order to speed up the matching operation. In some cases, this

[6]Under the assumption that correct matching and construction criteria are given.

eliminates the need for doing certain computations. For example, Ukkonen's algorithm [34] for edit distance returns distance infinity if the difference of the lengths of the input strings is larger than k, where k is a parameter of the algorithm that corresponds to the maximum acceptable distance. The cost of the algorithm is proportional to kn where n is the length of the strings. Therefore, any information that can reduce the value of k can reduce the cost of the expensive edit operation.

5 **Short-Circuited Computation:** One way to eliminate external function calls altogether is by noticing what is going to be done with the results of those computations. For example, if the goal of a computation is to determine whether a triple (x, y, z) is pair-wise similar above a threshold Ths, then the triple can be rejected if x and y by themselves do not reach the threshold. This eliminates the need to compare y and z as well as x and z. Analogously, the work by Levy and Mumick [21] presents a framework for reasoning with aggregation constraints, namely they infer predicates that can be pushed down through aggregation.

6 **Cached Computation:** It is helpful to avoid unnecessary calls to external functions, because they are often expensive. For example, if an external function is deterministic, then two calls to that function having the same arguments will yield the same result. Therefore, we cache the arguments and results of each external function call as in [11]. When preparing to call a function, we could first check whether we had already computed the answer.

7 **Parallel computation:** The parallelization technique used for *hash joins* can also be used to parallelize the execution of the matching operator, if the *neighborhood hash join execution* optimization described above has been applied. Even if our edit function does not permit the use of hash joins, parallelism will help using the obvious technique of partitioning one of the source tables and replicating the other tables across the sites. The results obtained are then merged to produce the final matching result.

We describe optimizations 1, 3, 4, and 5 in this section, since these are unique to our problem and are partly novel. We would like to remark that optimizations like Neighborhood Hash Join or Decision Push-Down exploit particular characteristics of often used similarity functions [20].

3.1 Mixed Evaluation

Our optimization approach takes advantage of the optimizations supported by the underlying SQL execution engine. However, some DBMSs perform very badly in the presence of a nested loop join with external function calls. For example, executing the entire matching operation inside the DBMS might be prohibitively expensive since the cost of calling an external function from an SQL statement that performs a Cartesian product can be very high (e.g. Oracle 8 calling external C functions implies a call to a PL/SQL function that then calls

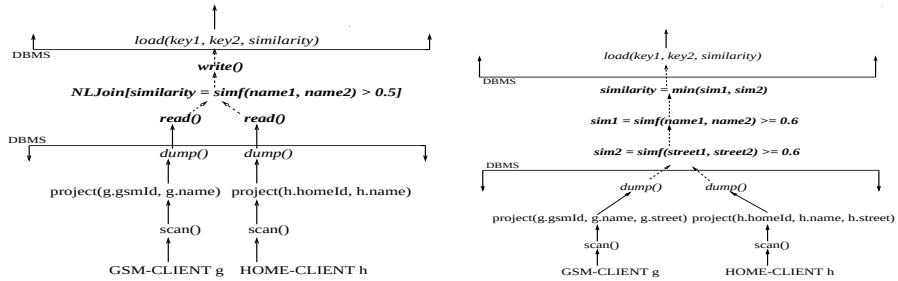

Figure 3: Left: Mixed evaluation, Right: Partial evaluation of similarity functions

```
Input:  k (the maximum acceptable distance)
{
    let S1 be the smaller of the two sets of strings (in number of bytes)
    and let S2 be the larger of the two sets
    if S1 fits into memory then {
        partition (in memory) the set S1 according to the length of strings
        compare each element s2 ∈ S2 with each element in the partitions of S1 of length l
            where | length(s2) − l | ≤ k
        }
    else {
        partition S1 according to the length of strings
        partition S2 according to the length of strings
        for i = smallest length of S1 to largest length of S1 {
            read the partitions of S2 containing strings of length i − k, ..., i, ..., i + k
                that are not already in memory
            compare each element in S1 of length i with each element in the above partitions
        }
}
```

Figure 4: Generalized neighborhood hash join algorithm

the C function[7]). Hence, only part of the execution is done inside the DBMS. The nested loop join with external function calls is completely executed outside the DBMS by programs that handle data files. The left side of figure 3 represents one possible query execution plan implementing the matching operation between GSM-CLIENT and HOME-CLIENT tables..

[7]To illustrate the magnitude of response times obtained in Oracle, version 8 takes 0.68 seconds to execute a simple external C function that compares two strings. This test was done on a Sun UltraSPARC with 143 MHz, 384M of physical memory, 1.2G of virtual memory and running SunOS release 5.5. Other DBMSs may have a better behavior, however we believe that the overhead of calling external functions remains a bottleneck.

3.2 Neighborhood Hash Join

Often we use a distance metric as a measure of similarity. The lower the distance, the greater the similarity. Further, we are often uninterested in strings whose distance is greater than a certain threshold t. In the case where some additional knowledge about the distance metric used is provided, the nested loop evaluation can be replaced by a special form of approximate hash join called a *neighborhood hash join.*

Suppose that the distance function obeys the following property: there exists a function $f()$ such that for every pair (s_1, s_2), the following inequality holds: $Distance(s_1, s_2) \geq | f(s_1) - f(s_2) |$. For example, $f(s_1)$ may be the length of s_1. Then, the two input sets of strings can be hashed according to the result of the application of function $f()$. In that case, bucket x consists of those strings s such that $f(s) = x$. In a neighborhood hash, the strings in bucket x need only be compared to buckets y where $(x - t) <= y <= (x + t)$. When f is the length as it is for the edit distance[33], neighborhood hashing entails bucketing the input strings by their length. Suppose that an application is interested in edit distance with a threshold $t = 1$. Then strings of length 10 may potentially match strings of length 9, 10, or 11. Similarly, strings of length 9 may potentially match strings of length 8, 9, or 10. Another example of a function $f()$ for which this property also holds for an edit distance function is the one that returns the number of occurrences of a particular character within a string.

The Neighborhood Hash Join algorithm is presented in figure 4. When big input files of strings have to be handled, only the partitions to be compared are loaded into memory. Note that the second *for* loop economizes on input/output because one partition of S_2 may be used for $2k$ partitions of S_1.

Finally, we note that the standard hash join between a table T_1 and a table T_2 can replace the nested loop evaluation in classical situations, such as when the WHERE clause explicitly includes a predicate of the form $f(t_1.attribute) = f(t_2.attribute)$. We claim that in data cleaning applications, this kind of predicate appear frequently, since they correspond to intuitive heuristics in comparing strings. For example, a filter predicate comparing the first letter of the two strings is used frequently in practice, justified by the fact that data entry errors in the first letter are quite uncommon.

3.3 Decision Push-Down

When a semi-automatic decision by "best match first" follows a matching operation, the corresponding decision threshold may be pushed down to the matching. The optimization of the pairwise best match first algorithm takes place as follows.

