matchIT SDK

Information Pack

www.helpIT.com

Contents

Introduction

Welcome to the matchIT® SDK Product information pack. The matchIT SDK has been developed as a starting point to allow developers to integrate the matchIT API with existing applications and run batched data cleansing. The document assumes that you have knowledge of programming, and makes use of example code written in c#, although the actual SDK is available in VB and C++.

The document has been split into the following three chapters.

- Product Overview – an overview of what the SDK does by default
- Product Tutorial – tutorial detailing how you would go about making code changes to modify the default SDK to work with your own database and matching keys.
- Product Technical Reference – technical reference describing each of the classes which can be found in the SDK code.

You should also refer to the matchIT API user manual when requirements warrant customisations to the default matchIT API settings.

Product Overview

The matchIT SDK consists of two demo applications with source code that enable software developers to quickly build batch applications using the matchIT API, either by using the supplied source with modifications specific to the application's data source, or by using the supplied source as a guide in writing a new application.

The sample applications are for single file matching (e.g. deduplication of a customer database) and matching across two files (e.g. to identify the overlap when feeding a new batch of records into a master database, or to suppress existing customers from a mailing).

Currently, the matchIT SDK includes sample Visual C++ and VB.NET applications, for matching name and address data within a database (eg. SQL Server, Access, or MySQL). The developer can select some or all of the objects in the SDK to easily deploy the matchIT API for deduplication within one file or matching records across two files. It is intended to supply further programs for the matchIT SDK in C#.NET.

The matching functionality in the matchIT API uses several proprietary "fuzzy matching" methodologies, including:

- Phonetic matching to match "sounds like" names such as Deighton and Dayton. Three levels of phonetic routine are available, the choice depending primarily on the nationality of the data.
- Lookup tables to match names such as Bill and William, The Guthrey Group and Guthrey Ltd.
- Acronym and initial matching to match inconsistencies like Bill and W. (e.g. Bill Deighton and Mr. W. Dayton), The Guthrey Group and TGG Inc.
- Non-phonetic fuzzy matching to match keying errors (such as transpositions like Wilson and Wislon) and reading errors (such as Morton and Horton).
- Element matching to match names with elements missing or reversed, such as Mr J R Gonzalez, Jose Gonzalez and Gonzalez Jose.

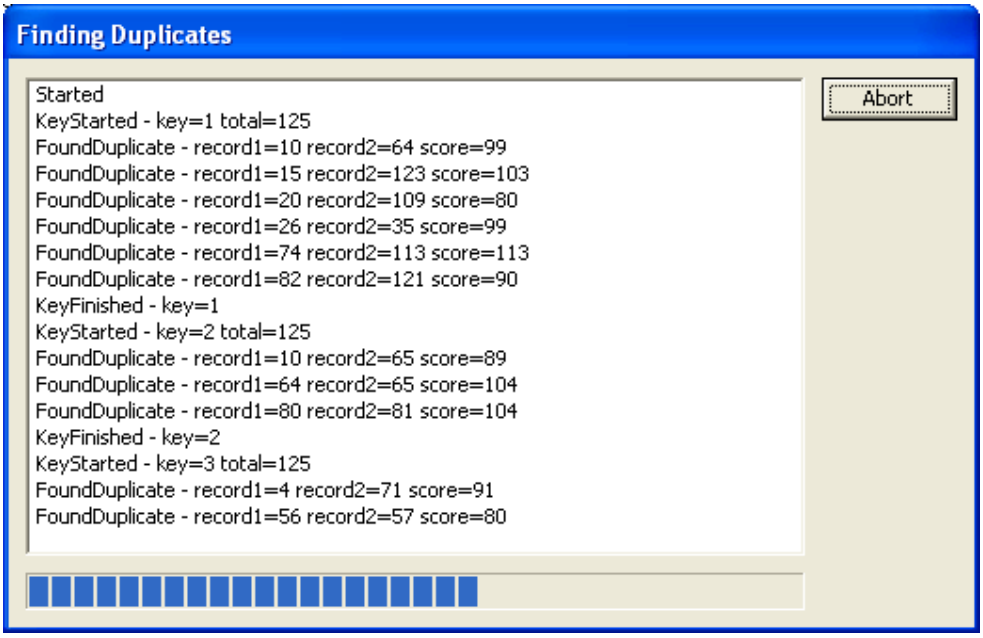
The matchIT API grades duplicates by "score" – a higher matching score indicates a higher certainty of match.

The sample application consists of discrete objects, which handle the functionality of single and two file matching. A detailed description of each object is contained in the matchIT SDK Technical Reference Chapter later on in this document, and examples of how the code can be modified to work with your own database can be found in the Product Tutorial (next chapter).

Single File Deduplication

Selecting the 'OneFileDedupe demo' option from the Windows Start menu runs the matchIT SDK on a single example file, which is provided with the SDK installation.

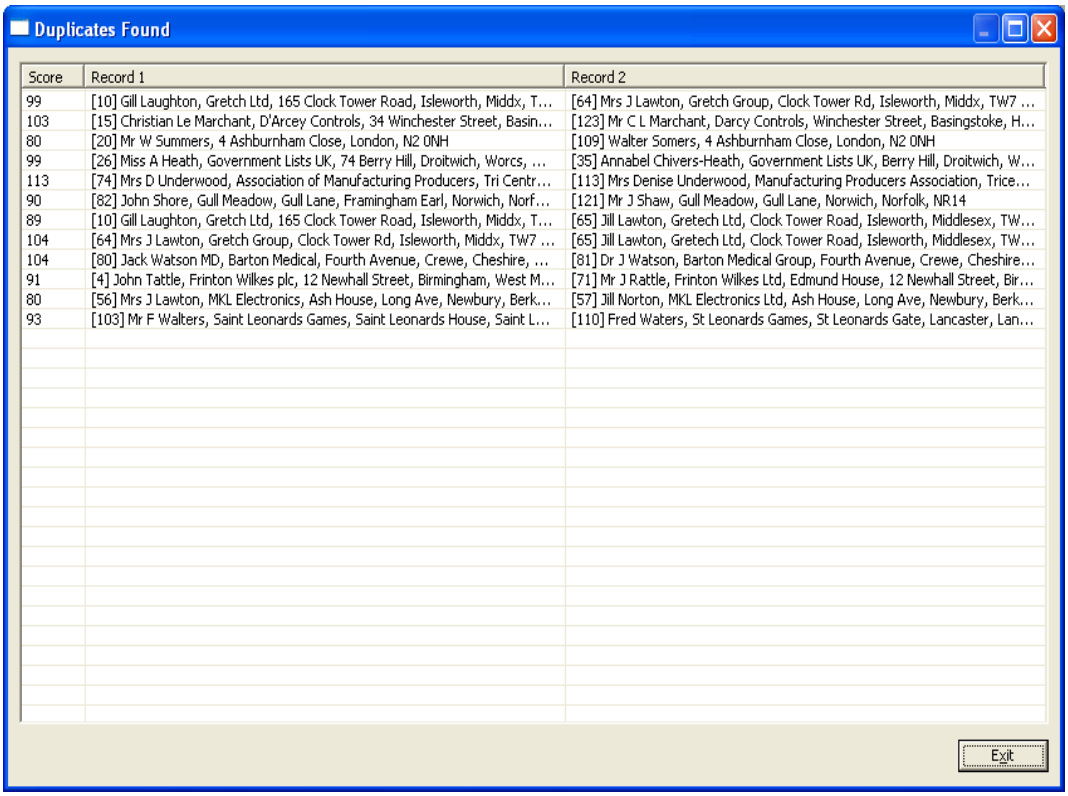
The deduplication process then begins showing a progress window as displayed below.



