Center for Advanced Research in Entity Resolution and Information Quality (ERIQ)

Introduction to Entity Resolution with OYSTER v3.3

Document Version: 1.10, Date: 02 December 2012

Edited by:

Fumiko Kobayashi

Authors:

Fumiko Kobayashi and John Talburt

Revision History

Version	Date	Prepared By	Position	Reason for Update
1.1	10/27/2010	Fumiko Kobayashi	RA	• Added Appendixes • Updated Document and Figures to reflect OYSTER v2.1
1.2	12/01/2010	Fumiko Kobayashi	RA	• Updated Figure 2 with new data flow • Updated text and figures to reflect OYSTER v2.4.1
1.3	12/15/2010	Fumiko Kobayashi	RA	• Corrected erroneous and outdated scripts • Added section to discuss new startup operation that doesn't expect Z-drive • Modified cover page layout • Modified Revision History layout
1.4	2/4/2011	Fumiko Kobayashi	RA	Modified document to reflect new changes to Oyster V2.6
1.5	3/2/2011	Fumiko Kobayashi	RA	Modified document to reflect new changes to Oyster V3.0
1.6	4/30/2011	Fumiko Kobayashi	RA	Modified document to reflect new changes to Oyster V3.1
1.7	6/14/2011	Fumiko Kobayashi	RA	• Added Java memory argument explanation • Updated new changes made to 3.1
1.8	6/22/2011	Fumiko Kobayashi	RA	Removed references to RunScriptName
1.9	10/26/2011 04/15/2012	Fumiko Kobayashi	RA	Update with changes made to 3.2
1.10	8/24/2012 12/02/2012	Fumiko Kobayashi	RA	Update with changes made to 3.3

Table of Contents

Introduction

In this document you will learn about the basic concepts of Entity Resolution (ER). This document touches on the five activates of ER, the four architectures of ER, and the four techniques used to determine equivalence in ER. Once a foundation is established you will learn what the OYSTER system is and how it can be used to perform ER. This document also briefly explains Entity Identity Information Management (EIIM) and how OYSTER provides functionality not found in any other ER system to integrate EIIM. This document details not only what OYSTER was designed to do but also the files that are required by OYSTER and how to configure the files to perform different types of ER.

This document is written in the hopes that once you finish reading it, you will have a sufficient understanding of both ER and OYSTER to allow you to effectively use the OYSTER system to help resolve your ER needs. The various sections of this document were designed to not only guide you through the use of OYSTER but also to provide you a reference that can be used when you need help configuring your OYSTER runs.

Entity Resolution

Introduction to Entity Resolution (ER)

Entity Resolution is the process by which a dataset is processed and records are identified in the dataset that represent the same real-world entity. Each record in the dataset is defined by a group of attributes. These attributes vary depending on the type of data; some examples of attributes would be FirstName, LastName, and SocialSecurityNumber if referring to person data or SerialNumber, Description, and ModelNumber if referring to product data.

Five Activities of Entity Resolution

The process of entity resolution encompasses a broad set of activities. These activities are defined to be the following five major activities:

ERA1 – Entity Reference Extraction - Locating and collecting entity references from unstructured information.

- Is only necessary when an entity reference source is presented as unstructured information.
- Interest in ERA1 has grown along with the realization that a great deal of an organization's useful information often resides in unstructured formats.

ERA2 – Entity Reference Preparation – The application of profiling, standardization, data cleansing, and other data quality techniques to structured entity references prior to the start of the resolution process.

- Extensive pre-processing of entity reference sources is necessary before effective resolution process can take place.
- Also known as the ETL (Extract, Translate, Load) process

ERA3 – Entity Reference Resolution - Resolving (deciding) whether two references are to the same or different entities.

- Often done by applying a process called matching**.**
    - A process by which the decision is based on the degree of similarity between the values of the identity attributes in the two references
- ER systems generally use four basic techniques for determining that references are equivalent and should be linked.

- Direct Matching
- Transitive Equivalence
- Relationship Resolution
- Asserted Equivalence

ERA4 - Entity Identity Management (EIM) - Building and maintaining a persistent record of
entity identity information over time.

- ER systems that support identity management have a significant advantage because they have the potential to:
  - Maintain persistent link values
  - Allow transactional processing
  - Go beyond direct matching and link records by relationships and assertions
- ER system that always assign the same reference the same link value are said to provide persistent link values.
  - ER systems that support persistent links will assign the same link value to a reference each time it is presented.
- ER systems that build entity identities as they process references are called identity capture systems
- Entity Identity Information Management (EIIM) is a fairly new concept in ER. It is a component of entity identity management (EIM) and is a set of rules and utilities that can be used to maintain and update entity identity information based on asserted knowledge and identity monitoring.

ERA5- Entity Relationship Analysis - Exploring the network of relationships among different, but related entities.

- Exploring entity relationships is at the intersection of entity resolution and data mining.

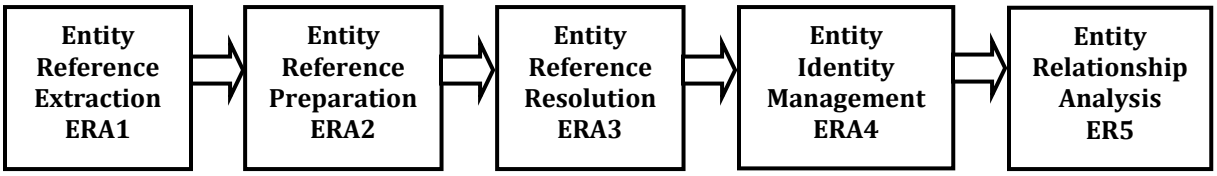


Figure 1: Five activities of Entity Resolution

These activities, as illustrated in Figure 1, define the whole of what ER is (Big ER). When most people refer to ER they are referring to only a subset of these activities, generally ERA2 and ERA3 (Little ER). Not every ER process involves all five activities, and different ER tools and systems are designed to handle different parts of the overall ER process.

Four Architectures of ER

- **Merge-purge**
  - Entity references are systematically compared to each other and separated into clusters (subsets) of equivalent records
  - Most common form of ER
  - Also known as record linkage

- **Heterogeneous database join**
  - A transactional ER system where attribute values from an input reference is translated into queries to different databases and database tables. The query results are analyzed to determine if there are references in databases that are equivalent to the input reference.

- **Identity resolution**
  - All incoming references are resolved against a predefined set of managed identities.
  - Each identity in an identity resolution system has a fixed identifier that can be used to link references that are equivalent to the identity, thus creating a persistent link

- **Identity capture**
  - A form of identity resolution in which the system builds (learns) a set of identities from the references it processes rather than starting with a known set of identities.

Four Techniques for Determining Equivalence

- **Direct Matching** – determining equivalence between two references based on the degree of similarity between the values of corresponding attributes. The five basic methods for determining the similarity of attribute values are:
  - Exact match
  - Numerical difference
  - Approximate syntactic match
  - Approximate semantic match
  - Derived match codes

- **Transitive Equivalence** – determining equivalence through the use of an intermediary or common record. This means that if record A = record B, and record B = record C, then through transitive equivalence record A= record C. This is the simplest break down of transitive equivalence; there can be any number of intermediary records that form a link between other records.

- **Relationship Resolution** – discovering links by exploring patterns of relationships among references that don't rise to the level of equivalence. This is often done through the use of techniques borrowed from graph theory and network analysis. Relationship

resolution allows multiple relationships to be considered and multiple equivalence decisions to be made at the same time. This is a growing method when approaching the issue of resolving co-authors.

- **Asserted Equivalence** – the instantiation of a link between two references based on a prior knowledge that they are equivalent. An asserted equivalence often takes the form of one record carrying the attribute values of two non-matching references. This means a single identity may contain two first names and two last names that do not match but based on prior knowledge, such as the person changed their name, both first and last names refer to the same real-world entity.

Principles and Law of ER

There are seven principles of entity resolution as defined by Dr. John Talburt. These principles are as follows:

ER Principle #1: Information systems store and manipulate references to entities, not the entities.

ER Principle #2: ER is fundamentally about linking equivalent references, not record matching.

ER Principle #3: ER false negatives are generally a more difficult problem to detect and solve than are false positives.

ER Principle #4: ER processes are generally designed avoid false positives at the expense of creating false negatives.

ER Principle #5: Entity Resolution is not the same as Identity Resolution.

ER Principle #6: ER systems that provide persistent link values must also implement some form of identity management.

ER Principle #7: ER systems link equivalent references through inferred and asserted linking. Inferred links can be created through direct matching, transitive linking, or relationship resolution.

The following is the fundamental law of Entity Resolution.

Fundamental Law of ER: Two entity references should be linked if and only if they are equivalent (reference the same real-world entity).

OYSTER

Introduction to OYSTER

OYSTER is an open-source software development project sponsored by the ERIQ Research Center at the University of Arkansas at Little Rock (ualr.edu/eriq). OYSTER (Open sYSTem Entity Resolution) is an entity resolution system that supports probabilistic direct matching, transitive linking, and asserted linking. To facilitate prospecting for match candidates (blocking), the system builds and maintains an in-memory index of attribute values to identities. Because OYSTER has an identity management system, it also supports persistent identity identifiers. OYSTER is unique among other ER systems in that it is built to incorporate Entity Identity Information Management (EIIM). OYSTER supports EIIM by providing methods that force identifiers to be unique among identities, maintain persistent IDs over the life of an identity, and by allowing the ability to fix false-positive and false-negative resolutions, which cannot be done with matching rules, through the use of assertion, traceability, and other features.

OYSTER is written in Java and the source code and documentation are available as a free download on file page of the OYSTERER SourceForge website (http://sourceforge.net/projects/oysterer/). OYSTER is free for use under the OYSTER open-source license and the GNU General Public License version 2.0 (GPLv2).

Although the original version of OYSTER was developed to support entity resolution (ER) for student records in longitudinal studies, the system design readily accommodates a broad range of ER domains and entity types. A key feature of the system is that all entity and reference-specific information is interpreted at run-time through user-defined XML scripts. This allows OYSTER to be configured as a merge-purge, identity capture, identity resolution, identity update, or assertion system.

The OYSTER Project has been guided by several design principles

- OYSTER does not use an internal database for its operation.
- System inputs and outputs can either be text files or database tables.
- XML scripts are used to define
    - All entity identity attributes
    - The layout of each reference source
    - Identity rules for resolving each reference source

Entity Identity Information Management (EIIM)

As mentioned in the previous section, OYSTER is built to incorporate EIIM. It does this by providing facilities to do multiple types of assertions (some developed specifically for and

unique to the OYSTER project) and traceability to track the lifetime of a record. These features are discussed in the following sections.

Assertion

In entity resolution, assertion is the ability to introduce knowledge into an Entity Identity Structure (EIS) to force resolutions or to force reference to never resolve. Match Rules can only resolve references based on the algorithms and information available. There are four (4) types of assertion runs that are made available in the OYSTER System. These are as follow:

- Reference to Reference
- Reference to Structure
- Structure to Structure
- Structure Split

Each of these are in the following sections.

Reference to Reference Assertion

Reference to Reference Assertions (RefToRef) is the most well know and used form of asserted linking. RefToRef Assertion forces multiple references to be matched and resolved into the same cluster. It is performed on references that would not match on any defined matching rule but are known to represent the same entity. An example of this could be a person who has changed addresses and also changed their name. If the user has this knowledge they can use OYSTER to force the entity reference for the original name and address to match the new name and address forcing them to resolve to the same identity. OYSTER builds and maintains an Entity Identity Structure (EIS), in the form of an xml formatted idty file. Since the identity created by this assertion run contains both the old and new name and address any further reference for the person, no matter if it is old or new, will resolve to this identity.

Reference to Structure Assertion

Reference to Structure Assertion (RefToStr) is a type of assertion created for the OYSTER system that forces multiple references to be consolidated with an existing identity structure found in the OYSTER idty file. RefToStr Assertions are used to inject references into identity structures based on knowledge about the reference. For example, a magazine company has a centralized EIS for customer and potential customer information. A subscriber changes their name address but do not want to stop receiving their magazine so they contact the magazine company to inform them of the change. The magazine company can use RefToStr Assertion to inject this new information about the subscriber into an existing identity in their knowledge base. Without RefToStr Assertion the new information for the subscriber would not have matched the identity structure using standard matching rules.