Taking into account the relationship between similarity and distance previously given in section 2, suppose we have two table sources T_1 and T_2 with tuples $tuple_1$ and $tuple_2$, respectively. The function that compares the two tuples, $distFunc(tuple_1, tuple_2, thresholdDist)$ returns the edit distance between string fields $s_1 \in tuple_1$ and $s_2 \in tuple_2$ if

$editDistance(s_1, s_2) \leq thresholdDist$ and ∞, otherwise. Assume that the time to compute $distFunct()$ is monotonically increasing with *thresholdDist*, e.g. $min(\mid s_1 \mid, \mid s_2 \mid).(2thresholdDist + 1)$ as it is for the Ukkonen algorithm [34]. Suppose that $matchDist$ is the maximum distance for acceptance in matching (corresponds to the matching similarity threshold) and $decisionDist$ is the maximum allowed distance for decision (corresponds to the decision similarity threshold). Assume also that $decisionDist < matchDist$.

The algorithm given in figure 5 describes one way to evaluate the decision phase concurrently with the match phase, such that the total time of the two phases is minimized. To explain this precisely, we require a few definitions:

- Let t correspond to a matching tuple with tuple components identified by $t[j]$. Let also $t[j].str$ correspond to the field str of the jth component tuple of the matching tuple t.
- Define an *edge* between two pairs of matching tuples t_1 and t_2 as: $E = \{(t_1, t_2) \mid t_1[1] = t_2[1] \ \vee \ t_1[2] = t_2[2]\}$, where $t_i[j]$ stands for the identifier of the jth tuple component of the matching tuple i. If $t_1 \ E \ t_2$, then t_1 and t_2 are said to be *match neighbors*.
- Two matching tuples are said to be *tied* if there is an edge among them and if their distances are equal: $tied = \{(t_1, t_2) \mid t_1 \ E \ t_2 \wedge distFunc(t_1[1].str, t_1[2].str, thresholdDist) = distFunc(t_2[1].str, t_2[2].str, thresholdDist)\}$[8]. A matching tuple that is in some tied pair is called a *tie*.
- An equivalence class of ties is the transitive closure of the ties and is formally represented by: $eqtie = \{(t_1, t_2) \mid (tie(t_1, t_2)) \vee (\exists t_3, eqtie(t_3, t_2) \wedge tie(t_1, t_3))\}$.
- When it is decided that a matching tuple represents a valid match, then the tuple components of the matching tuples are said to be *married*. If the allowed marriages are 1:1, they are said to be *monogamous marriages*; if 1:n, then *polygamous marriages*; and if m:n then *group marriages*.

We begin by allowing only monogamous marriages. This is appropriate when each input database has no duplicates. The algorithm has two steps. In the first step we calculate three sets whose intuitive meaning is the following:

- The set *Good* contains the matching tuples with distance below $decisionDist$ and thus most likely to lead to marriage;
- The set $NotGoodEnough$ consists of matching tuples that are neighbors to elements in *Good* but are less likely to be married;
- And the set *Accept* are matching tuples that have to be analyzed by the user because their distance is between the decision and the match thresholds and they have no neighbors in *Good*.

[8] For the sake of the exposition, the distance between two component tuples is based on the comparison of a single field *str*. In a general case, it can involve the comparison of more than one field.

```
Input:      0 ≤ decisionDist ≤ matchDist
Output:  monogamous marriages
{
Step 1:
  1     Good={}
  2     Accept={}
  3     for each t : (t[1], t[2]) ∈ T1 × T2 that could be within matchDist {
            /* find all occurrences of t' in Good s.t.  t and t' have an edge between them */
  4         AlreadyGood = {t' : (t'[1], t'[2], dist) ∈ Good | t E t'}
  5         if AlreadyGood ≠ {}, then {
  6             minDistAlready = min({dist | t' : (t'[1], t'[2], dist) ∈ AlreadyGood})
                /* calculate the distance between t[1] and t[2], with a new
                    constraint minDistAlready which always satisfies minDistAlready ≤ decisionDist */
  7             newDist = distFunc(t[1].str, t[2].str, minDistAlready)
  8             if newDist ≤ minDistAlready then add (t[1], t[2], newDist) to Good
  9             else add (t[1], t[2]) to NotGoodEnough
              }
  10        else {
  11            newDist = distFunc(t[1].str, t[2].str, matchDist)
  12            if newDist ≤ decisionDist then add (t[1], t[2], newDist) to Good
  13            else if decisionDist < newDist ≤ matchDist then add (t[1], t[2], newDist) to Accept
              }
        }
Step 2:
  14    sort Good and Accept by increasing distance
  15    within each distance, cluster the tuples that belong to the same equivalence class within eqTies
  16    iterate over Good list {
  17        let t : (t[1], t[2], dist) be the current matching record in Good declared to be married
  18            optionally show to the user the following for validation:
  19                purge from Good, Accept and NotGoodEnough all tuples p s.t.  p E t
  20                for each purged tuple p from Good do
  21                    bringBackNotGoodEnough(p)
        }
  22    iterate over Accept list {
  23        let t : (t[1], t[2], dist) be the current matching record
  24        user analyses (t[1], t[2], dist)
  25        if the user marries t then remove from Accept all matching records a st a E t
        }

  26    bringBackNotGoodEnough(p) {
  27        if NotGoodEnough ≠ {}, then {
  28            for each e in NotGoodEnough such that e E p do {
  29                if not (e E g) with g in Good then {
  30                    newDist = distFunc(e[1].str, e[2].str, matchDist))
  31                    if newDist ≤ decisionDist then add (e[1], e[2], newDist) to Good
  32                    else if decisionDist < newDist ≤ matchDist then add (e[1], e[2], newDist) to Accept
  33                    re-sort Good and Accept by increasing distance
                    }
                }
            }
        }
}
```

Figure 5: Optimization of best match first algorithm by pushing down the decision threshold

These sets may change in the course of executing step 2. The following tasks constitute step 2:

1. Ties among the *Good* tuples are shown to the user. Other *Good* tuples are married automatically.
2. After each marriage of a matching tuple m, the following tuples are removed from *Good*: $\{t \mid t\ E\ m \wedge t\ in\ Good\}$. Also, the following are removed from *Accept*: $\{t \mid t\ E\ m \wedge t\ in\ Accept\}$. (The user may optionally be shown the result of this deletion operation and validate it.)
3. A tuple t in *NotGoodEnough* may be moved into *Good*. This will occur when all *Good* neighbors of t that had higher scores have been deleted and no neighbor of t is married. Notice that finding neighbors can be done efficiently with hash indexes on *Good* (and similarly for *NotGoodEnough*) such that *hashFunction*: component tuple $\rightarrow$ matching tuple id in *Good*. Such a data structure enables us to discover neighbors in expected constant time.

This algorithm would have to be extended if the user wants to be called to analyze sets of matching tuples that have an edge between them and whose similarities fall into a certain neighborhood interval. In such a case, *ties* are extended to be those tuples obeying: $tieNeighborhood(t_1, t_2) = \{(t_1, t_2) \mid t_1 E t_2 \wedge \mid dist(t_1[1], t_1[2]) - dist(t_2[1], t_2[2]) \mid \leq neighborhood\}$. In addition, the test in line *8* of the algorithm is replaced by: if $newDist \leq minDistAlready + neighborhood$ then add $(t[1], t[2], newDist)$ to *Good*.