During the deduplication process the matchIT SDK will run three match keys to find potential duplicates within the file. The keys utilized by the demo program are designed to find all the duplicates, without relying on a single item of data (such as phonetic surname) always being consistent. In this way, it can find duplicates with keying errors, one record with a postal code and one without etc.

As each key finds a potential pair of duplicate records, the pair is then scored by the matchIT API. A higher matching score indicates a higher certainty of match.

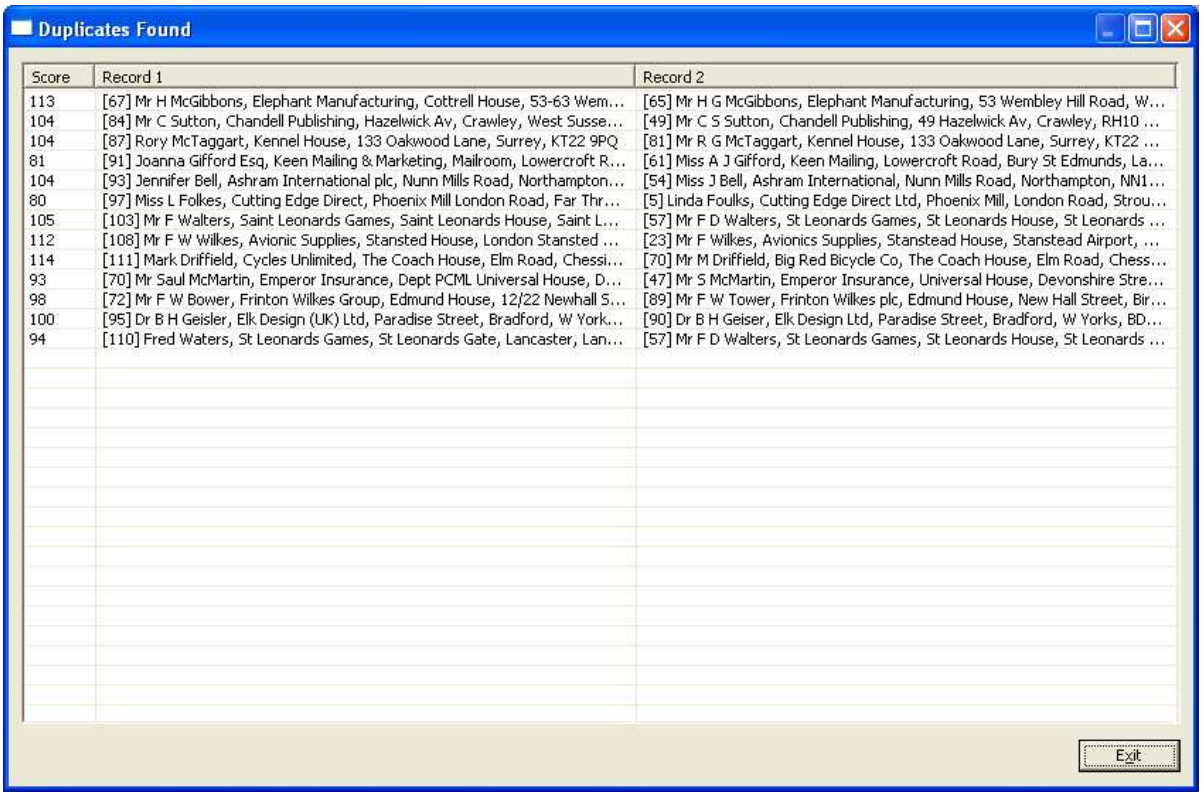
Once the duplicate search has completed, the following window will be displayed (see next page):



The window above displays the matching records found in a side by side view. The matching score is shown on the left side of the window. The display can be sorted by clicking any of the three column headers – the record columns will then be sorted by the ID of the records in that column. The ID (preceding each entry) is the record's unique reference number.

Two File Deduplication

The matchIT SDK also contains a separate demo for the two file batch process. This process will compare two separate databases and report back any overlapping records. The two file process is very similar to the single file batch process described above, with the only change coming in the 'Duplicates Found' window. The 'Record 1' column will list the records from the first table and the 'Record 2' column will list the second table entries.



Getting Started Tutorial

Introduction

The matchIT SDK, as you saw in the previous chapter, comes ready to run and display some sample results using 2 example Access (.mdb) databases. The first section, 'Quick start with example data', explains how to get up and running initially with this sample data, so you can see examples of the results that can be produced.

In terms of getting the matchIT SDK up and running with your own data, there are various classes that you will have to be amend in the source code. Let's start with looking at the bare minimum that you need to amend to get the SDK at least communicating with your data, then look at tweaking the associated methods thereafter. This is talked through step by step in the second section 'Getting set up with your own database schema'.

Note that this tutorial uses the C# project, but is entirely applicable to the C++ and VB.NET projects too as they share a common layout and feature set.

Quick start with example data

In its initial state, the matchIT SDK is set up to perform a single file de-duplication on one of two sample Access databases that are provided with the SDK. This can be run in 2 ways. The first is to open up the project in Visual Studio and run without debugging. The second is to run the toolkit_onefilededupe shortcut located in the demo folder of the application (which is also accessible via the windows Start > programs > matchIT SDK > Demo). In the latter case you will also see an option to run a 2 file de-duplication. In all cases, you should see a results dialog box appear showing a list of matching pairs and their associated match score (i.e. how well they match). Run either of these demos to view the results produced.

