matchIT SDK for SQL Server

Information Pack

Contents

Introduction

Welcome to the matchIT SDK for SQL Server Product information pack. This product was originally developed as a starting point to allow developers to integrate the matchIT API with existing applications and run batched data cleansing. However, the main core of the matchIT SDK for SQL Server functionality is now configurable through an XML file, so that SQL Server DBA skills and permissions rather than programming knowledge are sufficient to implement the matchIT SDK as a stored procedure within your SQL Server database. Should you need to modify the SDK source code to suit requirements not configurable via XML (such as customized data treatment), knowledge of C# will be required.

The matchIT SDK for SQL Server is a supplement to the matchIT SDK, enabling fast finding of matching records from within SQL Server 2005 databases.

This is achieved by making use of *SQL CLR*, a feature of SQL Server 2005 that allows developers to easily write stored procedures and functions for SQL Server using .NET languages.

The matchIT SDK, on the other hand, uses ADO to connect to any database (including SQL Server) to find duplicate records. The matchIT SDK for SQL Server offers significantly increased performance, because stored procedures are run in-process (i.e. within the SQL Server service) and the overhead of the ADO layer is thus eliminated.

The SDK consists of a Visual Studio 2005 project template containing C# (.NET) and unmanaged C++ source code. C# is used to implement stored procedures in a .NET class library for functionality including key generation, finding duplicate records in a table, and finding records that overlap two tables. C++ is used to implement an unmanaged DLL for batch generation and comparison using the matchIT API (a COM component).

The heart of the SDK for SQL Server is the XML configuration file which is used to modify settings of the matchIT API and to map your data to the matchIT Record object. The XML Configuration file is used in all CLR procedures that involve working with your data, and is described in more detail below. The source code of the SDK is fully functional and can be used with minimal necessary changes. There are no restrictions on its use; it can be used freely either on its own or as part of a larger application.

This guide should be read in conjunction with the documentation for the matchIT SDK and matchIT API and has been split into the following three chapters.

- Getting Started 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;
- Technical Reference – technical reference describing each of the classes which can be found in the SDK code;
- Troubleshooting – this section will help you with the most common error messages you are likely to see when working with the SDK.

Getting Started Tutorial

Introduction

The matchIT SDK for SQL Server comes with some pre-written SQL scripts to create tables and stored procedures, and also run various de-duplication methods on the example data included. Importing this example data and running the scripts is explained in the following section, 'Quick start with example data'.

A guide on modifying the XML Configuration file and how to set up the SDK to communicate with your own data is set out in the section 'Getting set up with your own database schema'. The guide will focus on setting up the SDK to perform a single file de-duplication on your own data.

Quick start with example data

The easiest way to run the SDK with the example data is to run the two scripts, 'Import Example Data' and 'Dedupe Example', which can be accessed through the Windows start menu (Start > Programs > matchIT SDK for SQL Server > Demo) once the application has been installed. Run them in order - The first will install the example database with the relevant tables, whilst the second script will actually perform the de-duplication process. Clicking on either script from the Start Menu will automatically open it in SQL Server management studio (provided that is how your .sql file association is set up) from where it can be executed. The second script will first run a single de-duplication on the first set of data, example1, producing a results table called matches_found, and then perform a two-file overlap between the example1 and example2 data, producing a results table called overlap_found. The contents of both are displayed in the results pane after the script has run.

Note that before running the second script, you will have to make a small amendment to the XML configuration file that was installed in 'C:\Program Files\matchIT SDK for SQL Server\demo' called Config.xml. As mentioned before, this file is like the heart of the SQL SDK, and this will be your first interaction with it. Open it up and navigate to the <datasources> node, which contains two <datasource> sub nodes. Within each of these sub nodes, the very first node that appears is the following –

<nonContextConnectionString value="Data Source=****; Initial Catalog=matchIT_API_demo; User ID=****; Password=****;" />

All you will need to do in this instance is amend the connection string to contain the correct parameters (note that the initial catalog value is already complete and matches the database name in the Import Example Data script.) Do this for both data sources.

