Contact Mapping Tool

Design and Impementation

Stephen Hui

Web Developer

Table of Contents

Introduction

Because specific tools were not available at the time, and there was a long backlog of leads that needed to be mapped to CRM contacts and accounts, the Marketing department formed a team to create such a tool that would electronically automate the manual processes involved.

That tool became the Contact Mapping Tool (CMT). Many hours of designing and redesigning, testing and retesting by many people have gone into the CMT as it is now. Many thanks go out to everyone who has contributed to the tremendous success of the CMT.

Although the CMT was originally designed to support the Americas region only, it should be theoretically possible to extend it to cover all regions. Much more time and many more resources must be devoted to make it a truly global, scalable platform.

The purpose of this document is to describe, in detail, the inner workings of the CMT, not only to record how it works, but to be a baseline from which the CMT can be reimplemented in other environments.

Programming environment

The current incarnation of the CMT (at the time of this writing) is built on a custom platform, powered by PHP and MySQL on a Linux server. The flexibility afforded by the PHP/MySQL/Linux environment allows much more freedom for performance enhancements. Although this was chosen as the base platform for the CMT, it can be also implemented on other platforms (e.g. ASP.NET, C#, Java). This document will describe how to implement the CMT in any environment.

Prerequisites

Understanding the CMT requires at least a working knowledge of object-oriented programming principles and SQL database design and programming. Since the CMT is very data-driven, the code is very database-heavy. The CMT code itself essentially serves as a window into the database tables.

Since the CMT is used in a collaborative environment, it must also have the capability to allow multiple users to map files, and it must be secure. A basic content management system, or other user-authentication platform should be used to restrict use of the tool to specific users.

Chapter 1: Searching and Matching

Searching and matching data to existing data records is a very difficult task. Consider the following scenario:

a) You have a database which contains millions of records of contact information, including first name, last name, email address, company, and so on.

b) You are given a list of names, email addresses, and companies to match to existing data records.

c) You are charged with looking at each given record (could be one, could be 1,000), and seeing if that record exists in the database.

d) The data in the database is not necessarily normalized, and neither are the given data records.

e) The names in the list to match may be nicknames (Mike, Michael, Mick, etc).

f) The names in the database may be nicknames.

g) The companies in the list and in the database may not exactly match.

The complexity of the searching and matching data is directly proportional to the amount of human involvement in the processing of said data. In other words, the more people create the data, the more variations there can be on each data field, even though they represent the same information, and so the more the process relies on human intervention to make the matches.

The Contact Mapping Tool was created to fill such a gap, between human-powered “eyeballing” and rules-based computerized data processing. But even in theory, such a computerized system can only do so much. The concession then is to match as much data as possible, as accurately as possible, then present enough data to a human user to finish the rest, reducing the amount of work required by the human analyst, and so increasing the overall throughput of each human analyst.

Algorithm overview

There are three stages of matching, each of which incrementally narrows the search field. The end goal is to find exactly one or exactly zero records in the database. If the CMT can find exactly one or exactly zero matches, then the degree of certainty of the match (or no match) increases significantly.

Stage 1: Account lookup

The first stage, Account Lookup, attempts to find accounts (companies) that match the given record, ignoring any contact data (name, email address, etc.). This is the broadest search, and has no guarantee of returning any results.

If accounts are found, Stage 2 (Contact Lookup) is narrowed to only the matching accounts.

If accounts are not found, Stage 2 is a broader search, encompassing all accounts.

Stage 2: Contact lookup

The second stage, Contact Lookup, attempts to find contacts that match the given record, based on email address, name, and country, and also includes account IDs, if any were found in Stage 1.

The Contact Lookup stage is done in 5 passes:

1. Exact Email – The CMT attempts to find any matching contacts with the exact email address given.
2. Exact Full Name – Next, the CMT looks at the exact full name (first and last) of the given record, and attempts to find matching database records.
3. Reversed Full Name – Next, the CMT assumes that the first and last names are reversed, and performs an Exact Full Name search with the first and last names reversed.
4. Exact Last Name – Next, the CMT attempts to match only by the last name of the given record.
5. Account Lookup – As a last resort, the CMT attempts to find any possible accounts that might match. If Stage 1 found accounts, then those accounts are returned. If Stage 1 could not find any accounts, then any possible matching accounts are returned.

Each pass broadens the search. As soon as at least one record is found in any one of the passes, the CMT stops searching and proceeds to score the results (Stage 3).