We will now look at linking the SDK in with your own database system to run similar tests on your own data and produce results in the style that you have just seen.

Getting set up with your own database schema

Overview of files

Initially, let's look at setting up the SDK to perform a single file de-duplication. The classes that you will need to amend are as listed below.

- Toolkit_cs.net.cs
- Database.cs
- Tasks.cs

Here is a brief overview of each.

Toolkit_cs.net.cs

This is the entry point for the application when run. It is in here that you can specify your data source(s), and exactly what you want to perform on the data.

Database.cs

This class contains all methods and properties needed for communicating with the database. Also in this class, the various columns of your data are mapped to various properties of the Record.cs class, which will have to be amended to suit your database structure.

Tasks.cs

his class basically contains all the available methods for manipulating the target data. The methods will need amending depending on the structure of your data in terms of what tables there are and how they are structured.

Amending the files

First of all, let's amend the Database.cs class. Firstly, at the top of the code, you will see some #define statements that specify the different database types. It is important here to make sure that the correct #define statement is uncommented here. For example, if you are using SQL Server, you would uncomment #define DB_SQLSERVER and comment out the other #define statements.

Once that is done, staying with the Database class for the moment, you need to map the table fields from your database to the properties of the matchIT record object in the region of the Database.cs class labelled 'Field mappings are defined here'. You will see that there is a whole list of field mappings, split into 3 groups by comment lines as InputFields, Matching Fields and KeyFields. For the purpose of getting started you just need to look at the input field mappings. To see how the mappings work, take the first one as an example -

```
new FieldMapping( "FullName",
                new Record.GetValueFn( Record.GetFullName ),
                        new Record.PutValueFn( Record.PutFullName ))
```

The second and third arguments are the functions that map the property of the record object to / from your database column, the name of which is the first argument. You will see underneath this mapping in Database.cs (i.e. the second mapping) the mapping of "Name" uses exactly the same functions in the second and third arguments, i.e. it maps the column "Name" to the same properties as it would the column "FullName". This is commented as a synonym, as a database could have a column containing the full name of a contact labled in either way – therefore both will get mapped in either case. If your column containing the full name of a contact is not labelled as either of these, for example say it was labelled "Full_Name", as it stands it will not get mapped to the record object. All you have to do in this case is add it to the code as another synonym. So, you would simply copy say the FullName mapping line of code, paste it directly underneath and change the first argument to "Full_Name".

Have a look through all of the field mappings in the input fields section and add any synonyms that you need for any of the other mappings to ensure that all the relevant columns in your database are mapped to the relevant properties.

Note - These columns may be spread across more than one table in your database, such as a contacts and an addresses table, however in all the processes of the SDK multiple tables can be joined together to form one big table. As a default, the SDK is set up to deal with data split into two tables, 'contacts' and 'addresses', which are joined together in various processes. However, if this is not the case and all your data is stored in one (or maybe more) tables, the joins in the various processes can be amended accordingly, which is what we will look at next.

Moving onto the Tasks.cs class. It is in this class that all the main methods are defined for manipulating the data. If we are using a SQL database, one of the first things that we will need to do is generate and populate a keys table in the database that will be used by the application to process the data and find duplicates. The keys table is generated in 2 steps – firstly the table is created by the method CreateKeysTable(), and then secondly the table is populated with the key data for each record in your database on a one to one basis.

Looking at the first step, CreateKeysTable(), you only need to make one change and that is to amend the name of the ID column in the create table sql statement. In the process of finding duplicates, the application will join the keys table to the table(s) of data that it is processing by the unique ID column of both the keys table and the table(s) of data – it is necessary that the names of these columns are the same for the join to work. So for example, if the ID column in your table(s) of data is called "Unique_ID", then the name of the ID column for the keys table in the function CreateKeysTable() will also have to be changed to "Unique_ID".

The second step, generating and inserting the data, can be done in two ways, either sequentially row by row (Import), or all at once (BulkImport). The latter only exists for SQL Server databases, so lets look at the former to cover everything (essentially the BulkImport is amended/implemented in the same way as described below, however its workings are different, and only supported for SQL Server at the moment). The normal import method is called Generate(), which is found in the class 'Import' within the Tasks.cs file. The amendment you need to make here is on the joining of various tables shown in the code below.

```
toolkit.Table contacts = new toolkit.Table( database, "contacts", "ID" );
toolkit.Table addresses = new toolkit.Table( database, "addresses", "ID", contacts );
toolkit.Table genkeys = new toolkit.Table( database, "genkeys", "ID", contacts );
// *** NB: If genkeys is empty then don't join it onto contacts ***
```

There are a couple of things to mention here. Firstly, each table in your database is represented as a table object in the application. If you look at how a table object is instantiated in the first line of code, the first argument is the database object that was created a couple of lines before, the second is the name of your table in the database and the third is the Name of the ID column (in this case "ID") of your table in your database, which will also be used to join to other tables if necessary. In this case, the object 'contacts' will now represent the table called "contacts" in the database.

As you can see in the other 2 lines of code, there is a 4th argument in the instantiation – this is the name of the table (object) that the table in question being created is to be joined to, joined by the column in the third argument. Notice how, as mentioned before, the two columns being joined on from either table have the same name (in this case "ID"). What this actually does is join the 'addresses' table object to the 'contacts' table object. Note that if all your data is contained within one table, for example "contacts", then the second line of code is not necessary and can be removed. If however your data is contained over three tables, then an extra line of code is needed to instantiate the third table as an object and join it to the contacts table. Obviously, the fewer tables there are, the better the performance of the application. It is important to note here also that the application works and joins tables on a one to one relationship i.e. in the example above, there would have to be one address entry for every contacts entry, each pair joined by the common "ID" column.

The last line of code creates an object representing the genkeys table created in the previous step. As you can see from the comment in the code, if the genkeys table is empty, it will not be able to join to the contacts table as there is no genkeys data to join to, which *will* be the case the first time around. So the third line of code should have the join to the contacts table taken out of it, like so.

```
toolkit.Table genkeys = new toolkit.Table( database, "genkeys", "ID" );
```

The other method in the Tasks.cs file that needs to be amended in this case is the de-duplication function itself. As mentioned in the beginning of this guide, you are going to be looking at running a single file de-duplication, for which the associated method is OneFileDedupe() within the class 'Dedupe', contained within Tasks.cs. Below is the default Table object instantiation code –

```
toolkit.Table contacts = new toolkit.Table( database, "contacts", "ID" );
toolkit.Table addresses = new toolkit.Table( database, "addresses", "ID", contacts );
toolkit.Table genkeys = new toolkit.Table( database, "genkeys", "ID", contacts );
```

The only amendment that needs to be made here is to the names of the tables in the second arguments, and removing / adding tables / joins where necessary as described above. Note that in this case, the genkeys table *will* be joined with the 'contacts' table object as it needs to in order to perform the matching / de-duplication.