The other way to import the example data and run the de-duplication processes is to simply browse to the scripts in SQL Server Management Studio manually. All scripts are located in 'C:\Program Files\matchIT SDK for SQL Server\demo'. When you browse to the location you will see that there are a wider variety of scripts available to run than are displayed in the start menu. To install the data, you need to run 'ImportExampleData.sql' to actually install the database. After that you can then run the script of your choice to perform the de-duplication you want. 'FindMatches.sql' will run a single file de-duplication on the example1 data, 'FindOverlap.sql' will run a two-file de-duplication on the example1 data and example2 data, and 'DedupeExample.sql' will run both. The same applies to running FindMatches.sql and FindOverlap.sql in that you will need to amend Config.xml as mentioned above before running them.

Getting set up with your own database schema

Now let's take a closer look at the XML Configuration of the SQL SDK, namely the file Config.xml located in 'C:\Program Files\matchIT SDK for SQL Server\demo'. It is easiest to break down the XML file into sections as they appear in the file itself.

Match Keys

The first section that appears in the XML configuration file is the <matchKeys> node, within which there are definitions for 3 types of keys - <fuzzyKeys>, <exactKeys> and the <duplicatePreventionKey>.

By default, three 'Composite keys' are defined for the <fuzzyKeys> node - A composite key is a collection of single keys that are used together in a query to search for data. You can add or remove composite keys (note that one search is done per composite key, so more composite keys means more processing time) by adding or removing <key> nodes within the <fuzzyKeys> node, as well as modifying the single keys of the <key> nodes themselves. The <fuzzyKeys> are used in the fuzzy matching processes such as findMatches and findOverlap.

Also by default, three 'Composite keys' are defined for the <exactKeys> node – They can be modified in exactly the same way that the <fuzzyKeys> are, and are used in the exact matching processes, such as findExactMatches and findExactOverlap.

The <duplicatePreventionKey> node is slightly different in that it can only contain one composite key, however the composite key it contains can be modified in exactly the same way as the others, by adding or removing 'keyX' attributes (where X is an integer).

It is also worth noting here that it is possible to use matchIT record fields as search keys as well as key fields themselves (i.e. fields with the 'mk' prefix) – This can be seen in the default UK keys in that 'Postcode' is used in the last composite key of the fuzzyKeys section. Note however that if there is no field mapping defined for a particular matchIT record field (discussed in the <datasource> section below), the key will not be able to be used as a match key, and an exception will occur if it is.

Note – It is possible to declare sql string functions with the key definitions, that is, LEFT, RIGHT and SUBSTRING. If you wanted to use the fist 2 letters of post out for example as a match key, you would set the 'keyx' attribute to be LEFT(mkPostOut, 2). You would do the same with the RIGHT and SUBSTRING functions (with SUBSTRING taking a third counter argument).

General Settings

The <generalSettings> section of the configuration contains nodes that define settings for various methods of the SDK. Explanations for each are listed below –

<progressInterval> sets the length of time (in milliseconds) between progress reports sent to the sql pipe for various methods in the SDK.

<tempFileDirectory> sets the directory that files such as the bulk generation file and log files are created in. If left blank, the windows Temp folder is used as a default.

<threadCount> sets the number of threads to run in the multithreaded procedures such as multiThreadedFindMatches and multiThreadedFindOverlap.

<recordBufferSize> sets the value (in bytes) of the buffer size to contain a matchIT record in string form, set by default to 1 MB.

<maxClusterSize> sets the maximum size that a cluster can be before being reported as a large cluster. Clusters are the groupings of records created by the initial matching done on the match keys.

<minimumIndividualScore> sets the minimum score for a match to pass at individual level.

<minimumFamilyScore> sets the minimum score for a match to pass at family level.

<minimumHouseholdScore> sets the minimum score for a match to pass at household level.

<minimumBusinessScore> sets the minimum score for a match to pass at business level.

<minimumCustomScore> sets the minimum score for a match to pass at custom level.

Output Settings

The <outputSettings> node simply contains custom definitions of two standard tables that are produced by the SQL SDK (namely the matches and matches_grouped) that can be created from the standard tables by running the relevant procedure. (In the case of producing a custom matches table, you would run msp_CreateCustomMatchesTable, and for the matches_grouped table you would run msp_CreatedCustomGroupedMatchesTable). For both the <matchesOutputTable> and <groupedMatchesOutputTable> nodes, you must define a custom table title with the 'customTitle' attribute, and then for each column subnode in either definition it is possible to define a custom column name. If you want to leave one of the columns from the standard tables out of your custom definition, simply leave the 'customName' attribute blank.