Structure to Structure Assertion

Structure to Structure Assertion (StrToStr) is a type of assertion created for the OYSTER system that forces multiple identity structures found in an existing EIS to be consolidated into a single identity structure. This is used to fix false negative matches that were produced by the OYSTER match rules. Through the use of StrToStr Assertions multiple identity structures that are later found to actually match can be forced to consolidate. These consolidations are based on previous knowledge of the references in the identity structures.

Structure Split Assertion

Structure Split Assertion (SplitStr) is a type of assertion created for the OYSTER system that forces a single identity structure found in an existing EIS to be divided into two (2) or more identity structures. This is used to fix false positive matches that were produced by the OYSTER match rules. Through the use of SpltiStr Assertion an identity structure can be forced to split and negative assertion rules are put into place in the EIS that will never allow these newly split identity structures to be merged in the future. These splits are based on previous knowledge of the references in the identity structure.

Traceability

Another aspect of OYSTER that allows for EIIM is its ability to trace the progress of a reference from the current run all the way back to its origin. This allows the ability to examine the origin and the evolution of a reference throughout its lifetime to figure out where a bad match was made and fix it with the use of one of the four assertions. This ability is known in OYSTER as Traceability and is a new feature that is introduced in version of OYSTER v3.2. Traceability is localized into the idty file that is generated by each run if the Trace value is set to "On" in the RunScript. Detailed explanations of the idty file and all of its elements can be found in the OYSTER Reference Guide.

Change Report

The last feature provided by OYSTER for EIIM is the Change Report. This feature allows for users to easily monitor changes that were made during the current OYSTER run. The change report is a consolidated list of all new merges, updates, and new identities that were done during the run. A new change report is produced with every run of OYSTER and detailed explanations of the change report can be found in the OYSTER Reference Guide.

User Define Index (UDI)

OYSTER v 3.3 introduced an exciting new feature that allows a user to define multiple customized indices. As mentioned earlier, OYSTER uses indices to build the candidate lists used when matching is performed.

In the original design of OYSTER, the attribute values of incoming references were inserted into an inverted index as a way to find the most probably match candidates for newly input references. However, it was found that this method of building the inverted index has three major drawbacks.

1. It is predicated on the idea that each identity rule will have at least one exact match term. System performance tends to degrade dramatically when rules have one or more inexact match terms (LED, Scan, etc.)
2. The index key is always a value from a single attribute. To find candidates for multi-term identity rules the system has to perform multiple lookups, one for each rule term with an exact match before reducing the candidate set to a manageable size.
3. The index logic is fixed. Therefore to gain maximum performance, users must tailor identity rules in a way that best fits the logic of the index scheme rather than using an index logic that best fits the identity rules being used.

The new user-defined index scheme will allow the user to direct OYSTER to create a single index value that represents multiple terms (attributes) in a rule. It will also allow the user to define more than one index which allows the user to customize an index for specific rules. Each index defined by the user will be a single value formed by concatenating a series of "hash values." Each hash value is created by applying a pre-defined transformation to the value of an attribute. Many of the hash algorithms may be the same as or directly related to a particular similarity function (Soundex, Scan, etc.)

This ability to build custom indices provides drastic improvements in runtimes. The tradeoff is that the user must have an intimate knowledge of the rules and data to build optimized indexes. When indexes are designed correctly, the OYSTER runtime can decrease from hours and possible days to a matter of minutes. It is important to note that with the introduction of UDI into the OYSTER system, the default inverted index has been removed and if not UDI is defined, OYSTER will do brute force comparisons (compare every record with every other record).

The syntax of UDIs is defined in detail in the Oyster v3.3 Reference Guide.

Cross-Attribute Comparison (CAC)

A second new feature that is introduced in OYSTER v3.3 is the Cross-Attribute Comparison (CAC). The purpose of the cross-attribute comparison is to allow users to create identity rules that compare different attributes between two references, such as comparing first name to last name or social security number to student number. In previous version of OYSTER a rule only allows the user to compare the values of the same attribute between two references.

The syntax of the CAC is defined in detail in the Oyster v3.3 Reference Guide.

Basic OYSTER Run Steps

An OYSTER run comprises the following steps

1. Start Run
    a. Read Run Script
    b. Load identities into memory from stored XML documents
    c. While there are reference sources to process
        i. Read reference source
        ii. Resolve each reference to an identity
        iii. Update identities (if in capture mode)
        iv. Update link index
    d. Write updated identities to storage as XML documents
    e. Write link index file
    f. Write change report
    g. Write merge map
2. End Run

The dataflow diagram in Figure 2 illustrates the above steps.

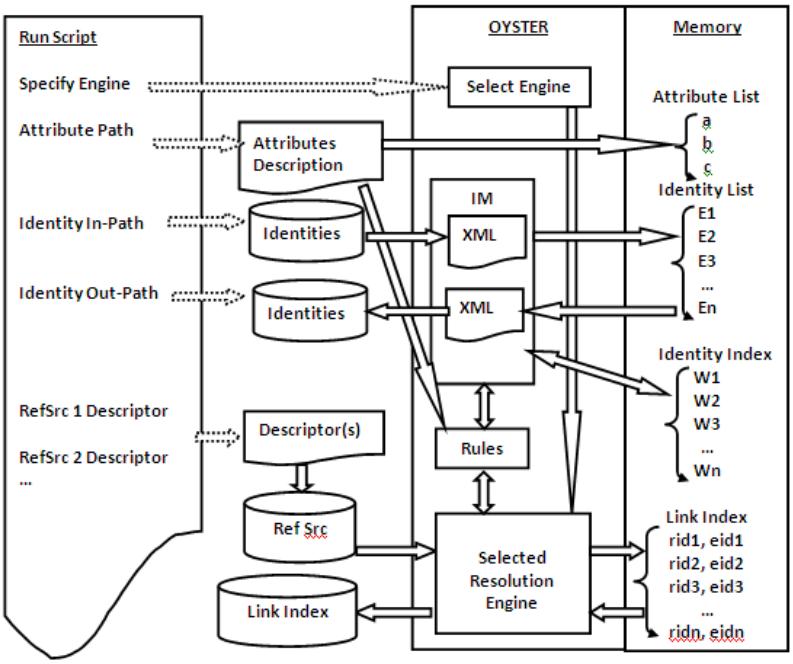


Figure 2: Basic OYSTER Run Steps Dataflow

Files and Structure

OYSTER requires multiple files to function and creates additional files when it runs:

- Input Files:
  - OysterRunScript.xml
  - OysterSourceDescriptor.xml
  - OysterAttributes.xml
  - Input Source files (if using a text file for the input source)
  - Input idty file (if performing Identity Resolution or Identity Update)
- Output Files:
  - Name.idty (Optional)
  - Name.link (Required)
  - Name.idty.emap (if Explanation="[On/Off]" and Debug="On")
  - Name.idty.indx (if Explanation="[On/Off]" and Debug="On")
  - Log file (always created, location and name can be defined in Run Script)
  - Identity Change Report.txt
  - Identity Merge Map.csv

When building these files there is no predefined location where to save them. It is recommended a Run folder be created for each new OYSTER run you attempt. This will allow you to easily organize and keep track of your used and generated files. Inside the Run folder create an Input, Scripts, and Output folder and place the files in their corresponding folder. The suggested structure should look similar to this:

Z:\Oyster\Run001\
\Input*(Place input source files here)*
\Output*(output should be generated here [specified in RunScript])*
\Scripts*(Place all scripts here)*

Figure 3 illustrates the above folders. By using this structure you can perform unlimited runs of OYSTER while staying organized. Please note drive Z: is not required by OYSTER, the above suggested folder structure could be stored in any location.

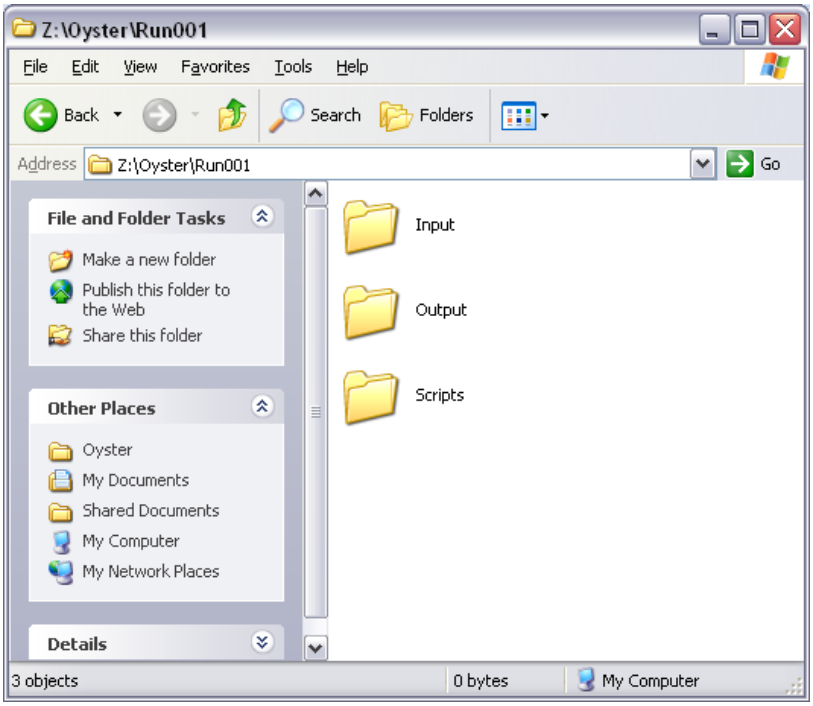


Figure 3: Run001 folder

There are a few other files and folders that are necessary when running certain features of OYSTER. Inside the OYSTER folder there must be a folder named 'data' that contains a file named 'alias.dat' (shown in Figure 4). This file contains a tab delimited file with two columns of data (illustrated in Figure 5). The first column contains a name and the second column contains an acceptable alias for that name. This is the lookup file that is used when "Nickname" is specified in the match rules.