Stage 3: Scoring

If any contacts are found, a “Match Confidence” score is assigned to each possible match. The Match Confidence is a floating-point number between 0 and 100, and gives a relative probability that the database record matches the given record. A score of zero indicates that the record is absolutely not a match, and a score of 100 indicates that the CMT is almost absolutely certain of the match.

> Probabilistically, the CMT can never really be exactly 100% certain that a record matches. There may be very slight variations in the data that would preclude such an event. However, the CMT matching rules define that the Match Confidence score can be 100, but only if all of the following conditions are met:
> - The email address is an exact match.
> - The first and last names match within defined tolerances (see “Text matching”).
> - The company name matches exactly.

In order to determine this “closeness” factor, the CMT compares the first name, last name, company, and email address, as well as assigning a weighting factor based on how the contact was found. Each of these scores is then weighted and averaged, producing the final Match Confidence score.

The scoring is broken down like this:

1. Compute the first name score (see “Text matching”).

2. Compute the last name score.
3. Compute the company name score.
4. Compute the “type confidence” score. This score corresponds directly to the first four passes of the Contact Lookup stage. The type confidence score starts at 100, and each pass decreases the score:
    a. Exact Email: -1 point
    b. Exact Full Name: -7 points
    c. Reversed Full Name: -10 points
    d. Exact Last Name: -15 points

    If there is exactly one matching record, the type confidence increases by 0.5 points. If the company name matches (see “Text matching”), the type confidence increases by 5 points.
5. The final Match Confidence score is computed by finding the weighted average of the first name score, last name score, and type confidence score. The first name score has a weight of 5, the last name 2, and the type confidence 1.

Once the final Match Confidence is computed, a confidence threshold is applied, effectively filtering out low-confidence matches. At the time of this writing, the confidence threshold is 60 (i.e. any record with Match Confidence score below 60 is unconditionally ignored).

Text matching

Text matching, especially with names, is a very tricky procedure. Ideally, the CMT would emulate how a human would do the matching. Enter the Double Metaphone and Levenshtein distance algorithms.

The double metaphone algorithm

The double metaphone algorithm attempts to breakdown a string into phonetic tokens. Double metaphone produces two phonetic keys, a primary and secondary key. The purpose of the two keys is to account for common ancestry of certain words. For more information, see this Wikipedia article: http://en.wikipedia.org/wiki/Double_Metaphone#Double_Metaphone

> There is one drawback to using double metaphone for matching: it only works with certain Latin alphabet-based character sets. Other languages, such as Chinese and Japanese, which are more difficult to breakdown phonetically, would have to revert to exact string matching.
>
> Ideally, the CMT could use a method for a similar string transformation for non-Latin alphabet languages. This would greatly increase the CMT accuracy for those languages.

The Levenshtein distance

The Levenshtein distance gives a number that represents how many edits (insert, delete, substitution) of single characters are required to transform one string into another. The fewer edits required, the closer the two strings are. For more information, see this Wikipedia article: http://en.wikipedia.org/wiki/Levenshtein_distance

> The same caveat applies to Levenshtein distance as for double metaphone: certain language character sets (e.g. Chinese and Japanese) cannot be used with the Levenshtein distance. These strings must revert to exact string matching, unless another method can be devised.

Putting the pieces together

By computing the double metaphone keys for two strings, then computing the Levenshtein distance between the keys, the CMT now has a fairly good indicator of how phonetically close two strings are. This “closeness” indicator is then used to score the first name, last name, and company name matches.

Nicknames

In addition to text matching, the CMT also takes into account nicknames for common names in Latin alphabet-based languages. When performing the Exact Full Name and Reverse Full Name passes in Stage 2, the first name of the given record is looked up in the nickname database, and any results are also included in the search during these passes.

The net result is that the CMT not only can find “Michael” which matches “Mike” phonetically, but it will also match “Harold” with “Hal”, or “Judson” with “Sonny.” A side effect of this form of matching primarily gives preference to exact matches, but can secondarily give preference to nicknames because of the phonetic comparison.

Comparing companies

Comparing names of companies proves to be equally as challenging as comparing names of people. There can be numerous variations on company names, especially if there are abbreviations or other modifications to names.

The CMT solves this problem by taking the two strings to compare, normalizing both, then breaking up the two strings into tokens and comparing the tokens.

Normalizing the company name

The first step is to remove any characters that are slashes (‘/’), backslashes (‘\’), single- or double-quotes, or hashes (‘#’).

Next, the normalization function removes common phrases or abbreviations that would otherwise be affected by the next step. These phrases and abbreviations include:

- `'s.a.', 'w.l.l.', 'ind school district', 'school district', 'isd', 'i.s.d.', 'univ', 'university', 'state univ', 'state university', 'business solutions', 'credit union', 'federal credit union'`

The third step is to tokenize the string (separate the string by spaces, i.e. individual words), and store the tokens as array elements.