Data Sources

The <dataSources> node is where the mapping of your data takes place. The <dataSources> node can contain a maximum of 2 <dataSource> sub nodes, within which you define your sources that you want to operate on. Each <datasource> node must have an 'id' attribute specified as it is used when calling procedures to specify which data source you want the procedure to work on.

The first sub node within the <dataSource> node is the <nonContextConnectionString> node, which contains a connection string to be used when there is no context connection available. By default, as mentioned in 'Quick Start with Example Data', it has blank credentials and points to a database called 'matchIT_API_demo', which is the sample database created by the SDK for use with the sample SQL scripts. You need to amend the connection string as necessary to point to your data.

The next node is the <purgeFrom> setting, which must be set to true for only one of the data sources when running purgeExactOverlap. If you were running an overlap between data source 1 and data source 2, and data source 1 had its <purgeFrom> node set to true, then matches (overlaps) found in data source 1 will be treated as records to 'delete'.

Following the <purgeFrom> node is the <tables> node. This node contains all the definitions of the tables within your database that contain the data to be worked with. A single table is defined in a <table> node. The <table> node can contain any of the following attributes –

- name – The name of the table in the database (mandatory for all tables).
- uniqueRef – The name of the unique reference column in the table (mandatory for all tables).
- join – The name of the table the that the table in question joins to (the main table of data will not have this attribute).
- joinType – The type of join to use between the table in question and the table it is joining to – either INNER, LEFT or RIGHT. Uses LEFT by default.
- JoinColumn – The name of the column in the join table that the unique ref column of the table in question joins with (mandatory for any table definition with a join attribute).
- isKeysTable – Indicates that the table in question is the keys table definition (can only have one keys table definition per data source, and it must join to the main table of data)
- tableHints – Here you can specify a comma delimited list of Table Hints that will be used in SELECT queries for the table in question, for example 'NOEXPAND' if the table you have defined is an indexed view and you want the query to reference the view rather than its base tables.

Within a <table> node, it is possible to have multiple <conditionalColumn> sun nodes that define columns in the table in question that must meet a condition for a record to be included (for example, a 'deleted' flag column). A <conditionalColumn> node must have the following attributes defined –

- name – The name of the column in the table.
- isEqualTo – Specifies whether the value should or should not be equal to the value.
- value – The value of the flag column (linked to the isEqualTo attribute as mentioned above).
- isIntegerType – Specifies whether the column is an integer type or not.

One thing to note about the <tables> section is that SQL server supports views, so it is possible to use the name of a view for the 'name' attribute of a table node.

The last node within the <dataSource> node is the <fieldMappings> node - It contains <fieldMapping> sub nodes that define how fields in your data map to the matchIT record object. The 'matchITField' attributes of these nodes are pre-set and must not be changed – The attribute that you will need to modify is the 'databaseField'. Simply put the values of the 'databaseField' nodes to the names of the fields in your data that best match the corresponding 'matchITField'. Any matchITFields that do not have an equivalent in your database should have their databaseField attributes left blank.

Note that it is also possible to specify custom fields that do not have a specific matchIT record equivalent, that you wish to match on, to any of the 9 generic custom fields that are available in the matchIT record object. By default, a there is a node included in the xml on installation with its matchITField attribute set to 'CustomField1' – You can in fact have 9 nodes in total with their matchITField attribute values ranging from CustomField1 to CustomField9. A database field such as 'National Insurance Number' could be mapped to one of these fields.

MatchIT API Settings

The <matchITAPISettings> node defines all the settings related to the matchIT API. Perhaps the one of the more important nodes in this section is the first node - <nationality>. This node needs to be set to whatever the nationality of the data is that you are working on.

The structure of the XML within the <matchITAPISettings> node follows the COM hierarchy that is described in the matchIT API guide. Please refer to the matchIT API guide for more detail on the settings and properties of the matchIT API and what they mean.

Overview of XML Configuration File

Below is a simple list summarising the sections of the XML Configuration that have been described above. If you are unsure of a particular section, refer back though the explanation above to be sure as it may cause the application to not work properly in your case if not configured correctly.