Another thing to note is that, if the database system that you are using supports creating views, you can of course create the view that you want the columns named how you want and instantiate that as a table object instead. This may be particularly useful if your data is related across tables but not on a one to one relationship (for example, multiple contacts at the same address).

Overview of set-up

Below is a simple check list summarising what has been described above. If you are unsure of a particular step, refer back though the explanation above to be sure as it may cause the application to not work properly in your case.

- Select the type of database being used (Database.cs)
- Map database table fields to the properties of the Record object (Database.cs)
- Amend the CreateKeysTable() method if necessary (Tasks.cs)
- Amend the Table joins in the Import() method (Tasks.cs)
- Amend OneFileDedupe() method if necessary (Tasks.cs)

Running the Application

Once you are happy that all of the above has been carried out, you are in a position to start running the application. The entry point (or Main() method) for the application is in the toolkit_cs.net.cs file. If you open up the file, you will see the following lines of code at the beginning of the main method.

```
string connectionString1 = "";
string connectionString2 = "";
string databaseName1 = "example1";
string databaseName2 = "example2";
int score = -1;
Tasks.Dedupe.OutputType output = Tasks.Dedupe.OutputType.Dialog;
matchIT.Country country = matchIT.Country.UK;
Tasks.Type task = Tasks.Type.OneFileDedupe;
```

The connection strings are automatically generated further down the code so there is not need to worry about them here. In this instance, you are only using one database, so rename databaseName1 as the name of the database in question. The score variable is the minimum score required by a matching pair to appear in the results as a duplicate. For the moment, this can be left as -1 (If the score at this point is set to less than 0, the score value is picked up from elsewhere in the application, namely the BatchProcessor.cs file). The last 3 lines of code are the more interesting ones.

The first is the definition of what kind of output you want to see after the processing has finished. There are 3 choices – Console, Dialog or Table – which are contained in an enumeration and can be viewed in the IntelliSense (assuming the use of Visual Studio) by retyping the ‘dot’ after the word ‘OutputType’. The last option, Table, is only valid for SQL Server, and it produces a table of duplicate pairs that are found along with the score. Console will basically write out the results of the process to the console window, and Dialog will write the results to a windows Dialog box. For the moment it may be easiest to stick with Dialog.

The second line defines the nationality of the data in question. This is again an enumeration, the contents of which can be viewed through the IntelliSense by retyping the ‘dot’ after the word ‘Country’. By default it is set to UK.

The last line specifies the task to be run. A complete list of these jobs is shown below:

- Record
- Query
- CreateKeysTable
- AddKeyFieldsToTable
- Import
- BulkImport
- PurgeExactMatches
- PurgeExactOverlap
- OneFileDedupe
- TwoFileDedupe
- MultiThreadedOneFileDedupe
- MultiThreadedTwoFileDedupe

The jobs in this example that you are interested in are CreateKeysTable, Import and OneFileDedupe. The way the code in this example is set up, you will have to run the 3 jobs, one after the other in that order.

Before you start running the jobs, there is one more thing you will need to amend in this file, and that is the format of the connection string further down in the file, shown below.

```
// Common connection string formats...

// Connection to Access database:
// OLEDB: "Provider=Microsoft.Jet.OLEDB.4.0; Data Source=<Database_Pathname>"
// ODBC: "Driver={Microsoft Access Driver (*.mdb)}; DBQ=<Database_Pathname>"

// Connection to SQL Server database:
//      OLEDB:      "Provider=SQLOLEDB;      Data      Source=<Server_Name>;      Initial
Catalog=<Database_Name>; Integrated Security=SSPI"
// ODBC: "Driver={SQL Server}; Server=<Server_Name>; Database=<Database_Name>"

// Connection to MySQL database:
// OLEDB: "Provider=MySQLProv; Location=<Server_Name>; Data Source=<Database_Name>"
//     ODBC:     "Driver={MySQL     ODBC     3.51     Driver};     Server=<Server_Name>;
Database=<Database_Name>; Option=3"

// Connection to Oracle database:
// OLEDB: "Provider=MSDAORA; Data Source=<Oracle_Database>"
// OLEDB: "Provider=OraOLEDB.Oracle; Data Source=<Oracle_Database>"
// ODBC: "Driver={Microsoft ODBC for Oracle}; Server=<Oracle_Server>.world"

// Use Access+OLEDB...
connectionString1 = "Provider=Microsoft.Jet.OLEDB.4.0; Data Source=" + databaseName1 +
".mdb";
connectionString2 = "Provider=Microsoft.Jet.OLEDB.4.0; Data Source=" + databaseName2 +
".mdb";
```

By default, the connection strings are in Access OLEDB format. If you are not using an Access database, for example if you are using a SQL Server database, you need to amend the connection string using the guidelines in the comments.

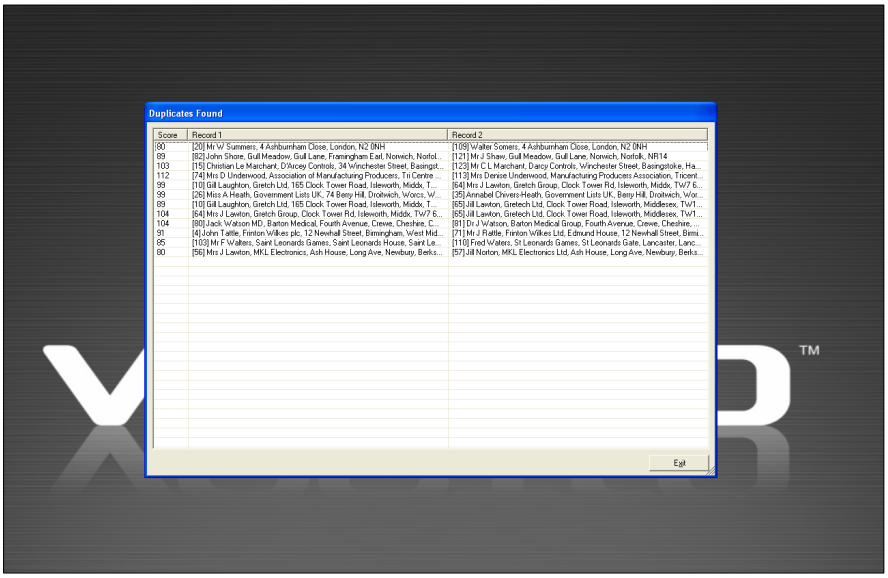
Once you are happy that you are using the correct connection string, you can begin to run the program in the order mentioned above. First run the CreateKeysTable method. After it has finished executing you should see, in whatever database you are using, an empty genkeys table appear, provided no errors have appeared in the console window during the execution of the method. If this is not successful, check the following –