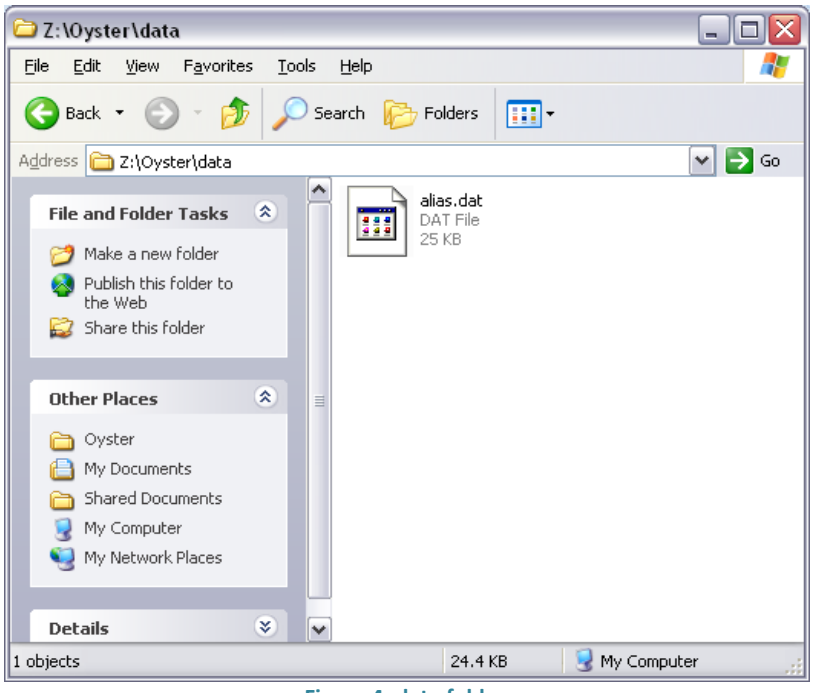


Figure 4: data folder

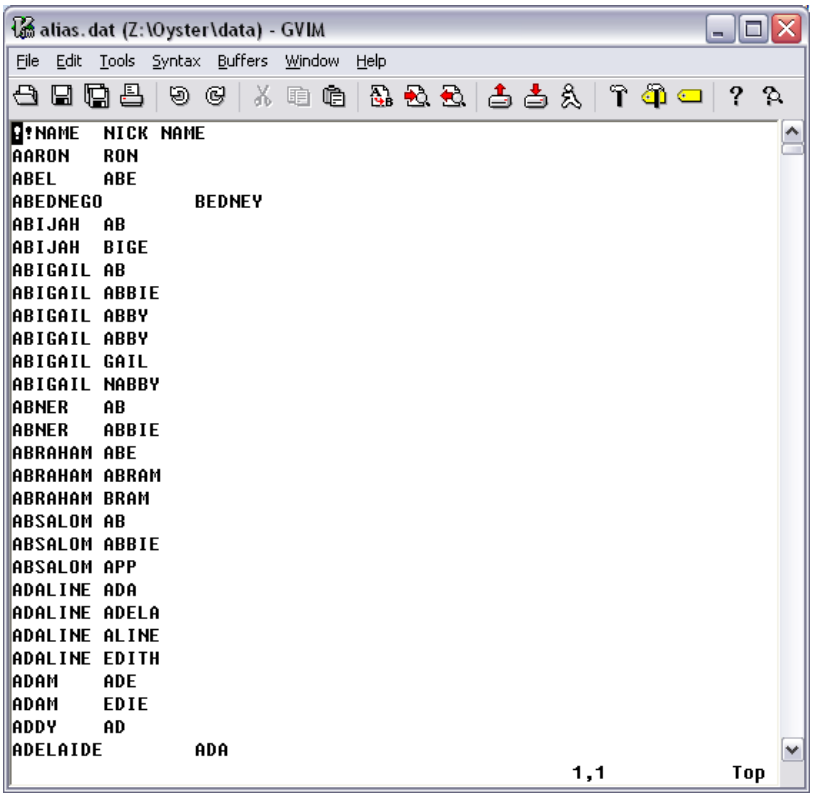


Figure 5: alias.dat file

The other required folder must be named 'lib' (shown in Figure 6) and must contain the following files:

- mysql-connector-java-5.0.8-bin.jar
- ojdbc14.jar
- sqljdbc4.jar
- sqljdbc.jar
- swing-layout-1.0.jar

These files are used by OYSTER to make connections and handle the data when sources other than text files are used.

Please note that the data and lib folder **MUST** be stored in the same folder in which the oyster.jar file is located. When the .jar file is run, it searches the directory in which it is run from for these two folders. If it fails to locate the folders then OYSTER will fail to make a connection to any data source other than a text file and will be unable to perform any matching on "Nickname".

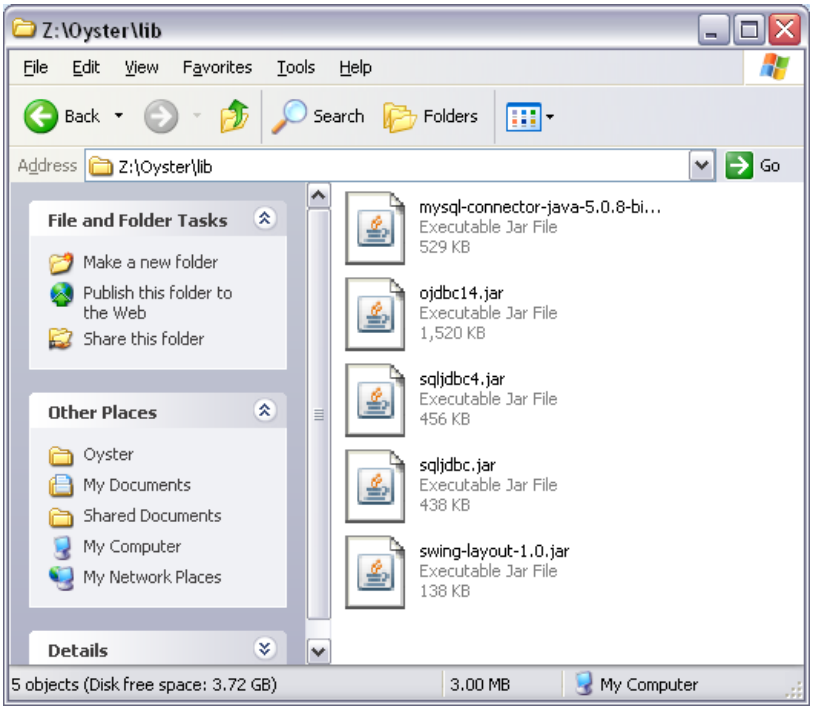


Figure 6: lib folder

The main OYSTER folder is depicted in Figure 7.

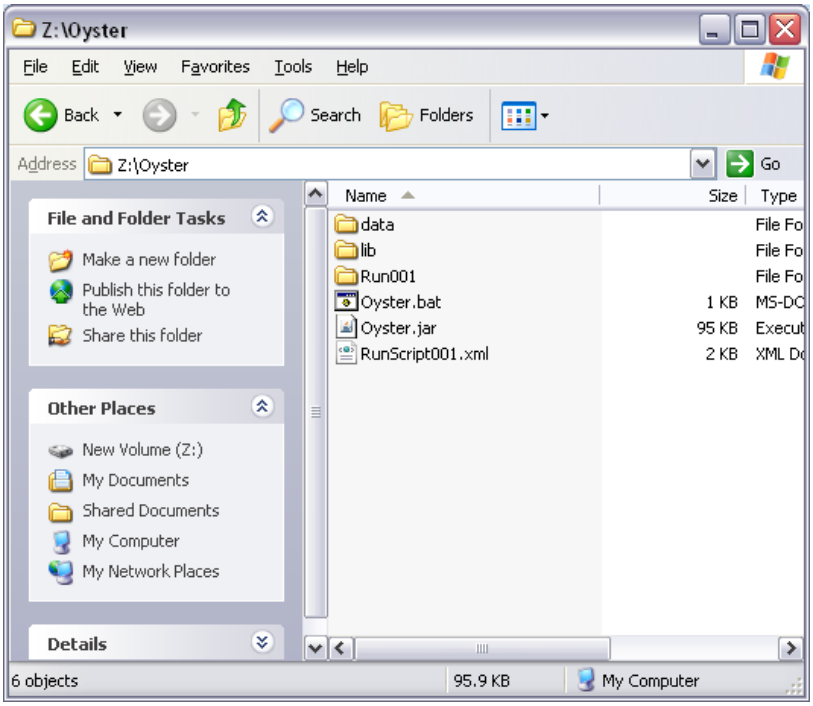


Figure 7: OYSTER folder

The above structure is purely a suggestion to allow for organized OYSTER testing and runs. It may also be beneficial to create a folder in which to store the generated log files

Launching OYSTER

There are various methods that can be utilized to launch OYSTER. The first method to run OYSTER is via the command prompt. The following steps will launch OYSTER.

NOTE: OYSTER requires Java version 1.5 or newer to run or it will error out.

1. Open a Commnad Prompt
    a. Select Start->Run
    b. Type *cmd*, press Enter.
2. From here there are two ways to launch OYSTER.
    a. Option 1: Navigate to the folder that contains the Oyster.jar file.
        i. Use the cd command to navigate, type *cd* followed by the folder path in which the .jar file can be found.
        Example: *cd d:\example\Oyster*
        ii. Once the command prompt is pointed to the correct folder, OYSTER can be launched by typing the following command:
        java –jar Oyster.jar
    b. Option 2: Use the absolute path to the Oyster.jar file in the java command.
        i. Through the use of the absolute path the Oyster.jar file can be launched no matter which folder the command prompt is pointed to.
        ii. An example command that uses the absolute path is:
        java –jar D:\Example\Oyster\Oyster.jar

The second method is to create a batch file that contains the command needed to call and run the Oyster.jar file. By creating a batch file it creates an additional file that must be stored in the same folder as OYSTER (for portability) but removes the necessity to know how to navigate to the folder containing the oyster.jar file as was required by the first method. The following steps will create a batch file that can be used to launch an OYSTER run.

1. Open Notepad
    a. Select Start->Run
    b. Type *notepad*, press Enter
2. Type *java -jar Oyster.jar*
    a. Please note that the absolute path can be used for the Oyster.jar file if the .jar file will always be stored in the same location but you want the ability to move the batch file to a more accessible location such as on the Desktop. The command with the absolute path would look like: *java –jar D:\Folder\Path\Oyster.jar*
3. Save the file as Oyster.bat in the same folder that the Oyster.jar file is located unless the absolute path was used as described above.
4. To launch OYSTER simply double click on the Oyster.bat file.
   Note: The path and file name in the batch file must always be edited to match the path and file name of the Oyster.jar file whenever they are changed.

Invoking the OYSTER Run Script

Once OYSTER is launched a Run Script can be invoked to perform the desired run. Figure 8 shows the prompt provided by OYSTER at the beginning of each run.

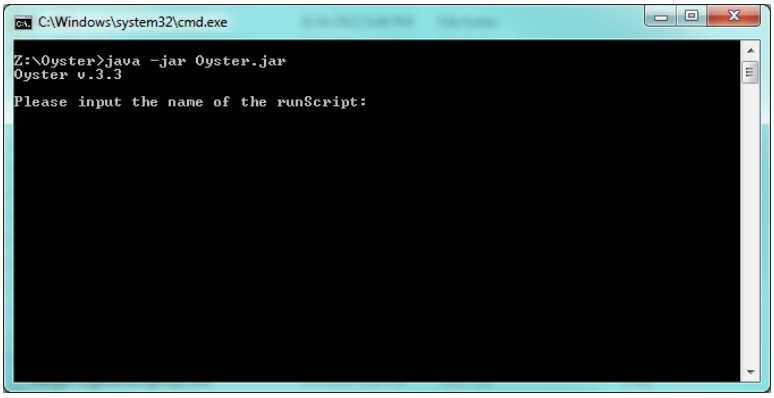


Figure 8: OYSTER Prompt

OYSTER expects to receive the name of the run script as input from the user. The specified run script may or may not require the user to specify the absolute path along with the name depending on the location from which the Oyster.bat file is launched. The following three scenarios all illustrate how an OYSTER run script can be invoked.

1. OYSTER is being run from the same folder as the run script using Absolute Path
    a. When prompted by the OYSTER run with "Please input the name of the runScript:", type *D:\Oyster\Run1\RunScript.xml*
    b. You can see this example uses the absolute path "D:\Oyster\Run1\" in addition to the name of the run script. In this scenario the absolute path is not actually required, due to both the run script and the .jar file being located in the same folder, but will not cause any error with the run.
2. OYSTER is being run from the same folder as the run script using Local Path
    a. When prompted by the OYSTER run with "Please input the name of the runScript:", type *RunScript.xml*
    b. You can see this example that since the Oyster.bat file is stored in and ran from the same folder in which the run script is stored, it does not require the absolute path, only the name of the run script.
3. OYSTER is being run from a different folder than the run script
    a. When prompted by the OYSTER run with "Please input the name of the runScript:", type *D:\Oyster\Run1\RunScript.xml*
    b. You can see this example uses the absolute path "D:\Oyster\Run1\" in addition to the name of the run script. In this scenario, the absolute path must be included since the Oyster.bat file is located in a different folder than the run script.

Once the folder structure is complete and OYSTER can be launched properly, the OYSTER run can be set up.

Run OYSTER with Extra Memory

The OYSTER system was designed to perform all of its processing in Random Access Memory (RAM). This decision was made for various reasons but mainly to provide a system that can process large quantities of records very quickly. When disk I/O is removed and the system relies strictly on RAM for its work space, the processing can be sped up significantly.

Since OYSTER is written in Java, there are limitations on the amount of memory that is allocated by default. The default memory allocation for Java is 64 megabytes. This is enough space to process smaller amounts of records but you will eventually run out of memory as you process more and more records into a Knowledge Base.

Java offers 2 parameters that allow the user to control the amount of memory (called a heap in Java) to be allocated.