- Match Keys – Define the keys that you wish to use for fuzzy matching, exact matching and duplicate prevention matching.
- General Settings – Define the various general settings you wish to use during processing.
- Output Settings – Define the schemas you wish to use to produce custom 'matches' and 'grouped matches' tables from the standard ones produced by the SDK.
- Data Sources – Defined the tables and mappings for your database(s) here.
- matchIT API Settings – Configure the matchIT API settings you wish to use in this section.

Lastly, there are a couple of amendments that need to be made to the sql file FindMatches.sql. If you open the file in SQL Server Management Studio, the first line that you need to amend is the following:

```
USE matchIT_API_demo

GO
```

Simply change this to the name of your database. The only other lines that you need to modify are the following:

```
EXEC msp_CreateKeysTable @xmlConfigurationFilePath='C:\Program Files\matchIT SDK for SQL Server\demo\Config.xml', @dataSourceID='1'
EXEC msp_BulkGenerateKeys @xmlConfigurationFilePath='C:\Program Files\matchIT SDK for SQL Server\demo\Config.xml', @dataSourceID='1'
EXEC msp_FindMatches @xmlConfigurationFilePath='C:\Program Files\matchIT SDK for SQL Server\demo\Config.xml', @dataSourceID='1'
EXEC msp_GroupMatches @xmlConfigurationFilePath='C:\Program Files\matchIT SDK for SQL Server\demo\Config.xml', @dataSourceID='1', @level='individual'
```

Simply substitute '`@dataSourceID='1'`' for whichever data source you wish to perform the procedures on and '`@level='individual'`' for whichever level you wish to group the matches at (with the levels being individual, family, household, business and custom).

Note also that the following line

```
CREATE ASSEMBLY StoredProcedures FROM 'C:\Program Files\matchIT SDK for SQL Server\demo\StoredProcedures.dll' WITH PERMISSION_SET = UNSAFE
```

Will need to be changed if you make any amendments to the source code of the SDK and rebuild the StoredProcedure.dll to the path to the newly rebuilt dll. By default on installation a copy of a dll of the compiled source code is installed to the demo folder and all scripts reference it.

Note here how all the procedures listed above accept 2 common parameters – one is the file path to the configuration file with your desired set up for the process, and the other is the id of the data source in the configuration for the procedure to run on.

Running the application configured to your data

Once you are happy that all of the necessary changes have been carried out, you are ready to execute the sql script FindMatches.sql in SQL Server Management studio.

As you will be able to see from the script, and as is shown in the code snippet above, this script runs 4 stored procedures. First the msp_CreateKeysTable procedure will run, creating the keys table called by whatever name it was given in the specified data source in the XML. Secondly, the msp_BulkGenerateKeys procedure will generate the keys for the data and populate the keys table. Thirdly, the msp_FindMatches procedure will run and produce 2 tables – 'large_clusters' and 'matches' in the specified data source. The former table stores logs of any instances where a cluster of matches has exceeded the default maximum cluster size (set in the <generalSettings> section of the XML). The latter table stores records of matches that occur, their associated scores at each matching level as well as a column containing an integer indicating which levels the matches passed at (with regards to the minimum scores set in the <generalSettings> section). The value in the last column is made up from the sum of values which correspond to the levels at which the match passed at, which are as follows

- Inividual = 1
- Family = 2
- Household = 4
- Business = 8
- Custom = 16

So for example, if a match contained a value of 9 in this column, that would mean that it passed at both individual and business level (1 + 8).

Any errors that occur with procedures will appear in the output pane in SQL Server Management Studio. From the error message you will be able to track down what the error applies to, and modify the configuration / sql script accordingly.

Once you have got this working and have a good idea about the configuration, you can have a look at running the other .sql scripts that are available in the demo folder, namely DedupeExample.sql and FindOverlap.sql. You will see in each script from the lines that start with `EXEC` what stored procedures are used. Note that in the case of overlapping in these two scripts, both data sources in the configuration are used, and you have the freedom to specify which data source to use as the main file and which one to use as the overlap.

Depending on your situation, you may wish to amend the sql scripts provided or write your own sql script from scratch to process your data. An overview of all stored procedures available is available in the Technical Reference section.