Example 3.1: The following example is based on tables GSM-CLIENT and HOME-CLIENT. The matching operation involves the comparison of names from both tables. The acceptance distance is 3 and the decision distance is 2. Assume the input to the match phase is constituted by the following tuples (we are just mentioning the key and name attributes):

GSM-CLIENT
(1, "smith j")
(2, "mitch")
HOME-CLIENT
(1248, "mitcheli")
(1247, "smith")
(1245, "smith jr")
(1246, "mitch jr")

Executing step 1 of the algorithm results in the following:

i) **pair (1, 1248)**: *AlreadyGood* is empty. *distFunc("smith j", "mitcheli", 3)* is evaluated with the acceptance distance and returns ∞ (its actual distance is 6, but we don't care). Pair $(1, 1248)$ is discarded.

ii) **pair (1, 1247)**: *AlreadyGood* is empty. *distFunc("smith j", "smith", 3) = 2* which is equal to the decision distance. So, tuple $(1, 1247, 2)$ is inserted into *Good*.

iii) **pair (1, 1245)**: since tuple 1 already belongs to *AlreadyGood*, *AlreadyGood* is not empty and *minDistAlready* equals 2. The edit distance for a maximum decision distance of 2 is computed: *distFunc("smith*

j", "smith jr", 2). It returns a distance equal to 1. Since 1 is less than or equal to *minDistAlready*, tuple $(1, 1245, 1)$ is inserted into Good.

iv) **pair (1, 1246)**: tuple 1 already belongs to *Good*, so only check the edit distance for a decision distance of 1 (*minDistAlready*). Compute: *distFunc("smith j", "mitch jr", 1)* which returns ∞ (3 is the real distance, but we don't calculate it). *MinDistAlready* equals 1, so tuple $(1, 1246)$ is put into *NotGoodEnough*.

v) **pair (2, 1248)**: neither tuple 2 nor tuple 1248 belong to the *Good* table, so compute the distance function *distFunc("mitch", "mitcheli", 3)*. The distance returned is 3 which is greater than the decision distance but equals the acceptance distance. Tuple $(2, 1248, 3)$ is inserted into the *Accept* table.

vi) **pair (2, 1247)**: tuple 1247 belongs to the *Good* table. *MinDistAlready* is 2, so compute the distance function *distFunc("mitch", "smith", 2)*. The distance returned is 2 that equals *minDistAlready*. Tuple $(2, 1247, 2)$ is inserted into table *Good*.

vii) **pair (2, 1245)**: tuple 1245 already belongs to the *Good* table and *minDistAlready* equals 1. So, we compute the distance function as *distFunc("mitch", "smith jr", 1)*. It returns ∞ that is superior to *minDistAlready*. It is then added to it NotGoodEnough.

viii) **pair (2, 1246)**: tuple 2 belongs to *Good*. *MinDistAlready* is 2. So, compute the distance function as: *distFunc("mitch", "mitch jr", 2)*, using the initial decision distance. It returns ∞ which is superior to *minDistAlready* and tuple $(2, 1246)$ is put into *NotGoodEnough*.

The following table summarizes the distances obtained and the table (*Good* or *Accept* to which they belong after step 1 of the algorithm:

GSM-CLIENT	HOME-CLIENT	distance	table
1	1248	6	-
1	1247	2	Good
1	1245	1	Good
1	1246	∞	NotGoodEnough
2	1248	3	Accept
2	1247	2	Good
2	1245	∞	NotGoodEnough
2	1246	∞	NotGoodEnough

According to the step 2 of the algorithm, the following is executed:

1. The table *Good* is ordered by increasing distance, as follows.

   (1, 1245, 1)
   (1, 1247, 2)
   (2, 1247, 2)

   First, matching record $(1, 1245, 1)$ is married. As a result, the record $(1, 1247, 2)$ is purged from table *Good* and the table *NotGoodEnough* is checked. Tuple (1, 1246) could be brought back but it is not since there exists an edge between it and the married tuple: $(1, 1245, 1)$. Next, tuple $(2, 1247, 2)$ from *Good* is declared a best match and and record $(2, 1248, 2)$ is purged from the table *Accept*.

2. No tuples are shown to the user.

The main benefit of this optimization is that it decreases the distance threshold for some distance calculations. As a result, the edit distance function becomes less expensive in many cases. The following approximate analysis presents a brief study of the computation times involved in the decision push-down optimization.

3.3.1 Approximate Analysis of the Decision Push-Down Optimization

We use the following additional notation: L is the size of the string fields being compared, making the pessimistic assumption that they all have the same length; N is the number of tuples put in $NotGoodEnough$.

1 Step 1: The computation saved during the first step of the decision push-down optimization is at least:

$N \cdot 2 \cdot (matchDist - decisionDist) \cdot L$.

We may end up by saving more than this since: *(i)* we also save time for tuples that are put into *Good* and already have neighbors in *Good*, *(ii)* it may be possible to invoke the edit distance function with a maximum allowed distance less than *decisionDist* as parameter, when $minDistAlready \leq decisionDist$.

2 Step 2: In the worst case, all $NotEnoughGood$ matching tuples are brought back and their distance is re-computed. If this is the case, the extra time consumed is: $N \cdot 2 \cdot (matchDist + 1) \cdot L$. The overhead of using this optimization, when comparing with 1. would then be: $N \cdot 2 \cdot (decisionDist + 1) \cdot L$.

According to point 2, a very bad scenario for which the decision push-down optimization would not work well is the following. Suppose a *Good* matching tuple $t : (t[1], t[2])$ is married. The *Good* table contains all matching tuples $g : (g[1], g[2])$ where $g[1] = t[1]$ and $g[2]$ can take all possible values of T_2 and $g : (g[1], g[2])$ where $g[2] = t[2]$ and $g[1]$ can take all possible values of T_1, i.e. all the neighbors of t. These ones are purged once t is married. As a consequence, all matching tuples e in $NotGoodEnough$ such that $e\ E\ g$ but not $e\ E\ t$, for all values of $e[1]$ *in* T_1 and $e[2]$ *in* T_2, are brought back and the edit distance function is re-invoked.

This is unlikely to happen since in a realistic situation we do not expect too many Good matches that are neighbors. Therefore, purging does not affect a large number of *Good* tuples and the tuples in $NotGoodEnough$ are seldom brought back.

3.3.2 Polygamous and Group Marriages of Matching Tuples

The above algorithm applies for monogamous marriages, i.e. one tuple from a first table should be married to *at most* with one tuple from a second table. Some modifications have to be introduced when a semantics of *1:m* is allowed for best match first decision (polygamy), which means that a tuple from one table can match more than one tuple from the other but the inverse does not happen. Reconsidering the example of the Telecom company, such a match semantics corresponds to the situation where one HOME-CLIENT may have several cellular phones but one cellular phone number corresponds at most to a single home phone number. In this case, an edge between two tuples t_1 and t_2 is less restrictive than before and we call it *polygamous edge*. The formal definition is the following:

$PE = \{(t_1, t_2) \mid t_1[i] = t_2[i]\}$ where i identifies the table whose tuples can marry only one other component tuple (the women in a polygamous society; the men in a polyandrous society). The definitions of *AlreadyGood* and *minDistAlready* remain the same except that each reference to edge is replaced by polygamous edge.