The final step is to remove any tokens that match the following:

- `'the', 'and', '&', 'of', 'at', 'inc', 'incorporated', 'corp', 'corporation', 'co', 'company', 'ltd', 'limited', 'gmbh', 'llc', 'lp', 'llp', 'pt', 'pty', 'sa', 'ag', 'bv', 'wll', 'manufacturing', 'industrial', 'cust', 'entertainment', 'group', 'source', 'financial', 'services', 'systems', 'north', 'south', 'east', 'west', 'central', 'international',`

The string to be normalized must pass through two separate filters because the longer, multi-word phrases in the first pass must be considered in context with the rest of the string. By filtering phrases first, then individual words, the CMT can more accurately judge what token is important when considering a company name.

Once the two company names to be compared are normalized, the CMT computes how many tokens are common to both strings, and looks at the ratio of common tokens to the number of tokens in the reference string (the match rate). In this case, the reference string is the name of the company provided by the input file, and the compare string is what was found in the database.

The final result is defined by the following rules:

1. If the reference string has more than three tokens, the two strings must have a match rate of 50%. In other words, at least half of the tokens by the input file must exist in the tokens found in the database.
2. If the reference string has exactly three tokens, the match rate must exceed 67%.
3. If the reference string has exactly one or two tokens, the match rate must be 100%.

If any one of these rules holds, then the CMT considers the company names to match. If none of these rules holds, then there is no match.

The final answer

Once everything is searched, scored, compiled and filtered, the CMT outputs its search results in CSV format. The Match Type indicates how the record was found, and the Match Confidence score is given, along with any contact or account data found.

A human analyst is then able to review the results of the mapping, and make corrections as necessary.

Chapter 2: Design

This chapter aims to provide a very high-level overview of what the CMT is, what it does, and how it works. The inner workings will be all laid out in excruciating detail in subsequent chapters.

PHP code architecture

The CMT is designed in an object-oriented architecture, to make implementing and extending simple and easy. PHP 4.x provides basic object programming, while PHP 5.x is much more advanced. The CMT codebase was originally developed on a PHP 4.x server, so the PHP 4.x objects are preserved.

Before the CMT, a custom platform was developed, implementing a web-based user interface complete with user logins, groups, permissions, templates and a database abstraction layer. This formed the basis for the CMT.

Also, being a web-based environment, a good user experience is required. This includes a nice, easy-to-use interface, as well as performance enhancements and visual feedback. The CMT employs three layers in the user interface to accomplish this:

1. HTML display
2. Interactive interface with AJAX
3. Server-side background processes

The HTML display provides a basic framework for displaying data views. This includes tables and regions where data is dynamically populated through AJAX calls to the server.

The AJAX layer provides interactivity with the end-user. This includes retrieving view data and populating the placeholders in the HTML code with Javascript, and giving visual feedback when something is happening in the background.

Many files that CMT imports or maps contain a large number of records, which cannot be processed in a practical way with a simple HTML form POST, and waiting for the results. Therefore, each import and map is created as a separate background process on the server. The import and map processes communicate with the display layers of the CMT through database tables and log files that track the progress. In this way, the user is free to log off and return later if a file will take a long time to process.

Data architecture

All CMT data is stored in a SQL database. The data must be optimized, and the tables structured in such a way that the greatest performance can be extracted from the database server software. More discussion on this topic in

Chapter 4: Performance. There are three main sets of database tables:

Contact records

The contact records consist of data pulled from a CRM system, and includes basic information such as name (first and last), company, email address, CRM contact ID, and account ID from an internal accounts database, among others. This data is used to compare against the input file data to find the most likely matches.

Since there are so many contact records (multi-millions), some special data optimizations must be done to give the end-user a good user experience. See “

Chapter 4: Performance” for more details.

The contacts records are split into numerous smaller tables, each with no more than 5,000 records.