- Your connection string
- Your database name
- Your database definition in Database.cs

Next run the Import() method. When it has finished you should see a summary in the console window of how many keys were generated and imported, as well as seeing on refresh that your genkeys table is now full. If this is not successful, check the following –

- Your table objects / joins in the Generate() method in Tasks.cs
- Your database table field mappings in Database.cs

The last method, OneFileDedupe(), is where records within the database are compared and duplicate pairs are produced. You will remember from earlier that you selected 'Dialog' as the output type, so you should, at the end of the process, see a windows dialogue box with all the resulting duplicate pairs displayed, along with their matching score in the left hand column, a little something like the below.



If for some reason you do not see any results, it could be that the default minimum match score is too high for your data. In this case you could set the default minimum score to 0 to see if anything comes through by amending the line of code setting the score in toolkit_cs.net.cs – change it from -1 to 0. If after that you still get nothing through, check the following –

- Your table objects / joins in the OneFileDedupe() method in Tasks.cs
- Your database table field mappings in Database.cs

When you are getting some results through you can then begin experimenting with the different methods and configurations available in the matchIT SDK. The toolkit_cs.net.cs file has been coded to run one method at a time as it stands purely as a starting point. Once you have a good grasp of what methods are available and how to implement them you can amend the toolkit_cs.net.cs file however you wish, or indeed incorporate parts of the SDK into your own solution.

Another good exercise to carry out to further familiarise yourself with the methods available in the SDK is to get a TwoFileDedupe working – this involves generating keys for the second database and then running the TwoFileDedupe() method with both databases.

Modifying the search weights and keys

Now that you have covered the basics in terms of getting the application set up and communicating with the data, we can move onto some customisation aspects such as what keys to match the data on, and the weights that the various matching parameters carry.

The files that we are concerned with are the following –

- Engine.cs
- BatchProcessor.cs

First of all let's look at the weights of the matching keys, which can be modified in the Engine.cs file. The code sample below shows where the weights are set in this file.

```
#else //BUSINESS_LEVEL

// Individual-level matching...

m_engine.Settings.MatchingRules.IndividualLevel.Weights.Name.SetMatrixWeights(60,    40,
20);

m_engine.Settings.MatchingRules.IndividualLevel.Weights.Name.OneEmptyScore = 20;

m_engine.Settings.MatchingRules.IndividualLevel.Weights.Name.BothEmptyScore = 20;

m_engine.Settings.MatchingRules.BusinessLevel.Weights.Organization.Weight      =      0;
m_engine.Settings.MatchingRules.BusinessLevel.Weights.Organization.OneEmptyScore   =   0;
m_engine.Settings.MatchingRules.BusinessLevel.Weights.Organization.BothEmptyScore = 0;

#endif //BUSINESS_LEVEL

if (country == matchIT.Country.USA)

  m_engine.Settings.MatchingRules.IndividualLevel.Weights.Address.Weight = 40;

else

  m_engine.Settings.MatchingRules.IndividualLevel.Weights.Address.Weight      =      30;
m_engine.Settings.MatchingRules.IndividualLevel.Weights.Address.OneEmptyScore    =    10;
m_engine.Settings.MatchingRules.IndividualLevel.Weights.Address.BothEmptyScore = 10;

m_engine.Settings.MatchingRules.IndividualLevel.Weights.Postcode.Weight      =      30;
m_engine.Settings.MatchingRules.IndividualLevel.Weights.Postcode.OneEmptyScore   =   10;
m_engine.Settings.MatchingRules.IndividualLevel.Weights.Postcode.BothEmptyScore = 10;

m_engine.Settings.MatchingRules.IndividualLevel.Weights.Telephone.Weight      =      0;
m_engine.Settings.MatchingRules.IndividualLevel.Weights.Telephone.OneEmptyScore   =   0;
m_engine.Settings.MatchingRules.IndividualLevel.Weights.Telephone.BothEmptyScore = 0;
```

What these settings will do is override the default weight settings of the matchIT API. As you can see from the code, there are 5 areas that scores are set on – Name, Organization, Address, Postcode and Telephone. In the case of the first two, Name and Organization, you can set what is known as the MatrixWeight manually using the SetMatrixWeight function. What this does is set the scoring for a **sure**, **likely** and **possible** match with the first, second and third parameters respectively. Note that if the SetMatrixWeights() method is not used, and instead the Weight property is set (as for the organization weight above), the value given will be

assigned to the **sure** match property. By default, the **likely** and **possible** properties will then be worked out as 2/3 and 1/3 of the **sure** value respectively. Remember that the SetMatrixWeights() function is *only* available for Name and Organization Matching. As you can see in the code also, all 5 can have a score configured for OneEmptyScore and BothEmptyScore properties. These matching weights can be configured as you like to make particular matches stand out far above the rest.

Secondly let's look at what are known as the matching keys. In simple terms, the matching keys are a combination of columns generated by the matchIT API that are used to group records from the database into clusters of *potential* matches for further processing / comparing. The keys to be used are specified in the BatchProcessor.cs file. The code that specifies the keys to be used in the initial pairing up process is shown below.

```
// Individual-level match keys for UK and other data with low level postal codes...
return new CompositeKey[]
{
new CompositeKey( Keys.PostOut, Keys.Name1 ),           //PostOut+PhoneticLastName
new          CompositeKey(          Keys.Name1,          Keys.PhoneticStreet          ),
//PhoneticLastName+PhoneticStreet
       new CompositeKey( Keys.Postcode )
       //Postcode
};
```

In simple terms, the database is looped through and pairing occurs for each key specified in the composite keys array. So in the case above, there would be 3 loops, the first one grouping records with the same PostOut and PhoneticLastName, the second grouping records with the same phoneticLastName and PhoneticStreet, and the third grouping records with matching postcode.