There is no need of a *NotGoodEnough* set but we still need an *Accept* table. The purging phase in step 2 changes and the modified algorithm for decision push-down optimization is shown in figure 6. Note that ties are still possible and the equivalence classes of polygamous ties are defined according to polygamous edges.

Group marriages may arise when a tuple from any of the tables is allowed to marry multiple tuples of the other table. For instance, one internet client may marry to several cellular phone records and vice-versa. The algorithm in this case does not take into account the existence of ties since every equally strong enough marriage is accepted. There is no need for the *NotGoodEnoughTable* and we cannot take advantage of decreasing decision thresholds for computing less expensive edit distance functions. Consequently, there is no optimization. The algorithm works as represented in figure 7.

3.4 Short-Circuited Computation

Compilers use short-circuiting to stop the evaluation of a boolean expression after its outcome is known. For example, if the boolean consists of a conjunctive expression, then as soon as one conjunct is discovered to be false, further evaluation can be halted. The optimizations described here are closely related to short-circuiting.

3.4.1 Short-Circuiting Based on Multi-Way Similarity

Consider the following example of a matching operator:

```
MATCH M3
FROM GSM-CLIENT g, HOME-CLIENT h
LET sim1 = simf(g.name, h.name),
    sim2 = simf(g.street, h.street)
    SIMILARITY = min(sim1, sim2)
THRESHOLD SIMILARITY >= 0.6
```

If the similarity threshold (0.6) is considered when computing *sim1* and *sim2*, it is enough that one of these variables is assigned a value smaller than 0.6 to eliminate the corresponding tuple. Those tuples will never belong to the matches returned by M3 because they do not satisfy the condition on the minimum allowed similarity. The execution plan for M3 is represented on the right of figure 3. Only those pairs of tuples from GSM-CLIENT and HOME-CLIENT that satisfy the condition $sim1 \geq 0.6 \wedge sim2 \geq 0.6$ give rise to matches. The optimization based on this fact is to bypass one of the partial similarity function calls (that assigns a value to *sim1* or to *sim2*) once the other one returns a value less than 0.6.

```
Input:      0 ≤ decisionDist ≤ matchDist
Output:  polygamous marriages
{
Step 1:
    Good={}
    Accept={}
    for each t : (t[1], t[2]) ∈ T1 × T2 that could be within matchDist {
        /* find all occurrences of t' in Good s.t.  t and t' have a polygamous edge between them */
        AlreadyGood = {t' : (t'[1], t'[2], dist) ∈ Good | t PE t'}
        if AlreadyGood ≠ {}, then {
            minDistAlready = min({dist | t' : (t'[1], t'[2], dist) ∈ AlreadyGood})
            /* calculate the distance between t[1] and t[2], with a new constraint minDistAlready
                which always satisfies minDistAlready ≤ decisionDist */
            newDist = distFunc(t[1].str, t[2].str, minDistAlready)
            if newDist ≤ minDistAlready then add (t[1], t[2], newDist) to Good
        else {
            newDist = distFunc(t[1].str, t[2].str, matchDist)
            if newDist ≤ decisionDist then add (t[1], t[2], newDist) to Good
            else if decisionDist < newDist ≤ matchDist then add (t[1], t[2], newDist) to Accept
        }
    }
Step 2:
    sort Good and Accept by increasing distance
    within each distance, cluster the tuples that belong to the same equivalence class within eqPolygamousTies
    iterate over Good list {
        let t : (t[1], t[2], dist) be the current matching record in Good declared to be married
            optionally show to the user the following for validation:
                purge from Good and Accept all tuples p s.t.  p PE t
    }
    iterate over Accept list {
        let t : (t[1], t[2], dist) be the current matching record
        user analyses (t[1], t[2], dist)
        if the user marries t then remove from Accept all matching records a s.t.  a PE t
    }
}
```

Figure 6: Optimization of best match first algorithm by pushing down the decision threshold for *polygamous marriages*

In general, once the total similarity of a matching is obtained by applying the *min*, *max*, or by computing the product of the partial similarities, the appearance threshold can be pushed down to the computation of partial similarities thus avoiding some of them.

3.4.2 Short-Circuiting Based on Clustering Threshold

A similar observation works for clustering. Assume the following matching operator, followed by a clustering by diameter.

```
Input:      0 ≤ decisionDist ≤ matchDist
Output:  group marriages
{
Step 1:
    Good={}
    Accept={}
    for each t : (t[1], t[2]) ∈ T1 × T2 that could be within matchDist {
        newDist = distFunc(t[1].str, t[2].str, matchDist)
        if newDist ≤ decisionDist then add (t[1], t[2], newDist) to Good
        else if decisionDist < newDist ≤ matchDist then add (t[1], t[2], newDist) to Accept
    }
Step 2:
    sort Good and Accept by increasing distance
    iterate over Good list{
        let t : (t[1], t[2], dist) be the current matching record in Good declared to be married
    }
    iterate over Accept list {
        let t : (t[1], t[2], dist) be the current matching record
        user analyses (t[1], t[2], dist)
    }

}
```

Figure 7: Best match first algorithm by pushing down the decision threshold for *group marriages*

```
MATCH M4
FROM GSM-CLIENT g1, GSM-CLIENT g2
LET sim1 = simf(g.name, g.name),
    sim2 = simf(g.street, g.street)
    SIMILARITY = min(sim1, sim2)
THRESHOLD SIMILARITY > 0.6

CLUSTERING C3
FROM M4
LET GSIMILARITY = min(similarity)
BY diameter
THRESHOLD GSIMILARITY > 0.7
```

The clustering option *by diameter* and the cluster threshold similarity require that all matching tuples resulting from M4 to have a similarity value greater than 0.7 in order to be clustered. Assuming again that the similarity functions applied among tuples are based on edit distance, the triangle inequality [19] guarantees that for three matching tuples t_1, t_2 and t_3 with pairwise distance given by d_{12}, d_{23} and d_{13}, $d_{13} \leq d_{12} + d_{23}$.

Let us consider the relationship between similarity and distance defined before. By performing the corresponding substitutions, the following inequality is verified: $sim_{13} \geq sim_{12} + sim_{23} - 1$. Applying this result to the sequence of operations M5 and C3, let us

suppose that two matching tuples returned by M5 are: $(gKey1, gKey2, similarity_1)$ and $(gKey2, gKey3, similarity_2)$. If $similarity_1 + similarity_2 - 1$ is superior to 0.7, the similarity between tuple $gKey1$ and $gKey3$ is guaranteed to be also superior to 0.7. So, tuples identified by $gKey1$ and $gKey3$ need not be compared and all three $gKey1$, $gKey2$ and $gKey3$ are put into the same cluster.

4 Experimental Results

This section presents the results of a set of experiments whose goal was to evaluate some of the optimization techniques proposed in the last section. In these experiments we used one of the databases created for the validation of the WHIRL system [7]. It consists of two tables T_1 and T_2, both containing information about existing companies (e.g. name, url, industry). The operation we tried to optimize is an approximate join between the two tables, followed by a decision operation by best match first. The similarity between tuples is computed by an edit distance function, which is implemented using the Smith-Waterman and Ukkonen algorithms. In order to obtain a large volume of data we replicated the original database a certain number of times. Then, a randomized error was introduced in each handled field in order to avoid exact duplicates. We tested our evaluation strategies with three scale factors: 1,000, 10,000 and 100,000 tuples in each table. The size of the resulting files is 50K, 500K and 5M respectively .