- `-Xms`
  - This argument allows a user to define the initial size of them memory to be allocated.
  - Defaults to 2MB
- `-Xmx`
  - This argument allows a user to define the maximum memory that can be used during the Java run.
    - **NOTE**: on Windows based 32-bit computers, the maximum memory allowed to be allocated by Java is 2048MB (2GB). This limitation does not exist in 64-bit systems or in UNIX.
  - Defaults to 64MB

  NOTE: When setting the Java memory size, the memory argument should be specified using one of the letters '`m`' or '`M`' for MB, or '`g`' or '`G`' for GB. 'MB' or 'GB' will not work.

These parameters are added to the *`java -jar Oyster.jar`* command ran from the command prompt or stored in a run script.

A finished command that allocates 64MB of memory for the initial run with a maximum of 2GB of memory would be: *`java -Xms64M -Xmx2G -jar Oyster.jar`*

OYSTER XML Files

OYSTER functions are directed by a series of customizable XML documents. These documents tell OYSTER everything from where the input source files are located, to what attributes are located in the input source, to what rules should be applied in order to perform ER. OYSTER requires a minimum of three XML documents in order to run correctly. These three documents are:

- OysterRunScript
- OysterSourceDescriptor
- OysterAttributes

OYSTER always requires one OysterRunScript and one OysterAttributes file for every run and always at least one OysterSourceDescriptor file. There must be one OysterSourceDescriptor for each input source to be used during the OYSTER run.

OysterRunScript

The OysterRunScript is the main file for any OYSTER run. It is the first file requested by OYSTER when the run is initiated. It is used to specify the following:

- The RunScriptName
    - This attribute value must match the file name without the extension.
- If Explanation and Debug are turned on or off.
    - These options restrict the amount of information written to the log files.
- If extra detail is required in the Change Report.
- If Trace is enabled.
- If memory statistics are displayed at various stages in the run.
- The location, size, name, and number of log files that can be created.
- The RunMode for the run
- The resolution engine to be used during the run.
    - OYSTER offers three resolution engines (Described in detail in the Reference guide):
        - RSwooshStandard
            - Uses Attribute-Based Matching
        - RSwooshEnhanced
            - Uses Attribute-Based Matching
        - FSCluster
            - Uses Record-Based Matching
- The path to the XML OysterAttributes document defining the entity attributes to be used during the run
- The path to the XML Identity documents, if any, to be loaded at the start of the run

    - This is an identity file generated by a previous OYSTER run.
- The path to a location where the updated XML Identity documents will be stored at the end of the run if any are specified for the run
- A list of paths to the OysterSourceDescriptor XML document(s)
    - One per source file to be included in the run.
- The path to a location where the Link Index will be written at the end of the run
- A path to each source input file and assertion input file

A simple OysterRunScript is illustrated in Figure 9.

```
<?xml version="1.0" encoding="UTF-8"?>

<OysterRunScript>
    <Settings RunScriptName="TestRun" Explanation="Off" Debug="Off" SS="No"
ChangeReportDetail="No" Trace="On" />

    <LogFile Num="5"
Size="100000000">Z:\Oyster\Log\MergePurge_%g.log</LogFile>

    <RunMode>IdentityCapture</RunMode>

    <EREngine Type="RSwooshEnhanced" />

    <AttributePath>Z:\Oyster\OysterAttributes.xml</AttributePath>

    <!-- Identity Input Selection -->
    <IdentityInput Type="None"/>

    <!-- Identity Output Selection (Only needed when Capture="Yes") -->
    <IdentityOutput Type="TextFile">Z:\Oyster\Test.idty</IdentityOutput>

    <!-- Link Output Selection (Always Required) -->
    <LinkOutput Type="TextFile">Z:\Oyster\Test.link</LinkOutput>

    <!-- Sources to Run -->
    <ReferenceSources>
      <Source>Z:\Oyster\OysterSourceDescriptor1.xml</Source>
    </ReferenceSources>
</OysterRunScript>
```

Figure 9: Sample OysterRunScript

The OysterRunScript in Figure 9 makes use of only text files for the Output, OYSTER also has the ability to read and write to a database. This RunScript is for an IdentityCapture run, different tags are used when performing an assertions run. A full reference of the OysterRunScript file can be found in the OYSTER Reference Guide. Both types of matching, Record-Based and Attribute-Based, are also defined in detail in the OYSTER Reference Guide.

OysterSourceDescriptor

The OysterSourceDescriptor contains information about the input source file. It is used to specify the location of the input source, and the attributes in the source that make up a record. The OysterSourceDescriptor is different from the OysterAttributes and OysterRunScript files in that there can be multiple OysterSourceDescriptors in a single OYSTER run. This is due to OYSTER requiring an OysterSourceDescriptor file be created for each input source that will be used during the OYSTER run, if there are five input source files then there should be five corresponding OysterSourceDescriptors for those files. Each OysterSourceDescriptor should contain a unique value in the "Name" field or OYSTER will not function correctly.

A sample OysterSourceDescriptor is illustrated in Figure 10.

```
<?xml version="1.0" encoding="UTF-8"?>

<OysterSourceDescriptor Name="source1">
    <!-- Types of Sources (Only one can be defined) -->
    <!-- Delimited -->
    <Source Type="FileDelim" Char="|" Qual="" Labels="Y">Z:\Oyster\
Test_Data_1.txt</Source>

    <!-- Items in Source (One for each item in the source including
reference identifier -->
    <ReferenceItems>
        <!-- For Delimited -->
        <Item Name="IdentityID" Attribute="@RefID" Pos="0"/>
        <Item Name="Fname" Attribute="StudentFirstName" Pos="1"/>
        <Item Name="Lname" Attribute="StudentLastName" Pos="2"/>
        <Item Name="SSN" Attribute="SocialSecurityNbr" Pos="3"/>
        <Item Name="DOBYMD" Attribute="StudentDateOfBirth" Pos="4"/>
    </ReferenceItems>
</OysterSourceDescriptor>
```

Figure 10: Sample OysterSourceDescriptor

The OysterSourceDescriptor in Figure 10 specifies a delimited text file as the input source with the file being Delimited by "|" and the file having labels. All of this is specified in one line of the OysterSourceDescriptor as shown in Figure 11.

```
    <Source Type="FileDelim" Char="|" Qual="" Labels="Y">Z:\Oyster\
    Test_Data_1.txt</Source>
```

Figure 11: Source File Definition in OysterSourceDescritpor

The last section of the file, illustrated in Figure 12, identifies the Attributes in the input source and the position that each one is located at in the file.

```
<ReferenceItems>
    <!-- For Delimited -->
    <Item Name="IdentityID" Attribute="@RefID" Pos="0"/>
    <Item Name="Fname" Attribute="StudentFirstName" Pos="1"/>
    <Item Name="Lname" Attribute="StudentLastName" Pos="2"/>
    <Item Name="SSN" Attribute="SocialSecurityNbr" Pos="3"/>
    <Item Name="DOBYMD" Attribute="StudentDateOfBirth" Pos="4"/>
</ReferenceItems>
```

Figure 12: Attribute identifiers in OysterSourceDescriptor

A full reference of the OysterSourceDescriptor file can be found in the OYSTER Reference Guide.

OYSTER Reserved Words

During the design of the OYSTER system, it was decided that there is a need for a set of reserved words that can be used to specify special attribute types in the OYSTER Source Descriptor. In Figure 12, The @RefID keyword is used as the Attribute value for the item named IdentityID. This specific keyword informs OYSTER that this attribute is the unique identifier for the references in the source input file. OYSTER will exclude this value when creating OYSTER IDs but will use it to allow for future file tracking along with the source name specified.

A complete list of OYSTER reserved words, and their uses, can be located in the OYSTER Reference Guide.

OysterAttributes

The OysterAttributes file is used to identify each attribute that is present in the input source and the rules to be used by OYSTER to perform ER on the input source records. Each XML <Attribute> element in the document defines one OYSTER attribute labeled by the value of the XML "Item" attribute. These labels are used in the Source Descriptor to identify the logical items in a Reference Source that are OYSTER identity attributes. In addition to the attribute name, the value of the XML attribute "Algo" specifies a pre-defined matching algorithm that is to be used for comparing attributes value of this type. Specifying an algorithm is optional, and if it not given or a given name is not found, then a default matching algorithm is used.

A sample OysterAttributes file is illustrated in Figure 13.

```
<?xml version="1.0" encoding="UTF-8"?>

<OysterAttributes System="School">
    <Attribute Item="StudentFirstName" Algo="None"/>
    <Attribute Item="StudentLastName" Algo="None"/>
    <Attribute Item="SocialSecurityNbr" Algo="None"/>
    <Attribute Item="StudentDateOfBirth" Algo="None"/>

    <!-- -->
    <IdentityRules>
        <Rule Ident="1">
            <Term Item="Fname" MatchResult="Exact"/>
            <Term Item="Lname" MatchResult="Exact"/>
            <Term Item="SSN" MatchResult="Exact"/>
        </Rule>
        <Rule Ident="2">
            <Term Item="Fname" MatchResult="Initial"/>
            <Term Item="Lname" MatchResult="Exact"/>
            <Term Item="SSN" MatchResult="Exact"/>
        </Rule>
    </IdentityRules>
</OysterAttributes>
```

Figure 13: Sample OysterAttributes File

In this example the default matching algorithm will be used since Algo is specified as "None". This provides the users the ability to match using the following comparators:

- True/False
  - EXACT
  - EXACT_IGNORE_CASE
  - TRANSPOSE
  - INITIAL
  - NICKNAME
  - SOUNDEX
  - DMSOUNDEX

    - IBMALPHACODE
    - MATCHRATING
    - NYSIIS
    - CAVERPHONE
    - METAPHONE
- Functionalized
    - LED - default is 0.8 if LED match is used, signature for user defined threshold is LED(threshold)
    - QTR - default is 0.25 if QTR match is used, signature for user defined threshold is QTR(threshold)
    - SUBSTRLEFT(length)
    - SUBSTRRIGHT(length)
    - SUBSTRMID(start, length)
    - Scan(Direction, CharType, Length, Casing, Order)
    - SmithWaterman(Match, Mismatch, Gap, Threshold)

All the above comparators are described in detail in the "Oyster v3.3 Reference Guide".

The last section of the file, illustrated in Figure 14, specifies the identity rules that are defined by the user to be used to perform ER on the records in the input source.

```
<IdentityRules>
    <Rule Ident="1">
        <Term Item="Fname" MatchResult="Exact"/>
        <Term Item="Lname" MatchResult="Exact"/>
        <Term Item="SSN" MatchResult="Exact"/>
    </Rule>
    <Rule Ident="2">
        <Term Item="Fname" MatchResult="Initial"/>
        <Term Item="Lname" MatchResult="Exact"/>
        <Term Item="SSN" MatchResult="Exact"/>
    </Rule>
</IdentityRules>
```

Figure 14: Identity Rules defined in OysterSourceDescriptor

In this script 'MatchResult="Exact"' and 'MatchResult="Initial"' are being used. By default OYSTER can use the match codes listed above but this can be extended by the user by extending the base class OysterComparator.java as a new class with a name starting with "OysterCompare" and implementing the method String: getMatchCode(String, String).

A full reference of the OysterAttributes file can be found in the OYSTER Reference Guide.

As of OYSTER v3.3 the OysterAttributes file also allows for user Defined Indexes (UDI) to be defined for the run and also allows users to specify Cross-Attribute Comparison (CAC) rules. Both of these new functionalities are define din detail in the OYSTER Reference Guide.

Example Scenario

Now that we have covered the basics of how OYSTER runs and each of the XML files required for it to run successfully, consider the following data:

```
ID|FirstName|LastName|DateofBirth
10|MICHAEL|DANIELS|11/16/1938
58|HECTOR|SCHOONOVER|10/21/1966
6|CLINTON|REYES|2/20/1910
43|GRACIELA|HOANG|6/4/1913
37|MICHAEL|AMPEREZ|1/24/1987
11||DANIELS|11/16/1938
57|HECTOR|SCHOONOVER|10/21/1966
16|CLINT|REYES|2/20/1910
42|GRACIELA|HOANG|6/4/1913
```

Assume that a company has been having problems on an employee report that is generated monthly. The report has some employees listed multiple times which is skewing the numbers and causing erroneous data to be presented to upper management. The manager who receives this report reviewed the data and has noticed that the duplicate records match one of the following rules:

1. The first name is missing, the last names are the same, and the dates of birth are the same.
2. The first name was entered incorrectly but the last names are the same and the date of birth are the same.
3. The first names are the same, the last names are the same, the dates of birth are the same but the employee was supplied a new ID.