Triggers

You may wish to have a system in place that keeps the keys table in a <dataSource> produced by the SQL SDK kept in sync and up to date with any changes that occur within the tables of data specified in the data source. The SQL SDK offers a system to take care of this in the form of triggers. In the 'demo' folder, you will see two scripts called 'CreateTriggers.sql' and 'DropTriggers.sql', the names of which are self explanatory. If you open up the first script, 'CreateTriggers.sql', the lines which you will need to amend are the following.

```
USE matchIT_API_demo

ALTER DATABASE matchIT_API_demo
```

Simply amend these lines to point to the database that contains the tables you are creating the triggers on.

```
'C:\Program Files\matchIT SDK for SQL Server\demo\StoredProcedures.dll'
```

Amend this file path if necessary to point to the assembly you wish to create the stored procedures from.

```
'C:\Program Files\matchIT SDK for SQL Server\demo\Config.xml'
```

Amend this file path to point to the configuration file that contains the data source defining the tables you wish to create the triggers on.

Note – Once created, the triggers rely on the Configuration file being unchanged and staying in the same location as when created. Triggers are only really recommended for long term set ups that are not going to change. The triggers created on the tables in data source specified when calling msp_CreateTableTriggers also rely on the fact that the database in question has the msp_GenerateSingleKeys stored procedure on it too (which gets created along with the assembly in CreateTriggers.sql). If this procedure is not available on the database, the triggers will simply do nothing. Triggers should be deployed with care and testing to make sure no errors occur that could prevent database updates, inserts and deletions occurring in the tables that the triggers exist on.

As an overview then, and as can be seen in the following line

```
EXEC msp_CreateTableTriggers @xmlConfigurationFilePath='C:\Program
Files\matchIT SDK for SQL Server\demo\Config.xml', @dataSourceID='1'
```

Triggers are created on a particular database using the specified configuration file on the tables defined in the specified data source. Once created, the triggers, when fired, rely on the Configuration File they were created with being in the location that was specified on creation, and for accurate results, for it to be unchanged.

To remove triggers, and all associated procedures and the assembly, make the same style modifications as per the create script above – The configuration file and data source used to remove the triggers will need to be the same as when they were created.

Running in Passive Mode

It may be necessary, depending on the scenario, to run some match processing in the back ground that is not required in real time. An example of this might be an e-commerce web site that has members joining every week – where it may be necessary to run a weekly process to gather all the new joiners and then search for matches in the database. In this situation, the best thing to do would be to gather a collection of records for which you wish to wish to check against the database for duplicates, and run the msp_SingleRecordMatch stored procedure (described in the technical reference) for each one and log the matches for each record as suits the situation. See the description of msp_SingleRecordMatch for information on what it returns and the parameters it accepts.

Technical Reference

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

StoredProcedures.dll

This is a SQL Server Database project, written in C# (.NET), that implements a number of stored procedures used to operate on SQL Server databases.

A number of core classes have been taken from the matchIT SDK and, in some cases, have undergone significant modification – for example, the BatchProcessor class now buffers record pairs then passes the buffer to the APIInterface for comparison (see below for further information). ADO.NET now completely replaces all use of ADO, because the latter cannot be used to obtain a fast *context connection* (a SQL Server in-process connection to the database).

Classes

Configuration

This class represents a configuration object that gets used in most of the stored procedures. The properties of the configuration object are set when the object parses the XML configuration file in the Load method. Any errors with the file cause the configuration object to throw an exception.

ConfigurationObjects

This file contains some definitions of objects that are specific to, and are used and stored in, the configuration object.

ConfigurationFileReader

This class contains some static methods for parsing the XML Configuration file. The methods catch and throw meaningful exceptions to help the user locate errors in the configuration file easily.

Record

This contains a collection of field values that represent a row in the database, usually either a contact or a company.

Database

This class represents a connection to a database.

When generating keys or finding matches in one table (or a set of joined tables), a context connection will be opened.

When finding records that overlap two distinct tables (or sets of joined tables), unfortunately one of the two tables must be opened with a non-context connection using a standard ADO.NET connection string. For increased performance, the larger database should be opened using the context connection.

FieldMappings

This class contains the definition of a FieldMapping object, a list of which gets stored in the configuration per data source, and a method for getting a list of FieldMapping objects from a configuration field mappings definition.