Since the overhead of calling an external function from Oracle is extremely high, we did not consider the evaluation of the match operator within the DBMS as an interesting alternative. Hence, we ignored the DBMS and in all the experiments we manipulated files using C programs. For each experiment we measured the execution time of the program without taking into account the time for dumping the Oracle database into files[9], loading the data from files into tables, reading data from a file, or the time for writing the results into a file. Our experiments have been done on an INTEL Pentium II with 400Mhz and 128MBytes running Linux RedHat 6.0. We used Oracle version 8.

Our optimization target is the evaluation of a match and a decision between tables T_1 and T_2, specified in our language by the following sequence of operators:

```
MATCH M
FROM T1 t1, T2 t2
LET SIMILARITY = 1 - editDist(t1.company,t2.company)/1000
THRESHOLD SIMILARITY >= matchThreshold;

DECISION D
FROM M
BY bestMatchFirst
THRESHOLD SIMILARITY >= decisionThreshold
```

[9]The time to dump 100 000 tuples composed by $(key, field)$ from Oracle to files is around 1 min. The time spent for loading the file back into Oracle is 2 min.

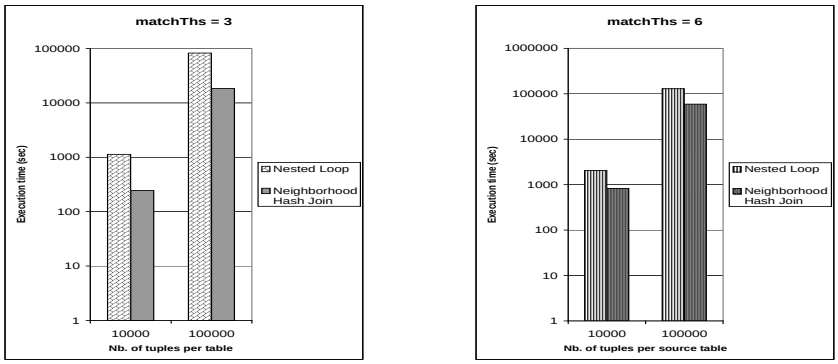

Figure 8: Nested Loop vs Neighborhood Hash Join Left: matchThs = 3, Right: matchThs = 6

The naive way of evaluating this sequence is to perform a nested loop, and compute, for each tuple in the Cartesian product the similarity value; then to execute the decision phase by iterating over the result of the match and deciding which are the correct matches.

As said earlier, there is a list of possible optimizations that we can apply in order to improve the performance of this naive evaluation. We observe that there are two dimensions associated with the execution of this sequence: first, the method used to find the matching records, and second, whether we execute the matching and decision phases separately or merge them. According to the first dimension, we have the choice between a nested loop evaluation and a neighborhood hash join, as explained in section 3.2. According to the second dimension, we have the choice between either executing the decision after the entire matching phase has finished (*Naive best match first*) as defined by the semantics, or merge the decision phase with the matching phase (*Decision push-down*), as described in section 3.3. By varying the possibilities along these two dimensions we obtain four evaluation strategies: (1) Nested Loop (NL), (2) Nested Loop and Decision Push Down (NL + DPD), (3) Neighborhood Hash Join (NHJ), and (4) Neighborhood Hash Join and Decision Push Down (NHJ + DPD).

The execution times for strategies (1) and (3) are presented in figure 8 (with a logarithmic scale for the y axis), where the company names are hashed by their length in the neighborhood hash join algorithm. As expected, the matching performance is improved when the hash join is applied. Note that the gain for a match threshold of 3 (around 80%) is greater than the gain for a match threshold of 6 (on average 55%). This is due to the fact that the number of partitions to be compared is smaller.

Figure 9 shows the results obtained when the four strategies are considered. We measured the execution time for step 1 of the decision push-down optimization for 1:1 marriage

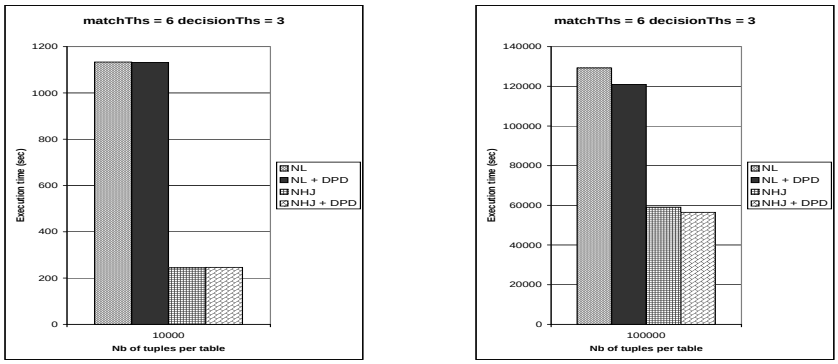

Figure 9: Naive Best Match First versus Decision Push-Down Left: 10,000 tuples, Right: 100,000 tuples

semantics. As in the previous case, the gain obtained when hashing is applied is still significant. However, the improvement achieved by the application of the decision push-down optimization is not significant for 10,000 tuples, match threshold of 6 and decision threshold of 3. The reason is the following. This optimization is advantageous when the Good table (see the description of the algorithm in 3.3) is filled in early in the process. This implies that the distance of some latter pairs of strings can be evaluated through a less expensive edit distance function. In this particular case, the number of tuples is not large enough to raise this situation. When considering 100,000 tuples and the same thresholds, gains of 6% and 11% are obtained. We should remark that the gain obtained with this optimization depends largely on the data itself and on the threshold values. Figure 10 shows the four dimensions for a match threshold of 3 and a decision threshold of 1. Even though the number of tuples is 100,000, no gain is obtained by using the decision push-down optimization in this case. This is expected since the decision threshold of 1 does not permit the execution of the algorithm to take advantage of calling a less expensive edit distance function.

We would like to remark that an additional improvement of the decision push-down optimization can be obtained when the neighborhood hash join is used. This can be achieved by comparing the partitions following a specific order. First, the buckets that correspond to strings with equal length are compared, then those with a difference of lengths equal to one and so forth. This way, the probability of finding first those matches that are more similar is higher. The net result is that the Good table is filled in as early as possible. Furthermore, once a match with distance d is found where d is equal to the difference of lengths of the current partitions, then there is no need to continue comparing buckets with a difference of lengths greater than d. The gains obtained when comparing with neighborhood hash

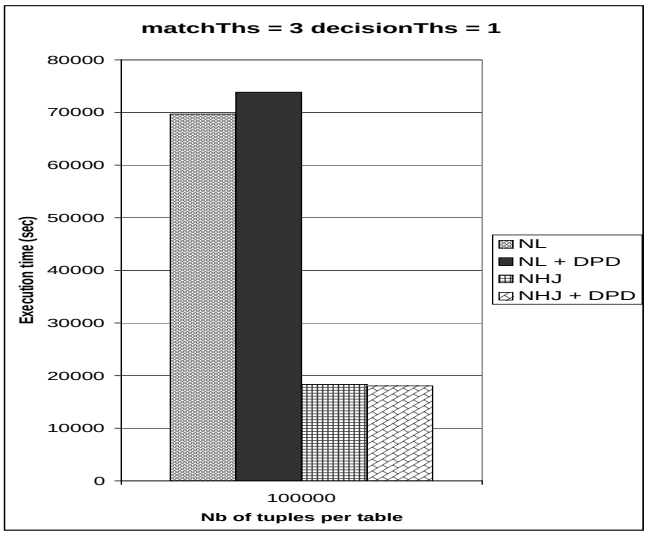

Figure 10: Naive Best Match First versus Decision Push-Down with matchThs = 3 and decisionThs = 1

join with decision optimization were around 13% for a match threshold of 6 and decision threshold of 3.