The manager has tasked you with cleaning up and consolidating the employee table based on his three matching rules.

With OYSTER

To perform this task using OYSTER, the first step should be to analyze the data set and create an OysterSourceDescriptor file. This file will assign logical names to each of the attributes in the data set.

This dataset contains four attributes: ID, FirstName, LastName, DateofBirth. These four attributes should be translated into the ReferenceItems section of the OysterSourceDescriptor file. Note that when using a delimited text file as the input source the “Name” and “Attribute” values defined for each item do not need to match the labels

given in the files. The “Pos” however has to match the relative position of the attribute in the input source.

The ReferenceItems sections would be defined as illustrated in Figure 15.

```
<ReferenceItems>
    <!-- For Delimited -->
    <Item Name="IdentityID" Attribute="@RefID" Pos="0"/>
    <Item Name="Fname" Attribute="StudentFirstName" Pos="1"/>
    <Item Name="Lname" Attribute="StudentLastName" Pos="2"/>
    <Item Name="DOBYMD" Attribute="StudentDateOfBirth" Pos="3"/>
</ReferenceItems>
```

Figure 15: Attributes Defined in OysterSourceDescriptor

The last step in building the OysterSourceDescriptor is specifying the type of file and where it will be located. Since this dataset is delimited by the character “|”, the entry would look like the Source in Figure 16 assuming the file has been saved as Z:\Oyster\Input\Data_1.txt.

```
<Source Type="FileDelim" Char="|" Qual=""
Labels="Y">Z:\Oyster\Input\Data_1.txt</Source>
```

Figure 16: Source File Definition

Those are all of the steps required in building the OysterSourceDescriptor. You should make a note of where the file is saved as this location must be specified later in the OysterRunScript. The completed OysterSourceDescriptor file is illustrated in Figure 17. Save this file in the Scripts folder discussed earlier in the Files and Structure section of this document.

```
<?xml version="1.0" encoding="UTF-8"?>

<OysterSourceDescriptor Name="source1”>
    <!-- Types of Sources (Only one can be defined) -->
    <!-- Delimited -->
    <Source Type="FileDelim" Char="|" Qual=""
Labels="Y">Z:\Oyster\Input\Data_1.txt</Source>

    <!-- Items in Source (One for each item in the source including
reference identifier -->
    <ReferenceItems>
        <!-- For Delimited -->
        <Item Name="IdentityID" Attribute="@RefID" Pos="0"/>
        <Item Name="Fname" Attribute="StudentFirstName" Pos="1"/>
        <Item Name="Lname" Attribute="StudentLastName" Pos="2"/>
        <Item Name="DOBYMD" Attribute="StudentDateOfBirth" Pos="3"/>
    </ReferenceItems>
</OysterSourceDescriptor>
```

Figure 17: Complete OysterSourceDescriptor File

Now that the OysterSourceDescriptor file has been created the OysterAttributes file must 000be created to define which algorithms should be used on each attribute during matching and what matching rules will be used during ER. Note that the attributes Item value in the OysterAttributes file must match the value assigned to Attribute in the OysterSourceDescriptor file. Since only Missing and Exact are required when doing matching for this dataset, the default OYSTER matching algorithm can be used. The defined attributes are shown in Figure 18. The Algo attribute can be left off since the default algorithm is to be used by OYSTER.

```
<Attribute Item="StudentFirstName" />
<Attribute Item="StudentLastName" />
<Attribute Item="StudentDateOfBirth" />
```

Figure 18: Defined Attributes in the OysterAttribute.xml File

The next step is to review the matching rules provided and translate them into Rules that can be used by OYSTER. Rule one states that the first name is missing, the last names are the same, and the dates of birth are the same. As discussed earlier, by default OYSTER can handle Exact and Missing matching (along with others). That means this rule could be translated into the OYSTER rule illustrated in Figure 19.

```
<Rule Ident="1">
    <Term Item="StudentFirstName" MatchResult="Missing"/>
    <Term Item="StudentLastName" MatchResult="Exact"/>
    <Term Item="StudentDateOfBirth" MatchResult="Exact"/>
</Rule>
```

Figure 19: Identity Rule 1

The next rule states that the first name was entered incorrectly but the last names are the same and the date of birth are the same. This rule can be translated into the OYSTER rule illustrated in Figure 20.

```
<Rule Ident="2">
    <Term Item="StudentLastName" MatchResult="Exact"/>
    <Term Item="StudentDateOfBirth" MatchResult="Exact"/>
</Rule>
```

Figure 20: Identity Rule 2

The last rule states that the first names are the same, the last names are the same, the dates of birth are the same but the employee was supplied a new ID. This rule can be translated into the OYSTER rule illustrated in Figure 21.

```
<Rule Ident="3">
    <Term Item="StudentFirstName" MatchResult="Exact"/>
    <Term Item="StudentLastName" MatchResult="Exact"/>
    <Term Item="StudentDateOfBirth" MatchResult="Exact"/>
</Rule>
```

Figure 21: Identity Rule 3

The complete OysterAttributes file is illustrated in Figure 22.

```
<?xml version="1.0" encoding="UTF-8"?>

<OysterAttributes System="School">
    <Attribute Item="StudentFirstName" />
    <Attribute Item="StudentLastName" />
    <Attribute Item="StudentDateOfBirth" />
    <!-- -->
    <IdentityRules>
        <Rule Ident="1">
            <Term Item="StudentFirstName" MatchResult="Missing"/>
            <Term Item="StudentLastName" MatchResult="Exact"/>
            <Term Item="StudentDateOfBirth" MatchResult="Exact"/>
        </Rule>
        <Rule Ident="2">
            <Term Item="StudentLastName" MatchResult="Exact"/>
            <Term Item="DOBYMD" MatchResult="Exact"/>
        </Rule>
        <Rule Ident="3">
            <Term Item="StudentFirstName" MatchResult="Exact"/>
            <Term Item="StudentLastName" MatchResult="Exact"/>
            <Term Item="StudentDateOfBirth" MatchResult="Exact"/>
        </Rule>
    </IdentityRules>
</OysterAttributes>
```

Figure 22: Complete OysterAttributes File

Save this file in the Scripts folder discussed earlier in the Files and Structure section of this document. This location will be specified in the OysterRunScript.

Now that the OysterAttributes and OysterSourceDescriptor files are created, the OysterRunScript can be created. This file requires you specify the location of your OysterAttributes and OysterSourceDescriptor as well as the location of where you want the output files to be stored. You are can also set the level of logging to take place and the RunMode that will be used for this particular run. The complete OysterRunScript is illustrated inFigure 23. Save this file as “Z:\Oyster\OysterRunScript.xml”.

```
<?xml version="1.0" encoding="UTF-8"?>

<OysterRunScript>
    <Settings RunScriptName="OysterRunScript" Explanation="Off" Debug="Off"
ChangeReportDetail="No" Trace="On" />

    <LogFile Num="5" Size="100000000">Z:\Oyster\Log\Run_%g.log</LogFile>

    <RunMode>IdentityCapture</RunMode>

    <EREngine Type="FSCluster" />

    <AttributePath>Z:\Oyster\Scripts\OysterAttributes.xml</AttributePath>

    <!-- Identity Input Selection -->
   <IdentityInput Type="None"/>

    <!-- Identity Output Selection (Only needed when CaptureMode=On) -->
    <IdentityOutput
Type="TextFile">Z:\Oyster\Output\Test.idty</IdentityOutput>

    <!-- Link Output Selection (Always Required) -->
    <LinkOutput Type="TextFile">Z:\Oyster\Output\Test.link</LinkOutput>

    <!-- Sources to Run -->
    <ReferenceSources>
        <Source>Z:\Oyster\Scripts\OysterSourceDescriptor1.xml</Source>
    </ReferenceSources>
</OysterRunScript>
```

Figure 23: Compete OysterRunScript File

Now that each of the three required XML files have been created, you can launch OYSTER through either of the methods discussed in the Launching OYSTER section of this document. Once launched, you will be prompted for the name of the OysterRunScript as shown in Figure 24:

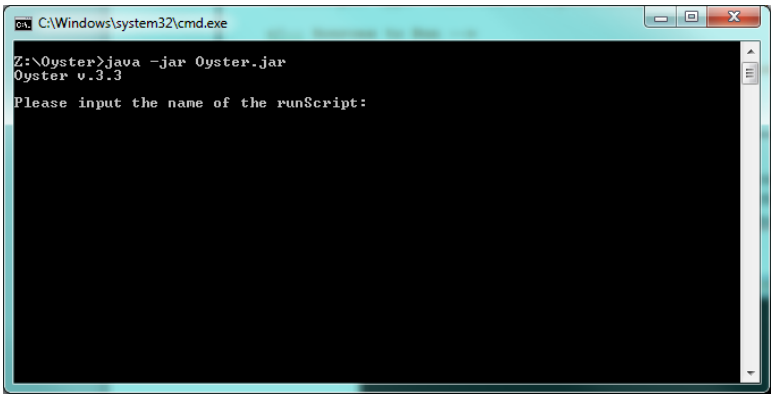


Figure 24: Command prompt opened by Oyster.bat file

The OYSTER version number is displayed when OYSTER is launched as depicted in Figure 24.

Once you specify the name of the OysterRunScript and press enter OYSTER will process your records and display the results on the screen, the results for this run are shown on Figure 25 and Figure 26.