KeyFields

This class contains a static list of KeyField objects that are used during key table creation and key generation.

Keys

This file contains definitions for key and composite key objects that are created dynamically during various stored procedures using the configuration match key definitions and the field mappings stored in the configuration for the data source(s) in question.

Table

This class simply represents a table in a database. Tables can be joined, but this requires that each row of a table be uniquely referenced; rows are then joined based on a reference that's unique to that row (the sample database illustrates this).

Note that for maximum performance we recommend that only one table should be used, containing names, addresses, and matching and key.

Query

This class constructs a SQL SELECT command and executes it on the specified table (with any joined tables). Records can be easily read in sequence from the query's result.

Thread

Provides simplified creation of a worker thread, used only within the MultiThreadedBatchProcessor class.

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.

ExactPurge

This is the abstract (non-creatable) base class for the PurgeExactMatches and PurgeExactOverlap classes. It provides functionality for locating *exact* matches (to a very high likelihood) and logging these in a new table, 'exact_matches'. Optionally (disabled by default) exact matches can be purged from the database (i.e. for each set of matching records, all records – except the first – are deleted from the table).

PurgeExactMatches

Warning: Use caution with this class as records will be deleted.

Implements an algorithm for finding *exactly* matching records within a single table (or set of joined tables). 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

Warning: Use caution with this class as records will be deleted.

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).

BatchProcessor

This is the abstract (non-creatable) base class for the OneFileDedupe and TwoFileDedupe classes. It provides *buffering* functionality, whereby record pairs are buffered; when full, the buffer is passed to the APIInterface for batched comparison of all record pairs, utilising the COM matchIT API in a highly efficient process.

OneFileDedupe

This implements an algorithm for finding duplicate record pairs within a single table (or set of joined tables). By default, three standard keys are used (these are fully customisable). For each key, a query is performed on all records in the table in key order. Then, records with the same key value are clustered and pairs of records in the cluster are compared.

TwoFileDedupe

This implements an algorithm for finding matching records that overlap two distinct tables (or sets of joined tables). This can be useful, for example, when merging two tables into one – duplicates can be removed before merging – or to locate records from a suppression list.

MultiThreadedBatchProcessor

This is the abstract (non-creatable) base class for the MultiThreadedOneFileDedupe and MultiThreadedTwoFileDedupe classes. It can be used to divide the database into a number of *blocks* (for example, the first digit of the Zip, or the first letter of the region), with one or more threads being used to process these blocks using round-robin scheduling (i.e. the next available thread processes the next queued block, until all blocks have been processed).

Note, however, that only one thread may use the fast context connection at any one time. Thus, all worker threads – other than the main thread that spawns them – must use a slower non-context connection with Mars enabled (MultipleActiveResultSets=True, ADO.NET defaults this to False when database connections are made), as is done in the standard matchIT SDK. Unfortunately this is a limitation of SQL CLR.

It is recommended that multithreaded batch processing only be performed on machines with two or more CPUs and/or cores; otherwise performance could be impacted.

MultiThreadedOneFileDedupe

A multithreaded version of the OneFileDedupe class.

MultiThreadedTwoFileDedupe

A multithreaded version of the TwoFileDedupe class.

GroupProcessor

This is the abstract (non-creatable) base class for the GroupMatches and GroupOverlap classes.

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 identified 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.

OutputMatches

Represents an object that outputs data that is used by various other objects such as the BatchProcessor and MultiThreadedBatchProcessor.

Utils

Contains methods for getting a temporary file path, and deleting any temporary files.

Stored Procedures

The SDK implements the following stored procedures.