A second set of experiments was done using real data. We used the INRIA suppliers database in particular the table of supplier organizations that contains approximate and exact duplicates. The total number of tuples is 3,500. The goal was to apply a matching operator to this table in order to find pairs of duplicates. We measured the execution time as well as the quality of results obtained for two evaluation startegies of the matching operator: (i) when using hashing on the first letter of organization names as described in 3.2 and (ii) when using a nested loop algorithm. The matching operation with a predicate specifying the first letter filter is described as follows:

MATCH M
FROM Organizations o_1, Organizations o_2
LET SIMILARITY = 1 - editDist(o_1.nomOrg,o_2.nomOrg)/1000
WHERE firstLetter(o_1.nomOrg) = firstLetter(o_2.nomOrg)
THRESHOLD SIMILARITY >= $matchThreshold$;

When using the filter, organization names are hashed according to the first letter and only partitions that correspond to the same first letter are compared through the edit distance

function. The time of execution obtained was 8 sec for this case. The naive nested loop evaluation was 10 times longer (81 sec). The percentage of matches lost by the first letter filter evaluation was around 3%.

The success of the application of this kind of filter depends of course of the domain of application. In fact, some of the matches found by the naive approach and lost by the filter can indeed be false matches and do not correspond to the same object.

5 Conclusions and On-going Work

In this paper, we addressed the problem of data cleaning, which entails three major problems: object identity problem, data entry errors, and data inconsistencies across overlapping autonomous databases. We proposed a new framework that consists of an open and extensible system platform, in which a data cleaning application is specified as a directed acyclic graph of data transformation phases. Each phase is specified by means of one of the three data cleaning services offered by the system platform: data transformation, duplicate-elimination, and multi-table matching. Each service is supported by a shared kernel of four macro-operators consisting of high-level SQL-like commands. The system platform is open in the sense that new macro-operators can be defined and added to the kernel, and new services can also be added based on the set of macro-operators.

The language composed of our four macro-operators has several features. First, it is more expressive than previously proposed languages for multi-table matching, or duplicate-elimination. In particular, the Cartesian product-based semantics of the matching macro-operator, contributes significantly to the expressiveness of the language, by guaranteeing that all potential errors and correct matches are captured. Second, it enables to specify all the steps required by a data cleaning process in a high-level manner. For instance, the Telecom example was coded using around 100 lines of code in our language as is presented in annex. We also experienced this fact with three larger applications: removing duplicates from the list of Inria suppliers, removing duplicates in biological bibliography references of the Pasteur Institute, and consolidation of a bibtex file using the DBLP database. Last, our language is extensible through the use of external functions coded in conventional programming languages.

An extension of SQL was proposed rather than using an existing language as SQL/PSM [18], as proposed by the Cohera database federation system [12]. First, it allows a *compact* specification of each cleaning process. As we showed, there is a syntax for each macro-operator that encloses its main functionalities like the *if-then-else* constructs for expressing decision trees and the *THRESHOLD* clause for imposing conditions on values of internal variables. Second, we obtained a *uniform* specification for all operators and introduced new ones as decision and clustering. And lastly, some of the language primitives permit the application of *optimization* techniques as pushing down thresholds.

We proposed novel optimization techniques to efficiently execute matching operations. A first technique is to execute macro-operators using both a relational database system and a

specifically tailored execution engine. A second technique is to exploit semantic knowledge of similarity functions, combined with hash join algorithms to dramatically reduce the number of records that have to be compared in approximate joins. Last is the push-down of decision operations within matching operations to avoid unnecessary accesses to matching records. First experiments conducted on a test database of varying size showed that these three optimization techniques bring a substantial performance improvement as the database gets large.

Work in progress encloses the study of classical edit distance functions in order to come up with properties that can be used for optimization purposes. We are doing some research on signature functions and *n-grams* algorithms. A more detailed study on the metadata to be stored in the process of cleaning is being done based on the notion of propagation rules [27]. Finally, our approach should be intensively experimented with real data that needs to be cleaned.

References

[1] P. Bernstein and T. Bergstraesser. Meta-data support for data transformations using Microsoft Repository. *IEEE Data Engineering Bulletin*, 22(1), March 1999.

[2] D. Bonachea, K. Fisher, and A. Rogers. Hancock: A language for describing signatures. In *USENIX*, 1999.

[3] M. Bouzeghoub, F. Fabret, H. Galhardas, M. Matulovic, J. Pereira, and E. Simon. *Fundamentals of Data Warehousing*, chapter Data Warehouse Refreshment. Springer-Verlag, 1999.

[4] D. Calvanese, G. De Giacomo, M. Lenzerini, D. Nardi, and R. Rosati. A principled approach to data integration and reconciliation in data warehousing. submitted to publication, 1999.

[5] K. Claypool and E. Rundensteiner. Flexible database transformations: the SERF approach. *IEEE Data Engineering Bulletin*, 22(1), March 1999.

[6] S. Cluet, C. Delobel, J. Siméon, and K. Smaga. Your Mediators Need Data Conversion. In *Proc. of ACM SIGMOD Conf. on Data Management*, 1998.

[7] W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *Proc. of ACM SIGMOD Conf. on Data Management*, 1998.

[8] W. Cohen. Some practical observations on integration of Web information. In *WebDB'99 Workshop in conj. with ACM SIGMOD*, 1999.

[9] U. Dayal. Processing queries over generalization hierarchies in a multidatabase system. In *Proc. of the Int. Conf. on Very Large Databases*, 1983.

[10] EDD. Home page of Data Cleanser tool. http://www.npsa.com/edd/.

[11] J. M. Hellerstein. Optimization techniques for queries with expensive methods. *ACM Trans. on Database Systems*, 23(2):113–157, 1998.

[12] J. M. Hellerstein, M. Stonebraker, and R. Caccia. Independent, open enterprise data integration. *IEEE Data Engineering Bulletin*, 22(1), March 1999.

[13] M. A. Hernandez and S. J. Stolfo. The Merge/Purge problem for large databases. In *Proc. of ACM SIGMOD Conf. on Data Management*, 1995.

[14] M. A. Hernandez and S. J. Stolfo. Real-world data is dirty: Data Cleansing and the Merge/Purge problem. *Journal of Data Mining and Knowledge Discovery*, 2(1):9–37, 1998.

[15] A. Hinneburg and D. A. Keim. Tutorial on cluster Discovery Methods for Large Bases. In *Proc. of ACM SIGMOD Conf. on Data Management*, 1999.