To see a list of keys that are available, you can look in the Keys.cs file, where you will find a list of keys in a class called 'Keys' defined in the following format -

```
public static Key NameKey = new Key( "NameKey",
                     new Record.GetValueFn( Record.GetNameKey ),
                     new Record.PutValueFn( Record.PutNameKey ) );
```

For example, there is a key available called OrganizationKey, which is not used by default in the code. So if you wanted to just initially match data on the company name, you would amend the CompositeKey[] array shown above as follows.

```
// Individual-level match keys for UK and other data with low level postal codes...
return new CompositeKey[]
{
new CompositeKey( Keys.OrganizationKey ),
//Organization
};
```

This would just loop the database once, grouping records that have a matching organization key. Notice how in the above example the CompositeKey now only has one argument. In fact, the CompositeKey elements of the CompositeKey[] array and have as little as 1 or as many as 8 key arguments to match data on. For example, if you wanted to match data on PostOut, Name1 and OrganizationKey, you would set the CompositeKey up as follows.

```
// Individual-level match keys for UK and other data with low level postal codes...
return new CompositeKey[]
{
new CompositeKey( Keys.PostOut, Keys.Name1, Keys.OrganizationKey ),
// PostOut+PhoneticLastName+Organization
};
```

A consequence of this would be a much narrower search than if you were just matching on organization. Clusters of potential matches will become smaller and performance could improve, but with increased risk of missing duplicate records.

The above is just a brief explanation of what is possible to configure with the matchIT SDK. It is also possible to go much deeper into the configuration and specify custom keys to match data on.

Technical Reference

This chapter guides you through each of the core classes contained in the matchIT SDK describing what each class does and how it operates.

Classes

The classes in the SDK have been designed with reusability in mind. They can easily be reused in applications (C++ or VB.NET) with minimal code changes required, and can easily be customized to work with different databases. Fields can be easily renamed (for example, if 'Addressee' is used instead of 'FullName'), and new fields can be added if they're not currently supported.

The sample code demonstrates how the classes can be used, such as how to open tables on the example database, how to search for records that phonetically match a surname, how to batch process a database, and how to implement progress and/or results dialogs.

Classes that wrap the matchIT API

Engine

This class instantiates a matchIT API engine, which is responsible for providing the underlying functionality of the matchIT SDK (key generation, record comparison). The class illustrates how to change various engine settings including, for example, the weights used to calculate record comparison scores.

Record

This class instantiates a matchIT API record, containing the field values of a row from a set of joined tables in a database. A record therefore represents either a contact or a company.

Classes that manipulate databases

Database

This class instantiates an ADO Connection object, which is a connection to a local or remote database.

Table

This class instantiates an ADO Recordset object, which represents a table in a database. Other tables from the database can be joined to this table.

Query

This class instantiates an ADO Recordset object, into which are retrieved the results of a query on the joined tables. This is used to find records in the database that match a given query using match keys stored in the database.

Classes for generating keys

Generator

This class generates matching and key fields for each record. Records are read from the query in blocks, then batch generated using a technique similar to the BatchProcessor, then matching and key fields are written to the table.

Note that the Generator is set up to write matching and key fields only for maximum performance. For cleansing and standardisation of names and addresses etc. this class would need modifying.

Key generation is significantly slower than finding duplicates. However, key generation only needs to be performed once on a table. When adding new records to such a table, either via a standalone application or merged from another table, each record's keys can be generated and written along with each new record.

BulkGenerator

Operates similarly to the Generator class, except that records are read sequentially and the generated matching and key fields are written to a temporary file. Then a Bulk Insert SQL command is used to import the data into a blank keys table. This significantly reduces the time taken for key generation, particularly when used with large databases.

Note, however, that the original data fields are not modified (i.e. no cleansing, standardisation, or casing). To do this the BulkGenerator will need to output all the relevant data from each record, and then perform a Bulk Insert of this data into a new (empty) table. This new table should replace the existing table(s), or the deduplication process can then be used on this cleansed data instead of the original data.

Classes for finding exact matches

PurgeExactMatches

Implements an algorithm for finding *exactly* matching records within a single table (or set of joined table). One search key is used to locate clusters of records. The matchIT API is not used to compare records in this process, as records that share the same key value are deemed exact matches.

PurgeExactOverlap

Implements an algorithm for finding *exactly* matching records that overlap two distinct tables (or sets of joined tables). Records can be purged from the first or second table (for example, if the second table is to be imported into the first, exact matches can be purged from the second table before *fuzzy* deduplication and purging is performed; then, remaining records can be imported into the first table).

Classes for finding fuzzy matches

OneFileDedupe

This class implements a batch processing algorithm that finds duplicate record pairs within a single database.

TwoFileDedupe

This class implements a batch processing algorithm that finds duplicate record pairs across two databases (for example, to identify the overlap when feeding a new batch of records into a master database, or to suppress existing customers from a mailing).

MultiThreadedOneFileDedupe

A multithreaded version of the OneFileDedupe class.

MultiThreadedTwoFileDedupe

A multithreaded version of the TwoFileDedupe class.

Classes for processing results

GroupMatches

This class can be run after finding fuzzy matches, using OneFileDedupe, to group matching record pairs into *sets* of matches. For example, if record A matches B, and B matches C, then the three records will be grouped into one set. All matching record pairs are written to the ‘matches_grouped’ table; each set is idenitified by a unique MatchRef.

GroupOverlap

As per the GroupMatches class, but performed after using TwoFileDedupe. Note that sets are identified by the Record2 column – i.e. the record from the second table.

Classes that implement dialogs

ProgressDialog

This class implements a dialog that displays progress information as a database is being batch processed. It demonstrates how to run the batch processor in a separate thread, and how to receive and respond to event notifications from the batch processor.

ResultsDialog

This class implements a dialog that displays a list of duplicate record pairs after batch processing has completed, along with the matching scores as determined by the matchIT API.

Reference

BulkGenerator

Operates similarly to the Generator class, except that records are read sequentially and the generated matching and key fields are written to a temporary file. Then a Bulk Insert SQL command is used to import the data into a blank keys table. This significantly reduces the time taken for key generation, particularly when used with large databases.

Note, however, that the original data fields are not modified (i.e. no cleansing, standardisation, or casing). To do this the BulkGenerator will need to output all the relevant data from each record, and then perform a Bulk Insert of this data into a new (empty) table. This new table should replace the existing table(s), or the deduplication process can then be used on this cleansed data instead of the original data.

Usage

Ideally create a Generator class in a worker thread of an application (to ensure a responsive UI), using an event sink to receive notification events from the Generator.

Methods

Import

Uses the Engine to generate the keys and matching fields for each record read from the specified table (or joined tables). The keys and matching fields are written to a temporary file which, once all records have been generated, is bulk-imported into the specified keys table. No input data is modified.