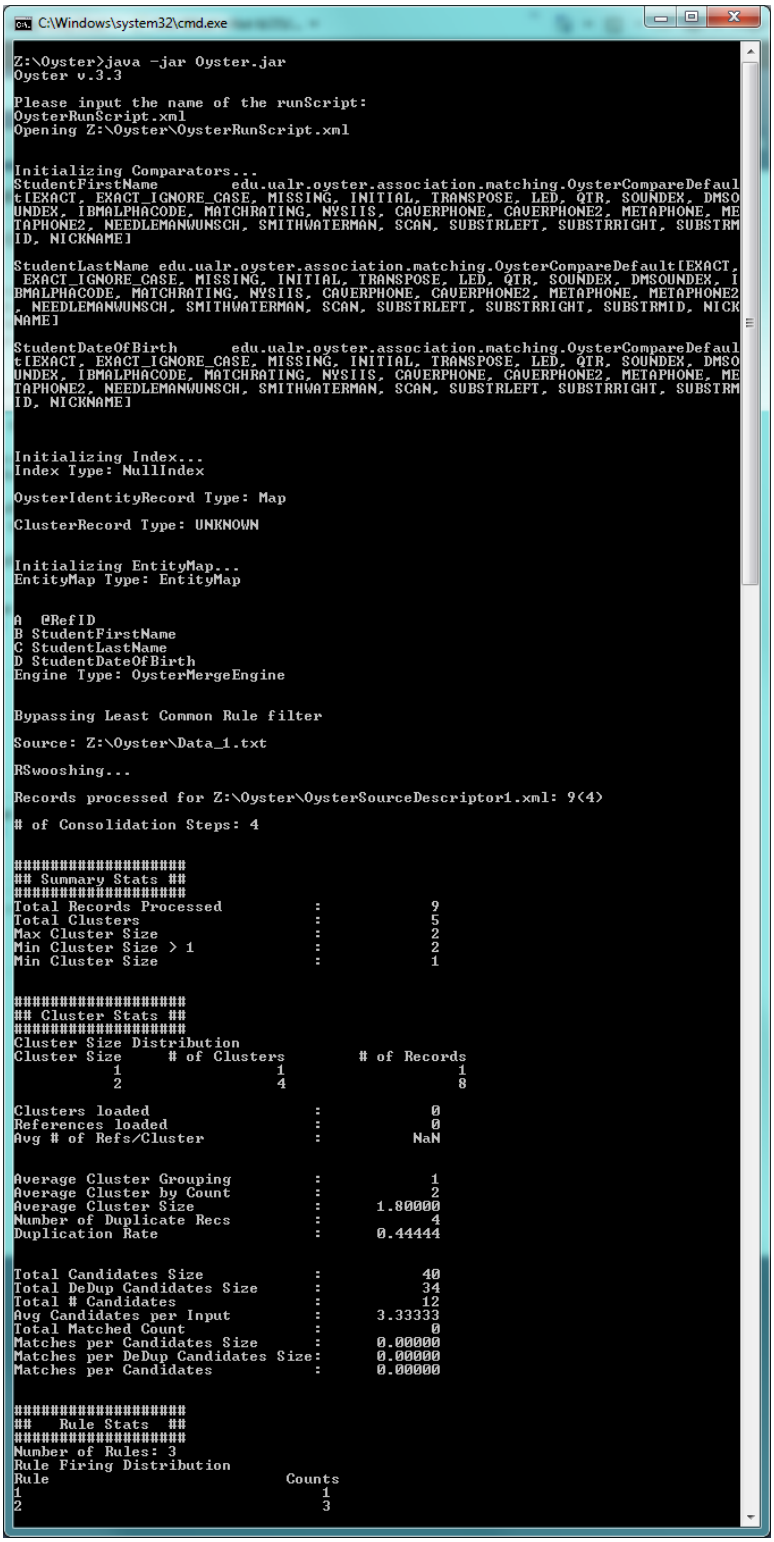


Figure 25: Output generated by OYSTER run as written to the Command Prompt - 1

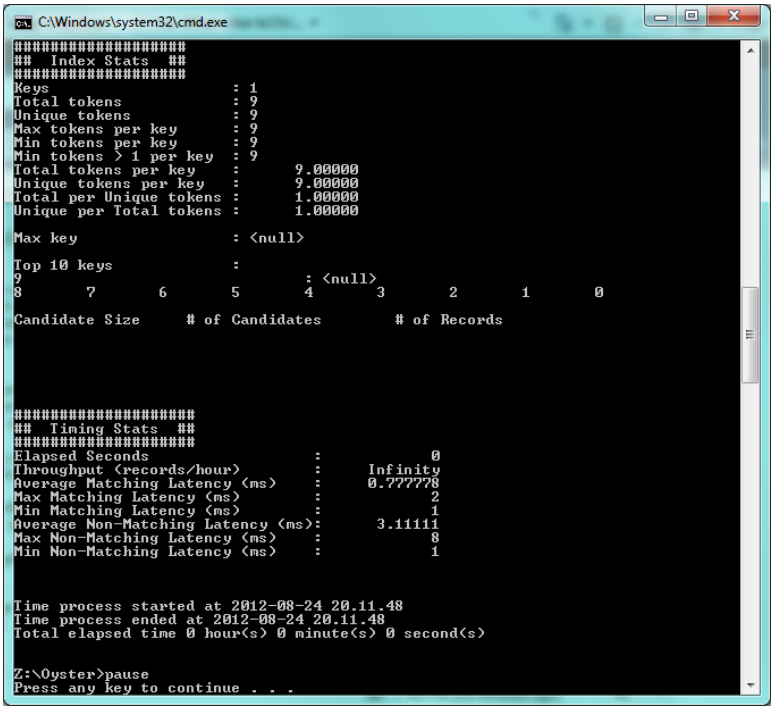


Figure 26: Output generated by OYSTER run as written to the Command Prompt - 2

From the results you can see that OYSTER was able to read nine records and resolve them to five entities (Clusters).

The OYSTER output is stored in XML format in the "Identity Change Report.txt", "Test.idty", and the "Test.link" file that was specified in the OysterRunScript. The contents of the files are shown in Figure 27, Figure 28, Figure 29 respectively.

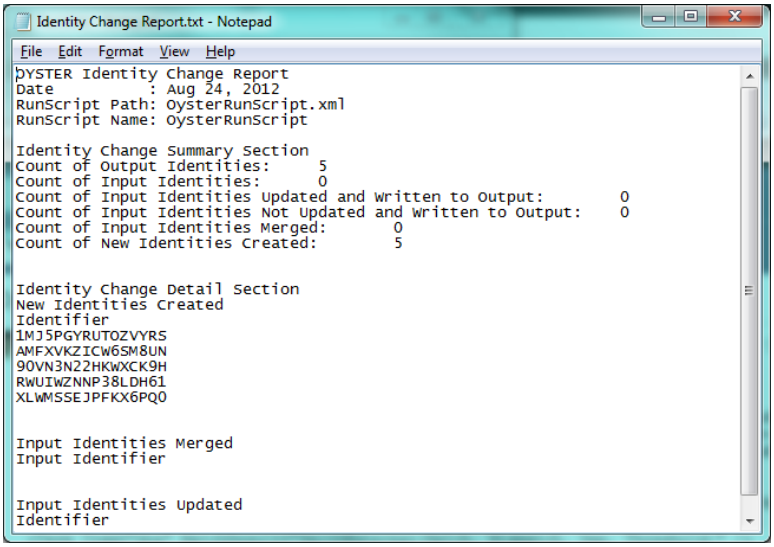


Figure 27: Change Report Generated by OYSTER Run

Figure 28: Identity File Generated by OYSTER Run

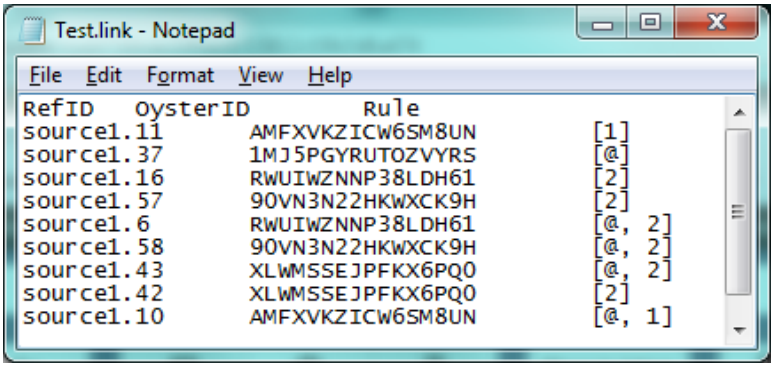


Figure 29: Link File Generated by OYSTER Run

What Is So Great About OYSTER

With the above scenario it may be hard to see what is good about using OYSTER. It may seem like creating the XML files is more work than it is worth to consolidate the nine records. Imagine, however, that these nine test records are an excerpt from a table containing thousands of records. Imagine again trying to process all these records by hand. This would be nearly impossible to accomplish without some degree of error and would take a ridicules amount of time. This is where OYSTER comes in. Through using the same XML scripts that we created for the nine records, OYSTER can perform the same Entity Resolution on the thousands of records by simply replacing the source file(s) with a file containing all of the records.

OYSTER Run Configurations

OYSTER can be configured to perform three of the four basic ER architectures and four types of asserted linking (discussed earlier). For OYSTER to run, a user must specify what "mode" it should run in. The mode is defined by the value defined in the XML <RunMode> tag of the source descriptor. This section will discuss the basic constraints and configuration requirements of setting up an OYSTER run and demonstrate how to use OYSTER for each of the following:

- Merge-purge
  - RunMode = "MergePurge"
- Identity capture
  - RunMode = "IdentityCapture"
- Identity build from assertions
  - Reference to Reference
    - RunMode = "AssertRefToRef"
  - Reference to Structure
    - RunMode = "AssertRefToStr"
  - Structure To Structure
    - RunMode = "AssertStrToStr"
  - Split Structure
    - RunMode = "AssertSplitStr"
- Identity resolution
  - RunMode = "IdentityResolution"
- Identity Update
  - RunMode = "IdentityUpdate"

Merge-purge

Figure 30 illustrates the dataflow for an OYSTER run configured to perform merge-purge.

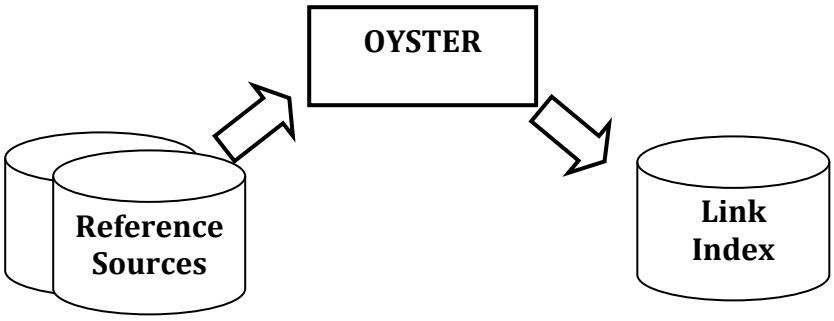


Figure 30: Merge-purge configuration

Configuration

When OYSTER is configured to perform merge-purge there should also be no input identities or output identities specified in the OysterRunScript, but a Link files must be specified as shown in Figure 31.

```
<!-- Identity Input Selection -->
<IdentityInput Type="None"/>

<!-- Identity Output Selection -->
<IdentityOutput Type=”None” />

<!-- Link Output Selection -->
<LinkOutput Type="TextFile">Z:\Oyster\Index.link</LinkOutput>
```

Figure 31: Defined Input and Output for Merge-purge

Lastly, the RunMode should be set to “MergePurge” as shown in Figure 32.

```
<RunMode>MergePurge</RunMode>
```

Figure 32: RunMode Set to “MergePurge” in OysterRunScript for Merge-purge

Example

This run will use the test data file named 'Merge-purgeTest.txt', illustrated in Figure 33. This data consists of six references composed by five attributes. The first attribute is the IdentityID, this is a unique identifier associated to each record. The other attributes consist of FirstName, LastName, SchoolCode, and DOB. When these attributes are combined as they are in the source file they are used to define a set of sample student references.

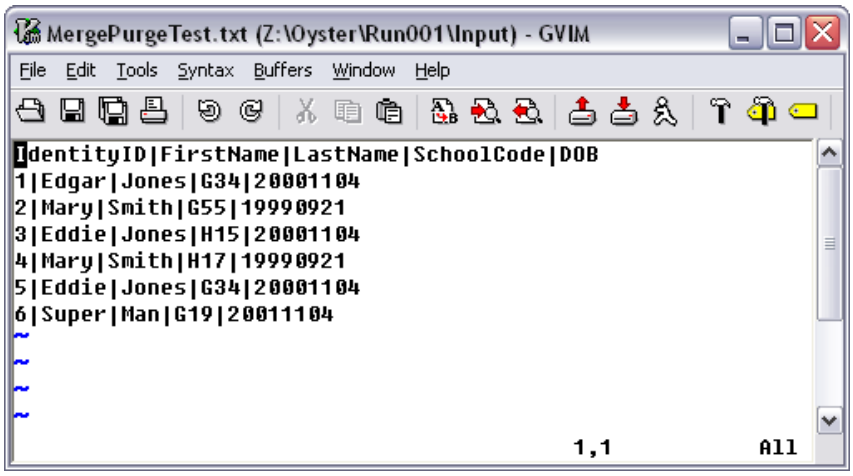


Figure 33: Source data for merge-purge run

After analyzing the source data the source descriptor file can be created. This file contains information including the location of the source data, the attributes to be used to define each reference in the source, and how to connect to the source. The contents of this file are illustrated in Figure 34.