[16] J. Hylton. Identifying and merging related bibliographic records. Master's thesis, Massachusetts Institute of Technology, June 1996.

[17] E. T. International. Home Page of ETI Data Cleanse. http://www.evtech.com.

[18] ISO. Database Language SQL - Part 4: Persistent Stored Modules (SQL/PSM). http://www.npsa.com/edd/.

[19] J. L. Kelley. *General Topology.* D. Van Nostrand Company, Inc., Princeton, New Jersey, 1955.

[20] K. Kukich. Techniques for automatically correcting words in text. *ACM Computing Surveys*, 24(4), 1992.

[21] A. Y. Levy and I. S. Mumick. Reasoning with Aggregation Constraints. In *Proc. of the Int. Conf. on Ext. Database Technology*, 1996.

[22] T. Milo and S. Zohar. Using schema matching to simplify heterogeneous data translation. In *Proc. of the Int. Conf. on Very Large Databases*, New York, 1998.

[23] A. E. Monge and C. P. Elkan. An efficient domain-independent algorithm for detecting approximately duplicate database records. In *Workshop on Research Issues on Data Mining and Knowledge Discovery in conj. with ACM SIGMOD*, 1997.

[24] Oracle. Oracle8 Server Utilities Manual, 1997.

[25] Y. Papakonstantinou, H. Garcia-Molina, and J. Ullman. MedMaker: A Mediation System Based on Declarative Specifications. In *ICDE*, 1996.

[26] I. Qualitative Marketing Software. Home page of Centrus Merge/Purge tool. http://http://www.qmsoft.com/.

[27] A. Rosenthal and E. Sciore. Metadata Propagation in Large, Multi-tier Database Systems. submitted to publication, 1999.

[28] A. Rosenthal and L. J. Seligman. Data integration in the large: the challenge of reuse. In *Proc. of the Int. Conf. on Very Large Databases*, Santiago, Chile, 1994.

[29] A. Roychoudhury, I. Ramakrishnan, and T. Swift. Rule-based data standardizer for enterprise data bases. In *Int. Conf. on Practical Applications of Prolog*, 1997.

[30] E. Sciore, M. Siegel, and A. Rosenthal. Using semantic values to facilitate interoperability among heterogeneous information systems. *ACM Trans. on Database Systems*, 19(2):254–290, 1994.

[31] L. Seligman and A. Rosenthal. A metadata resource to promote data integration. In *Proc. of IEEE Metadata Conference*, Silver Spring, MD, 1996.

[32] D. Shasha and T.-L. Wang. New Techniques for Best-Match Retrieval. In *ACM Transactions on Information Systems*, volume 8. Apr. 1990.

[33] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Theory*, 147:195–197, 1981.

[34] E. Ukkonen. Finding approximate pattern in strings. *J. Algorithms*, 6(1):132–137, 1985.

[35] Vality. Home page of Integrity tool. http://www.vality.com/html/prod-int.html.

[36] WizSoft. Home page of WizRule tool. http://www.wizsoft.com.

[37] G. Zhou, R. Hull, and R. King. Generating data integration mediators that use materialization. *Journal of Intelligent Information Systems*, 6(2/3):199–221, 1996.

A Cleaning and integration program for the Telecom Example

Let us consider the motivating example of section 1. A possible set of tuples is:
GSM-CLIENT

(1, "Smith J.", "Domaine de Voluceau BP 45", "096-11287", "Manager")
(2, "itch, J.", "17, rue Vasco da Gama", "096-76890", "Informatics")
(3, "itch, J.", "Domaine de Voluceau BP 45", "096-1128", "Manager")
(4, "Rabinovitch", "87, rue Gama"", "096-76800", "Bank Employee")
(5, "Smith J.", "D. Voluceau BP 45", "096-11287", "Manager")
(6, "Hans Oliver", "Champ de Mars", "096-15678", "writer")

HOME-CLIENT

(1245, "smith", "Domaine de Voluceau Rocquencourt", "01 39 63 56 32", "Director")
(1306, "J. Mitch", "r Vasco Gama 17 BP 20", "01 54 67 87 98", "Informaticien")
(1409, "ans oliverti", "Champ de Mars", "01 45 67 45 34', null)

INTERNET-CLIENT

(54, "John Smith", "Domaine de Voluceau, BP 45 Rocquencourt", "Smith")
(55, "J. Smitch', "rue Vasco Gama Paris", "Mitch")
(56, "smith", "Rocquencourt France", null)
(57, "Hanna Oliver", "23, Champ de Mars Paris", null, "Hanna Oliver")

The first step for cleaning and integrating these three tables is composed by the data and schema mapping operation represented below:

```
CREATE MAPPING MP1
SELECT gsmId, lowerName name, street, number, pob, city, gsm-phone, job
FROM GSM-CLIENT
LET lowerName = lowerCase(name)
    [street, number, pob, city] = extractAddressComponents(address)
```

The transformed MP1 tuples are:

(1, "smith, j.", "Domaine de Voluceau", null, "BP 45", null, "096-11287", "Manager")
(2, "mitch, j.", "rue Vasco da Gama", "17", null, null, "096-76890", "Informatics")
(3, "itch, j.", "Domaine de Voluceau", null, " BP 45", null, "096-1128", "Manager")
(4, "rabinovitch", "rue da La Fontaine Bordeaux", "87", null, null, "096-76800", "Bank Employee")
(5, "smith j.", "D. Voluceau", null, "BP 45", null, "096-11287", "Manager")
(6, "hans oliver", "Champ de Mars", null, null, null, "096-15678", "writer")

Analogous data and schema operations for HOME-CLIENT and INTERNET-CLIENT are:

```
CREATE MAPPING MP2
SELECT homeId, lowerName name, street, number, pob, city, home-phone, job
FROM HOME-CLIENT
LET lowerName = lowerCase(name)
    [street, number, pob, city] = extractAddressComponents(address)

CREATE MAPPING MP3
SELECT internetId, lowerName name, street, number, pob, city, username
FROM INTERNET-CLIENT
LET lowerName = lowerCase(name)
    [street, number, pob, city] = extractAddressComponents(address)
```

The next step consists in removing the duplicates from the GSM-CLIENT table. This is expressed by the sequence of operations matching, clustering, decision and construction as follows:

```
CREATE MATCH M1
FROM MP1 g1, MP1 g2
LET sim1 = simf(g1.name, g2.name)
    sim2 = simf(g1.street, g2.street)
    SIMILARITY = min(sim1, sim2)
WHERE G1.gsmId != G2.gsmId
THRESHOLD SIMILARITY > 0.6.
```

that outputs the following tuples:

```
(1, 3, 0.95)
(1, 5, 0.9)
(3, 5, 0.85)
(2, 4, 0.7)
```

Clustering by transitive closure after finding duplicates is specified by the operation:

```
CREATE CLUSTER C1
FROM M1
BY transitiveClosure
LET GSIMILARITY = min(SIMILARITY)
THRESHOLD GSIMILARITY >= 0.7
```

which returns the following sets of tuples with schema $(clustId, Key, clusterSim)$:

```
("cluster1", 1, 0.85)
("cluster1", 3, 0.85)
("cluster1", 5, 0.85)
("cluster2", 2, 0.7)
("cluster2", 4, 0.7)
```