Properties

ProgressInterval

The class will report the current processing progress at regular intervals – e.g. every thousand records it will call Sink.Progress() to report how many records have so far been processed. This can be used to display a progress bar on a form or dialog, for example. (Setting to 0 prevents the progress from being reported.)

DefaultProgressInterval

A read-only property that returns the default progress interval.

Database

This class instantiates an ADO Connection object, which is a connection to a local or remote database.

Usage

Create the Database and connect using a connection string and username/password if required. Any type of database can be connected to – for example Access, SQL Server, MySQL, or Oracle – provided there is a driver installed on the system for it.

Methods

Connect

Establish a connection to a database with a connection string, username, and

password.

Disconnect

Disconnect from the connected database.

ExecuteSQL

Execute a SQL statement on the connected database.

Properties

Handle

Returns a handle to the wrapped ADO Connection object.

Engine

This class instantiates a matchIT API engine, which is responsible for providing all the underlying functionality of the matchIT SDK (key generation, record comparison, data cleansing). The class illustrates how to change various engine settings including, for example, the weights used to calculate record comparison scores.

Usage

An Engine must be created for each thread that will use the matchIT API. If using C++, remember to call CoInitialize to initialize COM before the Engine is created.

Methods

None

Properties

Handle

Returns a handle to the wrapped matchIT API engine instance.

Country

Returns the country enumeration value that the engine has been initialized using. This specifies the main nationality of the data that the engine will operate on.

Generator

This class generates matching and key fields for each record. Records are read from the query in blocks, then batch generated using a technique similar to the BatchProcessor, then matching and key fields are written to the table.

Note that the Generator is set up to write matching and key fields only for maximum performance. For cleansing and standardisation of names and addresses etc. this class would need modifying.

Key generation is significantly slower than finding duplicates. However, key generation only needs to be performed once on a table. When adding new records to such a table, either via a standalone application or merged from another table, each record's keys can be generated and written along with each new record.

Usage

Ideally create a Generator class in a worker thread of an application (to ensure a responsive UI), using an event sink to receive notification events from the Generator.

Methods

Import

Uses the Engine to generate the keys and matching fields for each record read from the specified table (or joined tables), writing them to the keys table after each block of records has been generated. No input data is modified.

Properties

ProgressInterval

The class will report the current processing progress at regular intervals – e.g. every thousand records it will call Sink.Progress() to report how many records have so far been processed. This can be used to display a progress bar on a form or dialog, for example. (Setting to 0 prevents the progress from being reported.)

DefaultProgressInterval

A read-only property that returns the default progress interval.

OneFileDedupe

This class implements a batch processing algorithm that finds duplicate record pairs within a database.

Usage

Ideally create a OneFileDedupe class in a worker thread of an application (to ensure a responsive UI), using an event sink to receive notification events from the batch processor – the ProgressDialog class demonstrates how this may easily be done.

Methods

Dedupe

Runs the batch processor on the specified database.

Properties

MinimumMatchScore

Specifies the minimum score used when comparing records that share a common match key. All matching record pairs, with a score that equals or exceeds the minimum score, are added to the list of duplicate pairs. Note that the higher the minimum score, then the less chance of a false positive occurring (i.e. a duplicate pair not in fact being duplicate records) but also a greater chance of a legitimate duplicate pair not being found.

DefaultMinimumMatchScore

A read-only property that returns the default minimum match score.

MaximumClusterSize

When batch processing with a single database, records are grouped into ‘clusters’ that share a common key. Each record in the cluster is compared with every other record in the cluster (for example, 1-2, 1-3, 1-4, 2-3, 2-4, and 3-4). The maximum cluster size restricts the number of comparisons performed in a cluster, so as to prevent the batch processing taking too much time in the event of an unexpectedly high number of records for the same match key value – this can happen due to e.g. bad data such as non-null dummy values in a key field, or (where address information forms all or part of the match key) high volumes of records in the database leading to high geographic concentration of records. (Setting to 0 prevents such restrictions from being enforced.)

If the batch processor finds a key with more records than the MaximumClusterSize value, it logs this information (via the attached event sink) and proceeds to the next cluster.

For example: if you have MaximumClusterSize set to 200, you are matching on PhoneticLastName+PostOut and there are more than 200 records for postal code 10026 and PhoneticLastName equal to "dyvys" (e.g. Davies or Davis): the batch processor will compare the first such record with all the others, log the number of occurrences, then carry on with the next record with a higher key.

If you have a database which gives rise to cluster counts that exceed MaximumClusterSize for all or most of the match keys used, you should either increase MaximumClusterSize or use more precise match keys which lead to smaller cluster sizes. If you have a database with a large number of exact matches e.g. a bookings database in which the name and address was captured again for every order, you can use a very precise match key, such as NormalizedName + Gender + PostOut + PostIn + Premise (from Address.Elements.Premise) and set a MaximumClusterSize value of 2 – in this scenario, all the matches will be found by comparing just the first record in the cluster with each of the other records, as they will all be matches of the first record.

DefaultMaximumClusterSize

A read-only property that returns the default maximum cluster size.

ProgressInterval

The batch processor will report the current processing progress at regular intervals – e.g. every 10 records it will call Sink.Progress() to report how many records have so far been processed. This can be used to display a progress bar on a form or dialog, for example. (Setting to 0 prevents the progress from being reported.)

DefaultProgressInterval

A read-only property that returns the default progress interval.

ProgressDialog

This class implements a dialog that displays progress information as a database is being batch processed. It demonstrates how to run the batch processor in a separate thread, and how to receive and respond to event notifications from the batch processor.

Usage

Create the dialog and display using either the C++ DoModal method or the .NET ShowDialog method. Pass the necessary parameters (including the table(s) to be batch processed, and the list that will receive the duplicate record pairs) into the dialog's constructor.

Methods

None

Properties

None

PurgeExactMatches

Implements an algorithm for finding *exactly* matching records within a single table (or set of joined table). One search key is used to locate clusters of records. The matchIT API is not used to compare records in this process, as records that share the same key value are deemed exact matches.

Usage

Ideally create a PurgeExactMatches class in a worker thread of an application (to ensure a responsive UI), using an event sink to receive notification events from the class.

Methods

Dedupe

This runs the exact matching algorithm on the specified table (or joined tables). An exact matching key is used to group all records into clusters; multiple records in the same cluster (i.e. same key value) are deemed exact matches.