(Note that all table names are hard-coded and so the stored procedures are specific to a database. A convention could be followed whereby all databases contained in SQL Server follow the same table- and field-naming scheme. However, if this is not possible then new stored procedures could be created for deduplication of other named tables, or a parameter could be added to the stored procedure so that the table name is specified elsewhere. The list of field mappings in the Database class could also be expanded to include more synonyms of relevant field names.)

msp_CreateKeysTable

Simply creates a new table in the current database that contains all necessary matching and key fields.

msp_AddKeyFieldsToTable

Simply appends all necessary matching and key fields to the specified table in the current database. This is the recommended approach for maximum matching performance. -

msp_GenerateKeys

Generates the matching and key field values for all records in the current database.

msp_BulkGenerateKeys

As msp_GenerateKeys, except that the matching and key field data is written to a temporary file that is then bulk-inserted into an empty keys table. Significantly reduces the time taken for key generation, particularly when used with large databases.

msp_CreateCustomMatchesTable

Creates a custom schema of a matches table from the standard matches table as defined in the configuration object for the specified data source. Relies on a matches table existing in the specified data source. The table name defined in the configuration can be given a suffix as the third parameter of the procedure if creating multiple instances to distinguish between them.

msp_CreateCustomGroupedMatchesTable

Creates a custom schema of a grouped matches table from the standard matches_grouped table as defined in the configuration object for the specified data source. Relies on a matches_grouped table existing in the specified data source. The table name defined in the configuration can be given a suffix as the third parameter of the procedure if creating multiple instances to distinguish between them.

msp_CreateTableTriggers

Creates dynamic key update triggers on each table defined in the specified data source in the specified configuration file.

msp_DeleteTableTriggers

Deletes dynamic key update triggers on each table defined in the specified data source in the specified configuration file.

msp_CreateUniqueRefField

Creates a primary key column with a particular name on the named table in the data source in the specified configuration.

msp_GenerateSingleKeys

This method is called by the key update triggers on tables defined in the data source. It updates, creates or deletes the keys entry(s) for the specified record in the specified data source.

msp_PurgeExactMatches

Finds all exactly matching record pairs in the current database.

msp_PurgeExactOverlap

Finds all records that exactly overlap the current and specified databases.

msp_FindMatches

Finds all matching record pairs in the current database.

msp_FindOverlap

Finds all records that overlap the current and specified databases.

msp_MultiThreadedFindMatches

Finds all matching record pairs in the current database using multiple threads, each thread operating on a subset of the database.

msp_MultiThreadedFindOverlap

Finds all records that overlap the current and specified databases using multiple threads, each thread operating on a subset of the database.

msp_GroupMatches

After running msp_FindMatches, this will group all matching record pairs into *sets* of matching records.

msp_GroupOverlap

If performed after msp_FindOverlap then the results are similar to running msp_GroupMatches, except that grouping overlap matches is much simpler because the overlap results are already partially grouped into sets.

msp_SingleRecordMatch

This procedure finds all records that match a generated record made up from the parameters supplied as arguments for the procedure (Namely full name, company, address1, town, region, postcode), in the specified data source in the specified configuration.

msp_UnloadAPIInterface

This procedure unloads the API interface instance from the SQL server process. Note that to reload the API Interface (i.e. to use a procedure that makes use of the API interface) after calling this procedure, all stored procedures and the StoredProcedures assembly should be deleted and recreated to avoid any exceptions (as per the process in the script UnloadAPIInterface.sql). A typical error that might occur if this process is not followed is described in the troubleshooting section.

APIInterface.dll

Written in unmanaged C++ to implement batch generation and comparison. This is unfortunately necessary because the performance of COM Interop is significantly reduced when used in SQL CLR projects.

To counter this, records are batched by the C# code then passed to the interface DLL for batch processing by the matchIT API (either key generation or record comparison), instead of directly using the API to perform multiple generations or comparisons which significantly reduce performance.

Exported Functions

APIInterface.dll exports a number of functions used by the StoredProcedures.dll assembly.

APIInterface_Intialise

Creates all required matchIT API COM objects and performs various matchIT API initialisation, including setting matching weights and constraints.

APIInterface_SetApiSettings

Configures the settings of the matchIT API. Called after initialise

APIInterface_SetMinimumScores

Sets the minimum score thresholds for each scoring level. Called after initialise.

APIInterface_Generate

Uses the matchIT API to *generate* all buffered records passed from the StoredProcedures.dll assembly. Generate, in this context, means to both clean and standardise fields of the record (name, company, address, etc.), and create matching and key fields. Generated records are buffered and passed back to the assembly.

APIInterface_Compare

Uses the matchIT API to compare all buffered record pairs passed from the StoredProcedures.dll assembly. Each duplicate record pair (in the form of two unique references and a matching score) are buffered and passed back to the assembly.