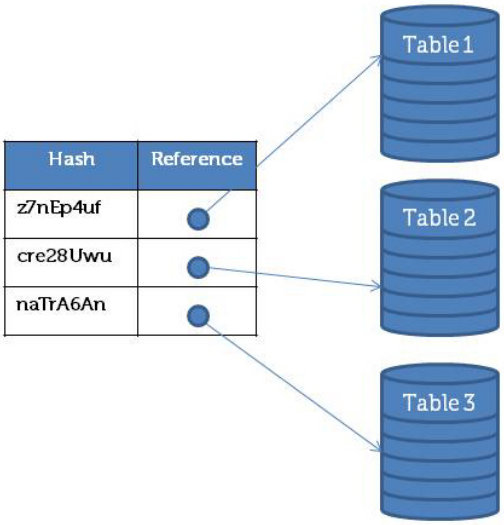

Figure 1

Account records

The account records reflect the relationships between customer accounts. These relationships can be parent-child, or aliases. For example, if Company B is a subsidiary of Company A, their relationship is encoded in the account records. Similarly if Company D is a doing-business-as name for Company C, then that relationship is also encoded.

The purpose of the account records is to be able to more accurately match either contacts to contacts by their common account IDs, or contacts to accounts when the contact does not exist.

Analytics data

The CMT records analytics data for each file mapped, including time spent processing, match rate, and so forth. Storing this data allows an auditor to go back and review the statistics for certain files, or a developer to diagnose and fix problems with the CMT. For more details, see ‘

Chapter 5: Analytics.”

Chapter 3: Application Interface

The main focus of the design of the Contact Mapping Tool (CMT) was to create something that would be easy to use, easy to develop, and easy to extend.

Foundations

At the core of the CMT is a single base object from which other objects are built, `_MapBaseClass`. This object encapsulates all of the common functionality required to import and purge a database, map input against the imported data, and produce output. `_MapBaseClass` also carries some important internal metadata which aids in debugging and analytics. The developer creates implementation objects which extend `_MapBaseClass`.

The _MapBaseClass interface

`_MapBaseClass` provides a number of API methods. The following methods provide basic input/output services to the implementation object:

- `_MapBaseClass`: the constructor method. Creates and initializes the database connection, and initializes internal data members.

- `setUserId()`: Stores the user ID of the current user into the `_MapBaseClass` object. Useful for tracking, debugging, and analytics.

- `setInputFile($file)`: Stores the full path to the file being processed, whether for an import or for mapping.

- `getInputFile()`: Returns the file path set by `setInputFile()`.

- `setRealInputFile($file)`: In order avoid collisions (and secondarily secure) the files being imported and mapped, a unique hash is assigned to each uploaded file, and that hash is used to reference the file internally (see `setInputFile()` and `getInputFile()`). `setRealInputFile()` stores the actual real-world name of the input file for later retrieval.

- `getRealInputFile()`: Returns the real-world name of the uploaded file.
- `setOutputFile($file)`: Stores the desired output file name.
- `getOutputFile()`: Returns the desired output file name.
- `setRequiredFields($args)`: Each implementation object can specify fields that are required in the input field before processing can begin. `setRequiredFields()` accepts a variable-length argument list specifying the required fields.
-
- `getRequiredFields()`: Returns the list of required fields (see `setRequiredFields()`).
- `clearRequiredFields()`: Clears the list of required fields (see `setRequiredFields()`).
- `hasRequiredFields($headers)`: Indicates whether there are any required fields. `hasRequiredFields()` accepts the fields defined in the input file, and compares them against the list of required fields.
- `setMappedFields($args)`: Each implementation object can specify data fields that are appended to each line of the output file. `setMappedFields()` stores these append fields. `setMappedFields()` accepts a variable-length argument list specifying the append fields.
- `getMappedFields()`: Returns the list of append fields (see `setMappedFields()`).
- `clearMappedFields()`: Clears the list of append fields (see `setMappedFields()`).
- `setData()`: The input file is read line-by-line, and the parsed input data is fed into the implementation object through the `setData()` method. `setData()` accepts key-value pairs.
- `getData($key)`: The input data row is stored as key-value pairs (see `setData()`), so data must be retrieved by the key. The keys here correspond to the column headings of the input CSV file.

- `setOutput($col, $val)`: The output file is first cached internally as a multi-dimensional array, then rendered as CSV output. `setOutput()` sets the value of a column in the current row of output.

- `clearOutput()`: Clears the output cache.

- `breakOutput()`: Starts a new row of output.

- `writeOutputHeaders()`: Copies the headers from the input file to the output file, then appends the mapped column headers (see `setMappedFields()`).

- `flushOutput()`: Flushes the output cache to the output file, rendering the data in CSV format.

- `setColumnAlias($key, $val)`: When mapping a file, to provide the greatest flexibility, the user must specify which columns in the input file correspond to which data fields (e.g. first name, last name, company, etc.). In this way, the input CSV is not required to be in a certain format, and does not require specific column headers. setColumnAlias() stores the mapping between the input file column and the internal field name.