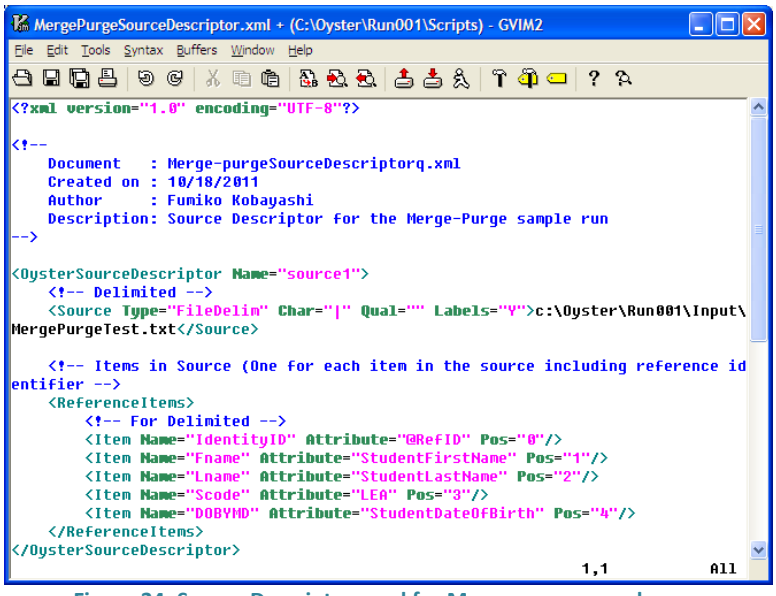


Figure 34: Source Descriptor used for Merge-purge sample run.

Once the source descriptor is defined the source attributes file must be defined. This file is stored in the Source folder along with the source descriptor file above. The attributes file is used to define the attributes in the source along with the algorithm used to compare the attributes and the matching (identity) rules used to perform ER. For this sample run two identity rules will be used. The first rule states that the reference will be considered equivalent if the FirstName, LastName, and DOB attributes match. The second rules states that the references are equivalent if the LastName, DOB, and SchoolCode match. The source attribute file is named 'MergePurgeAttributes.xml' and is shown in Figure 35.

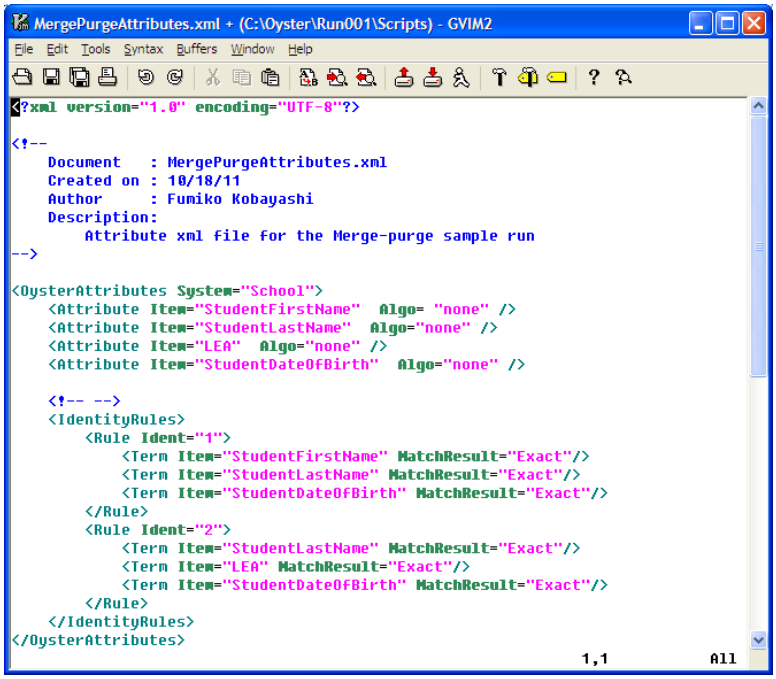


Figure 35: Attributes file used for the Merge-purge sample run

Once the source data is obtained, the source descriptor is created, and the attributes file is created the last step is to configure the run script since it is the controlling xml file that tells OYSTER where to find all the other files. Due to this being a merge-purge ER run, no input identities or output identities files should be specified in the run script. As mentioned above the run script should be stored in the root OYSTER folder as this is where the OYSTER program is expecting the file to reside. The file for this sample is named 'MergePurgeRunScript.xml' and is shown in Figure 36.

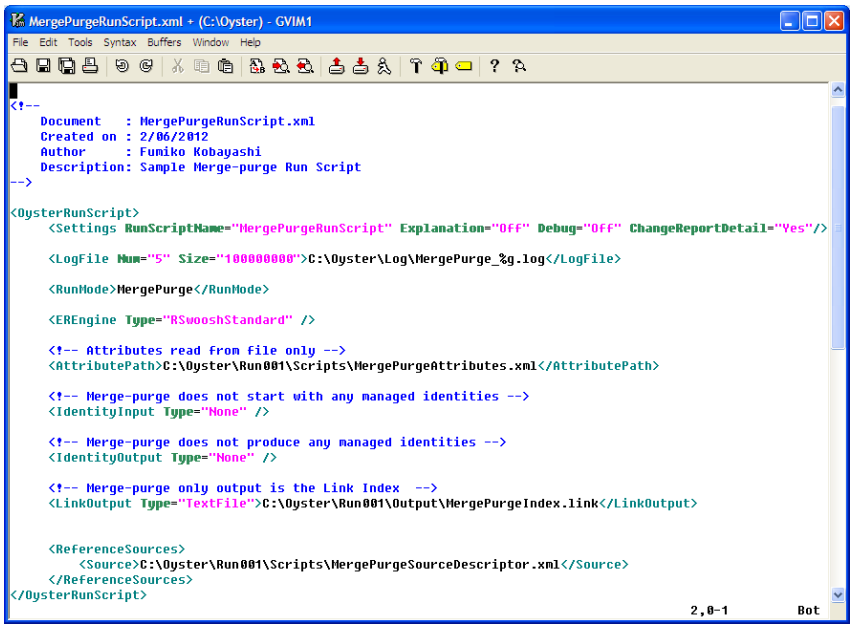


Figure 36: Run Script used for Merge-purge Example

Now that all the scripts for the Merge-purge sample have been created we can run OYSTER. To run OYSTER you double click on the Oyster.bat file (highlighted in Figure 37) that was described earlier in the Launching OYSTER section of this document.

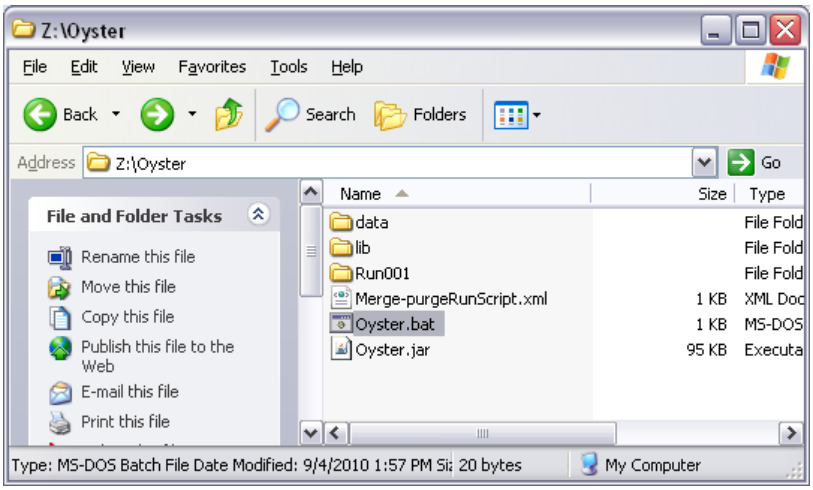


Figure 37: Oyster.bat file location

This action opens a command prompt and calls the Oyster.jar file to run via the command line. The command prompt requests that you type the name of the Run Script that was created, shown in Figure 38. Be sure to include the .xml extension along with the file name. This file name is not case sensitive.

Figure 38: Command prompted opened by Oyster.bat file

Once the name of the run script has been specified, as shown in Figure 39, press enter. (Please see the Invoking the OYSTER Run Script section for more information.)

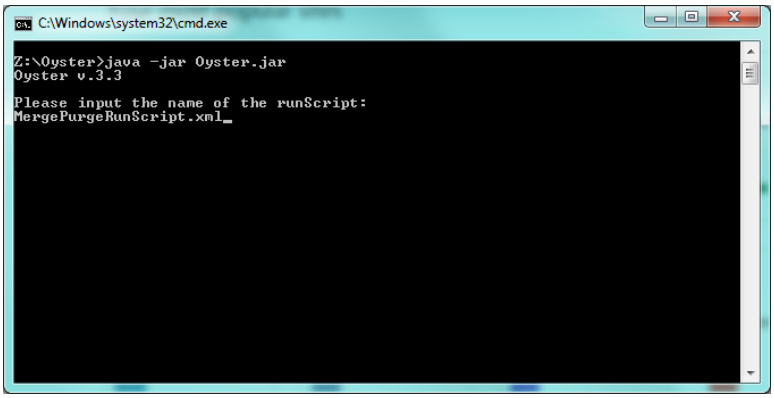


Figure 39: Command Prompt with RunScript name typed

Once the run is complete you will see the results of the run written to the command box window.

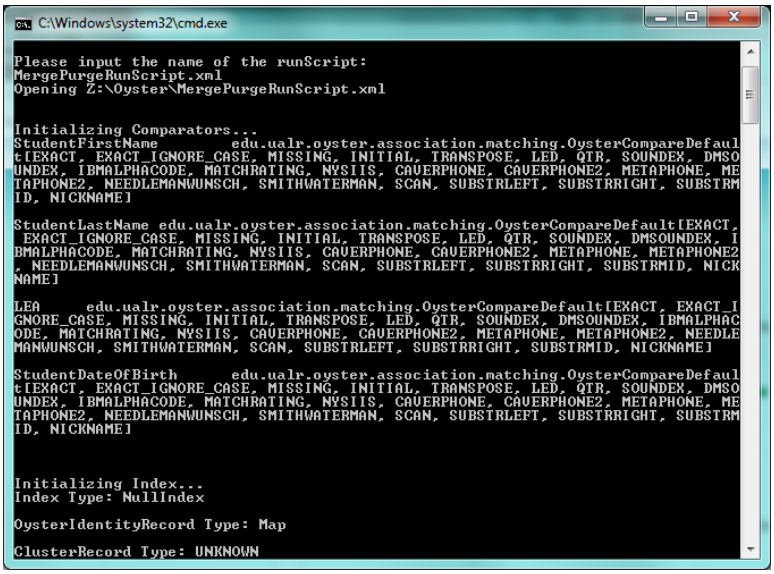


Figure 40: Information generated by OYSTER run to Command box - 1

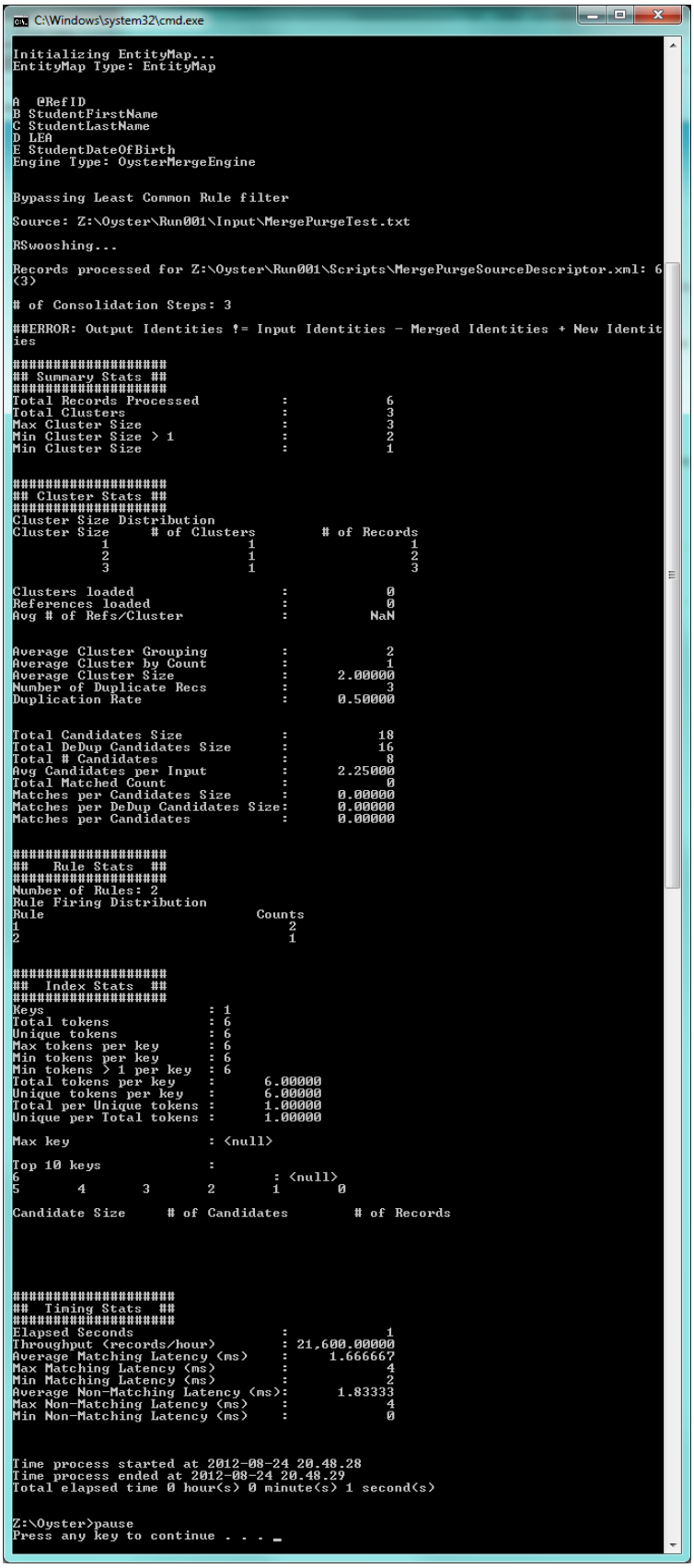


Figure 41: Information generated by OYSTER run to Command box - 2

By examining the output, as shown in Figure 40 and Figure 41, you can see that OYSTER processed 6 references and found that these 6 references belong to 3 real-world identities (Clusters).