Properties

None

PurgeExactOverlap

Implements an algorithm for finding *exactly* matching records that overlap two distinct tables (or sets of joined tables). Records can be purged from the first or second table (for example, if the second table is to be imported into the first, exact matches can be purged from the second table before *fuzzy* deduplication and purging is performed; then, remaining records can be imported into the first table).

Usage

Ideally create a PurgeExactOverlap class in a worker thread of an application (to ensure a responsive UI), using an event sink to receive notification events from the class.

Methods

Dedupe

This runs the exact matching algorithm across the specified tables (or joined tables). An exact matching key is used to group all records into clusters; multiple records in the same cluster (i.e. same key value) are deemed exact matches.

Properties

None

Query

This class instantiates an ADO Recordset object, into which are retrieved the results of a query on the joined tables. This is used to find records in the database that match a given query using match keys stored in the database.

Usage

After instantiating a Query object, construct the search condition and pass this to the BeginQuery method. A list of matching records can then be traversed in a loop. Start with GoToFirstRecord, then repeatedly call GetCurrentRecord and GoToNextRecord until one of these methods fails (i.e. no more records remain).

Methods

BeginQuery

Begins a query with a given condition. The query takes the form of a SQL Where condition (for example, Name1+PostOut='brytlyEH9'), which is used by this method to construct a SQL Select statement that joins all tables and appends the condition. The query is then executed and the results (i.e. records that match the search keys) go into the wrapped ADO Recordset object.

EndQuery

This must be called to end the current query.

GoToFirstRecord

Repositions the cursor to the first record in the Recordset.

GetCurrentRecord

Retrieves the fields of the current record into the given Record object.

GoToNextRecord

Advances the Recordset cursor to the next found record.

Properties

Handle

Returns a handle to the wrapped ADO Recordset object.

RecordCount

Returns the total number of records in the query.

EndOfFile

Returns true if the cursor has reached the end of the query.

Record

This class instantiates a matchIT API record, containing the field values of a row from a set of joined tables in a database. A record therefore represents either a contact or a company.

Usage

A record can be created with or without a parent Engine. If no engine is used then neither the Generate or Compare methods can be called on this record. Note that to compare two records, at least one of them must have a valid engine handle.

Note also that if your database contains fields that aren't currently contained in the Record class, then you'll have to add the field to the record and add the field mapping to the Database class.

Methods

Clear

Blanks all the fields of the record.

Apply

Applies (writes) all the fields of this record to the wrapped matchIT API record. This must be done prior to calling the Compare() method only if a Generate() has not been called on this record - for example, when comparing two records, one of which has been read from fields in a database's table; in this case Generate() has been called in a previous 'generation' stage so it is not therefore necessary to call Generate() a second time, in which case use Apply() instead.

Generate

Processes the fields of the record, which includes key generation, name and address element extraction, and data cleansing and reformatting. Usually the generated keys are used to search for records in a database, and is performed by a preparatory 'key generation' application.

Compare

Compares two records, both of which must have been previously generated (or generated and subsequently applied), and returns the comparison scores (name, organization, address, postcode, telephone, total).

Properties

FullName, Organization, NameKey, etc.

Accessors used for setting and retrieving the field values of a record.

ResultsDialog

This class implements a dialog that displays a list of duplicate record pairs after batch processing has completed, along with the matching scores as determined by the matchIT API.

Usage

Create the dialog and display using either the C++ DoModal method or the .NET ShowDialog method. Pass the list of duplicate record pairs into the dialog's constructor.

Methods

None

Properties

None

Table

This class instantiates an ADO Recordset object, which represents a table in a database. Other tables from the database can be joined to this table.

Usage

After creating a Database, create one or more named Tables from the Database. Note that a table can easily be joined to another during creation, and that adding, updating, and deleting records in a table will automatically account for joined tables.

Methods

Connect

Opens a table (via a Recordset) in a connected database.

Disconnect

This must be called to close the open table (by default a Database object will call Disconnect() on all its open tables, so it's not normally necessary to manually close each table).

AddRecord

Adds the given record to the table plus any joined tables.

UpdateRecord

Updates the given record to the table plus any joined tables.

DeleteRecord

Deletes the given record from the table plus any joined tables.

Properties

Handle

Returns a handle to the wrapped ADO Recordset object.

Name

Returns the name of the table.

UniqueRefFieldName

Returns the name of the field used to uniquely reference rows of this table.

Database

Returns a reference to the Database that contains this table.

JoinedTableCount

Returns a total recursive count of all the tables joined to this table.

JoinedTables

Returns a list of tables joined directly to this table.

TwoFileDedupe

This class implements a batch processing algorithm that finds duplicate record pairs across two databases (for example, to find the overlap between two independent databases, or to find matching records using a suppression file).

Usage

Ideally create a TwoFileDedupe class in a worker thread of an application (to ensure a responsive UI), using an event sink to receive notification events from the batch processor – the ProgressDialog class demonstrates how this may easily be done.

To find records that overlap two databases, simply pass the two databases to the constructor. To find records using a reference suppression file, make sure the suppression database is specified first.

Methods

Dedupe

Runs the batch processor on the specified databases.

Properties

MinimumMatchScore

Specifies the minimum score used when comparing records that share a common match key. All matching record pairs, with a score that equals or exceeds the minimum score, are added to the list of duplicate pairs. Note that the higher the minimum score, then the less chance of a false positive occurring (i.e. a duplicate pair not in fact being duplicate records) but also a greater chance of a legitimate duplicate pair not being found.

DefaultMinimumMatchScore

A read-only property that returns the default minimum match score.

MaximumClusterSize

When batch processing with two databases, records from the second database are grouped into ‘clusters’ that share a key with each record from the first database. That record is then compared with every record in the cluster. If the number of comparisons exceeds the maximum cluster size then, so as to prevent the batch processing taking too much time, comparison of the current cluster will stop when (or if) a matching pair is found. (Setting to 0 will prevent such restrictions from being enforced.)

Further information about use of cluster size limits can be found in the description of OneFileDedupe. The difference here is that as the typical scenario is comparing a customer or marketing file with a suppression file, processing stops as soon as a match is detected i.e. the record in the customer/marketing file is marked for suppression.

DefaultMaximumClusterSize

A read-only property that returns the default maximum cluster size.

ProgressInterval

The batch processor will report the current processing progress at regular intervals – e.g. every 10 records it will call Sink.Progress() to report how many records have so far been processed. This can be used to display a progress bar on a form or dialog, for example. (Setting to 0 prevents the progress from being reported.)

DefaultProgressInterval

A read-only property that returns the default progress interval.