- `getColumnAlias($key)`: Looks up the column header from the input file corresponding to the internal name specified by `$key`, then returns the data for that column in the current row.

- `setInputData($key, $val)`: Stores the input cell data corresponding to a specific internal column alias (see `setColumnAlias()`).

The following methods provide services to perform data import, data mapping, and other data abstractions:

- `map($columnMap)`: Given an input file (see `setInputFile()`), an output file (see `setOutputFile()`), and a comma-separated list of columns and their internal aliases (see `setColumnAlias()`), read the input file line-by-line, passing each data row into the map implementation method, `_map()`. The implementation object **must** implement `_map()` if it is to map data. The output cache as generated by `_map()` is then rendered as CSV to the output file (see `flushOutput()`).

- `import()`: Given an input file (set `setInputFile()`), read the file line-by-line, and perform the import implementation method, `_import()`. The implementation object **must** implement `_import()` if

it is to import data.

- `startTimer()`: Start the work timer. This method is deprecated in favor of services provided by the `MapMetrics` object.

- `markTime()`: Stop the work timer. This method is deprecated in favor of services provided by the `MapMetrics` object.

- `clearTimer()`: Reset the work timer. This method is deprecated in favor of services provided by the `MapMetrics` object.

- `getNumRecords()`: Calls the implementation method `_getNumRecords()`. Since each data source may have a different way of counting the number of records in the database, this API provides a customization hook. The implementation object must implement `_getNumRecords()`.

- `reindex()`: Calls the implementation method `_reindex()`. Since each data source may have a different way of rebuilding the database indexes, this API provides a customization hook. The implementation object **must** implement `_reindex()`.

- `purge()`: Calls the implementation method `_purge()`. Since each data source may have a different way of purging the database in order to preserve data integrity, this API provides a customization hook. The implementation object **must** implement `_purge()`.

- `getFileLength()`: Returns the total length of the input file, not counting the first (header) row.

- `logStats()` <<DEPRECATED>>: Writes the current import or map statistics to a log file. This method is deprecated in favor of services provided by the `MapMetrics` object.

The MapContacts object

The `MapContacts` object was the first object to be created that extends `_MapBaseClass`. MapContacts encapsulates everything necessary to mapping contact records against the imported database. In fact, the CMT begins mapping a file like so:

```
<?php
if (file_exists($in)) {
      $map = new MapContacts();
```

```
        $map->setUserId($id);

        $map->setInputFile($in);
        $map->setRealInputFile($realName);

        $map->seOutputFile($out);

        $map->map();
}
?>
```

Chapter 4: Performance

In the early development stages, the CMT asymptotically slowed to a crawl as the number of records in the contact records table steadily increased. This symptom manifested both in checking for duplicate records on import, and in mapping contacts. Through careful analysis of the test data, the database was found to be the main bottleneck. At the time, the CMT was being developed on a much older system that just did not have the horsepower to handle the number of records. This led to a number of seemingly strange but innovative database table optimizations.

Too many contacts

The contact records table proved to be the single worst bottleneck of the entire system. Searching through 4-million-plus contacts on an antiquated system with limited resources was less than ideal. Even migrating to a database server with better optimization options did not help much.

However, breaking up the full data sets into smaller tables, and keeping a streamlined cache table based on how the contact searching and mapping is done resulted in a massive performance boost.

Less is more

Many database servers lose performance when the index is long and there are very large numbers of records. Instead, the CMT keeps a cache table with a few crucial columns, and a large number of smaller tables with the full data sets. The contact import method creates a hash that is more easily indexed and searched, with a reference to the full data set. So when it maps a file, it generates a hash of the input data, and searches for the hash in the cache table. If found, the CMT then has the exact location of the full data set that it can retrieve for further processing.

The result is a speed up of over 10-times for importing and mapping, with almost no impact to performance as the number of contact records increases.

Too many accounts

Originally, the CMT was designed to map contacts to accounts, favoring certain accounts. But because it was using the account IDs from an internal accounts database, the mapping was not as clean, and so the CMT would find more accounts in the first stages of mapping than it needed to.

Now, the CMT uses the DUNS IDs to map accounts. These account IDs also reflect parent-child entity relationships, which greatly increases the accuracy. Additionally, the DUNS database identifies the segment, account owner, and the account status, so much of this filtering can be done early on in the mapping process. Switching to the DUNS database has increased mapping performance by more than double.

Chapter 5: Analytics

The CMT provides access to mapping analytics, including actual time spent in the database, actual time spent in the PHP code, and map metrics for each input record. This functionality is provided by the `MapMetrics` object.

The `MapMetrics` object