The decision applied after partitioning operation C1 is represented by:

```
CREATE DECISION D1
FROM C1
BY clusterCohesion
THRESHOLD GSIMILARITY > 0.8
```

that decides automatically that 1, 3 and 5 are duplicates since the cluster *cluster*1 similarity is superior to 0.8. Cluster *cluster*2 is analyzed by hand since its similarity is below the decision threshold. If the user validates it as a valid cohesive cluster as well then 2 and 4 are also duplicates

An entirely manual construction operation (MP4) specifies that the user must be called to build the CLEAN-GSM-CLIENT tuples.

```
CREATE MAPPING MP4
SELECT gsmId, name, street, number, pob, city, gsm-nb, job INTO CLEAN-GSM-CLIENT
FROM D1
BY userCall
```

The following set of tuples is decided by the user as being free of duplicates (keys are generated automatically according to the keys of the base table GSM-CLIENT):

```
(7, "J. Smith", "Domaine de Voluceau", null, "BP 45", null, "096-11287",
"Manager-Director")
(8, "J. Rabinovitch", "rue Vasco da Gama", "17", null, null, "096-76890",
"Bank Informatics")
```

An SQL view could be defined to gather the no-matching tuples of GSM-CLIENT (tuple 6) with the CLEAN-GSM-CLIENT tuples. This is defined as:

```
CREATE VIEW V1 AS
        SELECT *
        FROM GSM-CLIENT g
        WHERE g.gsmKey NOT IN
                (SELECT key1 FROM M1)
        UNION
        SELECT *
        FROM CLEAN-GSM-CLIENT
```

The resulting tuples are then:

```
(7, "J. Smith", "Domaine de Voluceau", null, "BP 45", null, "096-11287",
"Manager-Director")
(8, "J. Rabinovitch", "rue Vasco da Gama", "17", null, null, "096-76890",
"Bank Informatics")
(6, "Hans Oliver", "Champ de Mars", "096-15678", "writer")
```

HOME-CLIENT does not need to undergo a process of duplicate elimination. Let us now consider the following matching operation between the result of V1 and the normalized HOME-CLIENT table:

```
CREATE MATCH M2
FROM V1 g, MP2 h
LET sim1 = simf(g.name, h.name)
    sim2 = simf(g.street, h.street)
    SIMILARITY = IF (sim1 > 0.8 and sim2 > 0.9) THEN RETURN min(sim1, sim2)
                 ELSE IF (sim1 > 0.8) THEN RETURN sim2
                 ELSE IF (sim2 > 0.9) THEN RETURN sim1
                 ELSE RETURN 0;
WHERE h.phone-nb LIKE '01%'
THRESHOLD SIMILARITY >= 0.6;
```

The resulting table has the schema $(key_1, key_2, similarity)$ and the resulting matching tuples are:

```
(7, 1245, 0.95)
(8, 1306, 0.8)
(6, 1409, 0.7)
```

A duplicate elimination process applied to INTERNET-CLIENT (similar to the one applied on GSM-CLIENT) transforms tuples 54 and 56 in a new tuple as follows:

```
(58, "John Smith", "Domaine de Voluceau BP 45 Rocquencourt France", "Smith")
```

The resulting table, called CLEAN-INTERNET-CLIENT is then transformed through a view V2 similar to V1 and composed of the initial tuples 55, 57 and the new 58. The matching operation between HOME-CLIENT and V2 is specified by:

```
CREATE MATCH M3
FROM MP2 h, V2 i
LET sim1 = simf(h.name, i.name)
    sim2 = simf(h.street, i.street)
    SIMILARITY = IF (sim1 > 0.8 and sim2 > 0.9) THEN RETURN min(sim1, sim2)
                 ELSE IF (sim1 > 0.8) THEN RETURN sim2
                 ELSE IF (sim2 > 0.9) THEN RETURN sim1
                 ELSE RETURN 0;
WHERE h.phone-nb LIKE '01%'
THRESHOLD SIMILARITY >= 0.6;
```

and returns the tuples:

```
(1245, 58, 0.85)
(1306, 55, 0.82)
(1409, 57, 0.65)
```

The results of M2 and M3 can be gathered through an SQL view V3 that executes the join between the two resulting tables. V3 outputs the triples (g, h, i) that semantically represent the approximate join between the three tables.

```
CREATE VIEW V3 AS
        SELECT M2.key1 key1, M2.key2 key2, M3.key2 key3, min(M2.similarity, M3.similarity) similarity
        FROM M2, M3
        WHERE M2.key2 = M3.key1
```

The resulting tuples are:

```
(7, 1245, 58, 0.85)
(8, 1306, 55, 0.8)
(6, 1409, 57, 0.65)
```

A decision executed after the view V3 is represented by:

```
CREATE DECISION D2
FROM V3
BY bestMatchFirst
THRESHOLD SIMILARITY > 0.7
```

Tuples $(7, 1245, 58)$ and $(8, 1306, 55, 0.8)$ are decided automatically as best matches. Tuple $(6, 1409, 57, 0.65)$ is decided by the user since its similarity is below the decision threshold.

The following construction mapping operation is applied after decision D2 and intends to build tuples of the integrated view table CLIENT. D2 tuples whose construction similarity is below the threshold of 0.9 are manually constructed. The others are automatically constructed. The construction threshold states that if the three names or street names are not sufficiently similar then the user is called to construct by hand the CLIENT tuples.

```
CREATE MAPPING MP5
SELECT key, name, address, gsmNb, homeNb, username, job INTO CLIENT
FROM D2
LETkey = keyCreation(D2.key1, D2.key2, D2.key3)
    sim1 = simf(D2.key1.name, D2.key2.name, D2.key3.name)
    name = IF (sim1 > 0.9) THEN
               return D2.key2.name
    sim2 = simf(D2.key1.street, D2.key2.street, D2.key3.street)
    street = IF (sim1 > 0.9) THEN
               return first(D2.key1.street, D2.key2.street, D2.key3.street)
    number = first(D2.key1.number, D2.key2.number, D2.key3.number)
    pob = first(D2.key1.pob, D2.key2.pob, D2.key3.pob)
    city = first(D2.key1.city, D2.key2.city, D2.key3.city)
    address = concat (number, street, pob, city)
    gsmNb = IF (key1 != null) THEN return(D2.key1.gsm-nb)
                              ELSE return(null)
    homeNb = IF (key2 != null) THEN return(D2.key2.phone-nb)
                              ELSE return(null)
```

```
    username = IF (key3 != null) THEN return(D2.key3.username)
                                 ELSE return(null)
    job = concat(D2.key1.job, D2.key2.job)
    SIMILARITY = min(sim1, sim2)
THRESHOLD SIMILARITY > 0.9
```

The resulting CLIENT tuples are:

```
(C1, "J. Smith", "Domaine de Voluceau BP 45 Rocquencourt", "096-11287",
"01 39 63 56 32", "Smith", "Manager-Director")
(C2, "J. Mitch", "17, rue Vasco da Gama BP 20 Paris", "096-76890",
"01 54 67 87 98", "Mitch", "Informatics")
(C3, "Hanna and Hans Oliver", "23, Champ de Mars Paris", "096-15678",
"01 45 67 89 76", "Hanna Oliver", "writer")
```

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399