APIInterface_GetErrorLog

If errors occur in the matchIT API – for example, when calling Engine.Generate() or Engine.Compare() – then these will be logged to a temporary file and this function will return its pathname. If a blank string is returned then no errors were detected.

APIInterface_Release

Releases all COM objects, terminating use of the matchIT API.

Troubleshooting

Error Message:

```
Msg 6522, Level 16, State 1, Procedure msp_GenerateKeys, Line 0
A .NET Framework error occurred during execution of user-defined routine or aggregate "msp_GenerateKeys":
System.DllNotFoundException: Unable to load DLL 'C:\Program Files\matchIT SDK\matchIT SDK for SQL Server\source\APIInterface\Release\APIInterface.dll': This application has failed to start because the application configuration is incorrect. Reinstalling the application may fix this problem. (Exception from HRESULT: 0x800736B1)
```

Possible Causes:

APIInterface.dll does not exist at the location specified on the SQL Server machine.

Ensure a *release* build of the solution (*both* projects) has been compiled.

If deploying across machines, ensure the two DLLs have been copied to the correct locations on the SQL Server machine.

The Visual C/C++ 8.0 runtime libraries do not exist on the SQL Server machine.

Execute this program on the SQL Server machine to correctly install these library DLLs:

"C:\Program Files\Microsoft Visual Studio 8\SDK\v2.0\BootStrapper\...

...Packages\vcredist_x86\vcredist_x86.exe"

Either copy the installer to the machine or execute it via a share to the development machine.

Error Message:

```
Msg 6522, Level 16, State 1, Procedure msp_GenerateKeys, Line 0
A .NET Framework error occurred during execution of user-defined routine or aggregate "msp_GenerateKeys":
System.Data.DBConcurrencyException: Concurrency violation: the UpdateCommand affected 0 of the expected 1 records.
```

Possible Cause:

The 'genkeys' table exists and is not empty, and msp_GenerateKeys previously failed.

Subsequently running msp_GenerateKeys means that, because the table isn't empty, SQL Update commands are used to write data to the key fields rather than an Insert command (if the table was empty). But the exception is thrown when trying to update a row that doesn't exist.

To get around this, delete the 'genkeys' table then re-run the key generation. Also, consider using msp_BulkGenerateKeys, as that stored procedure is much less likely to leave the table in an incomplete state.

Error Message:

```
Msg 6532, Level 16, State 49, Procedure msp_GenerateKeys, Line 0
.NET Framework execution was aborted by escalation policy because of out of memory.
System.Threading.ThreadAbortException: Thread was being aborted.
```

Possible Cause:

There is insufficient memory available to SQL CLR, so a memory allocation request failed.

There are a number of places in the code where this could occur, but this can most likely be prevented by reducing the sizes of the input and output buffers used to send and receive data to and from the APIInterface.

In the BatchProcessor, Generator, and BulkGenerator classes, simply reduce the RecordBufferSize and OutputBufferSize constants. You'll have to determine the optimum sizes, but try halving them until the error is eliminated. Note, however, that smaller buffers could impact record generation and/or comparison performance.

In the Generator and BulkGenerator classes the output buffer should be larger than the input buffer (as records will 'grow' with their new keys and matching fields etc.), a factor of four to one is recommended for safety.

Error Message:

```
Msg 6522, Level 16, State 1, Procedure msp_BulkGenerateKeys, Line 0
A .NET Framework error occurred during execution of user defined routine or aggregate 'msp_BulkGenerateKeys':
System.AccessViolationException: Attempted to read or write protected memory. This is often an indication that other memory is corrupt.
System.AccessViolationException:
   at toolkit.DllImports.APIInterface_GetErrorLog(Int32 Engine, StringBuilder Pathname, Int32 PathnameSize)
   at StoredProcedures.APIInterface.Release(Configuration configuration)
   at StoredProcedures.msp_BulkGenerateKeys(String xmlConfigurationFilePath, String dataSourceID)
```

Possible Cause:

The procedure msp_UnloadAPIInterface was called previously.

If the procedure msp_UnloadAPIInterface is called, then the stored procedures and assembly on the database should all be deleted and recreated. Failure to do so will most likely end up causing the above error when executing a stored procedure that makes use of the API Interface (such as msp_BulkGenerateKeys).