Although multiple output files were created, the only output file desired for a Merge-purge run is the link file. This file can be found in the Output folder shown in Figure 42.

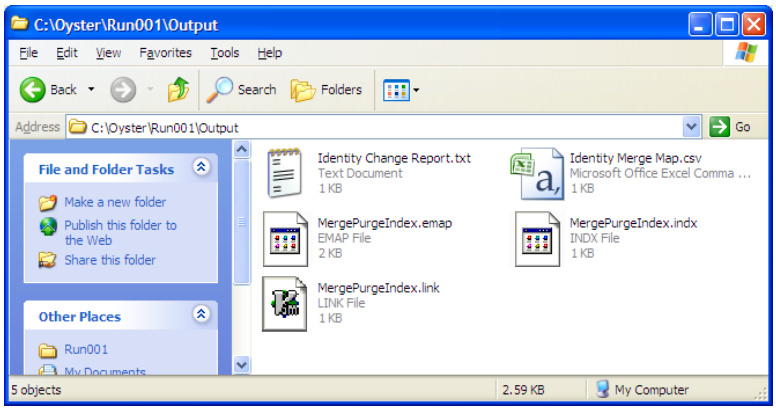


Figure 42: Folder containing output of Merge-purge OYSTER run

When the MergePurgeIndex.link file is opened, as shown in Figure 43, it lists all the references that were read in from the source by their source ID, it also lists the OYSTER ID that was assigned to each reference and what rule it used, if any, to match the reference to another reference. The references that share the same OYSTER ID compose the linked records meaning they are the same real-world entity as discovered by performing the Merge-Purge with OYSTER.

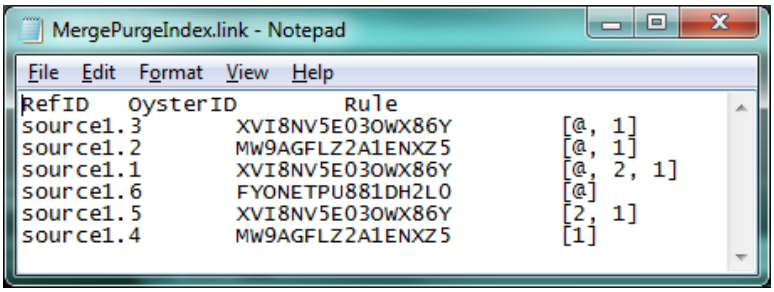


Figure 43: Link file generated by OYSTER Merge-purge run

This sample run was done using a delimited text file. Examples of how to connect to a Fixed Width text file, a Microsoft Access DB, MySQL, and Microsoft SQLServer can be seen in the OYSTER Reference Guide.

Identity Capture

Figure 44 illustrates the dataflow for an OYSTER run configured to perform Identity Capture.

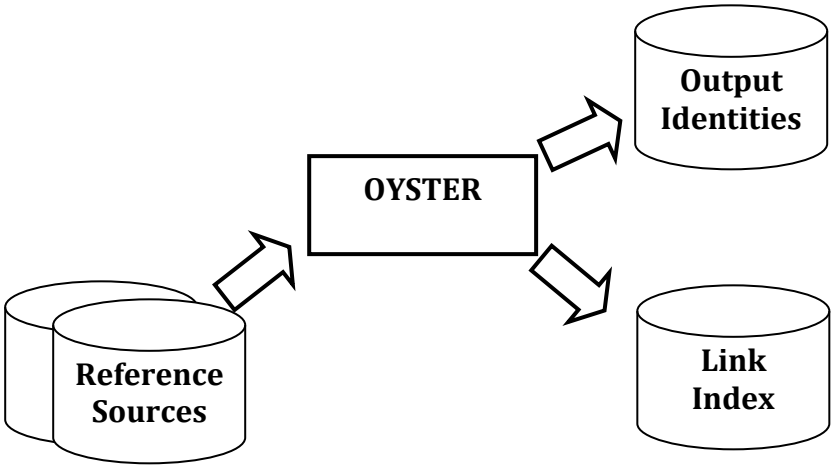


Figure 44: Identity capture dataflow

Configuration

When OYSTER is configured to perform identity capture there should be no input identities specified in the OysterRunScript but a link file and output identity file should be specified, the code is illustrated in Figure 45.

```
      <!-- Identity Input Selection -->
      <IdentityInput Type="None"/>

      <!-- Identity Output Selection -->
      <IdentityOutput
Type="TextFile">Z:\Oyster\Output.idty</IdentityOutput>

      <!-- Link Output Selection -->
      <LinkOutput Type="TextFile">Z:\Oyster\Index.link</LinkOutput>
```

Figure 45: Defined Input and Output for Identity Capture

The identity capture co
nfiguration allows for the identities that were discovered during the run to be saved at the end of the run.

Lastly, the RunMode should be set to "IdentityCapture" as shown in

Figure 46.

```
<RunMode>IdentityCapture</RunMode>
```

Figure 46: RunMode Set to "IdentityCapture" in OysterRunScript for Identity Capture Run

Example

This run will use the test source file named 'IdentityCaptureTest.txt', shown in Figure 47. This data consists of the same six references that were used for the previous Merge-purge example. Note that the same source file used in the Merge-purge example could have been used without creating a copy with a new name by placing the path to the Merge-purgeTest.txt file in the OysterSourceDescriptor.xml defined for this example. This was not done in order to provide this example with a sense of autonomy.

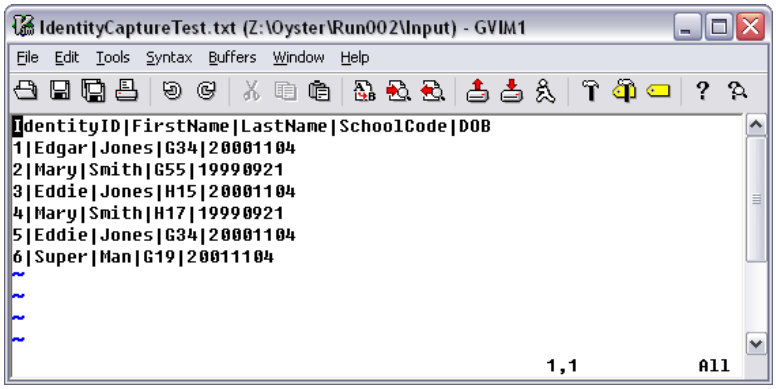


Figure 47: Identity Capture Source Input File

After analyzing the source data the source descriptor file can be created and is named 'IdentityCaptureSourceDescriptor.xml'. This file is shown in Figure 48.

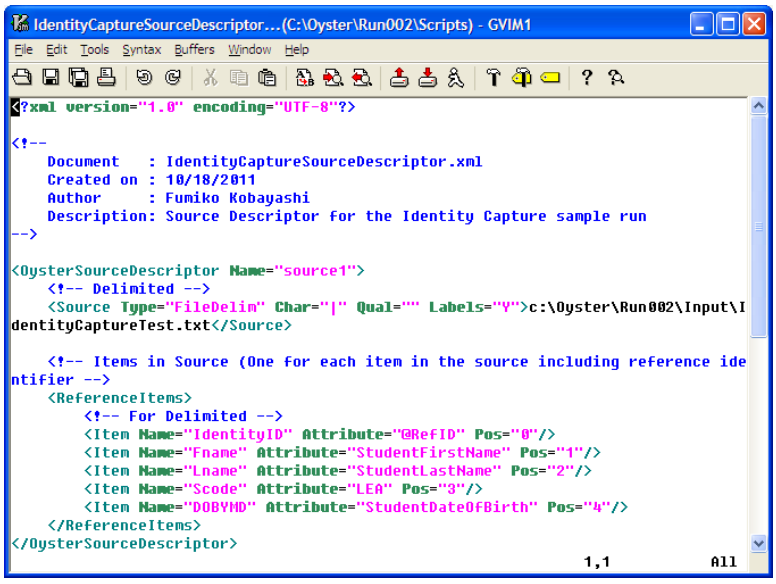


Figure 48: Source Descriptor Defined For Identity Capture Example

Following the same process as was performed when setting up the merge-purge example, once the source descriptor is defined the source attributes file must also be defined. This file is stored in the Source folder along with the Source Descriptor file. The attributes file is used to define the attributes in the source along with the algorithm used to compare the

attributes and the matching (identity) rules used when ER is performed. For this sample run two identity rules will be used. The first rule says that the reference will be considered equivalent if the FirstName, LastName, and DOB attributes match. The second rules states that the references are equivalent if the LastName, DOB, and SchoolCode (LEA) match. These are the same rules that were used for the merge-purge example. The source attribute file is named 'IdentityCaptureAttributes.xml' and is depicted in Figure 49.

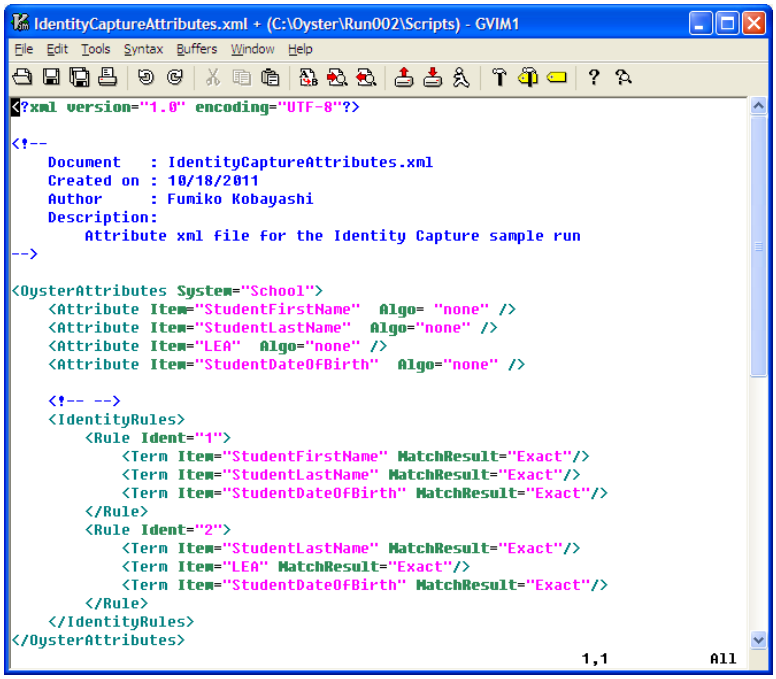


Figure 49: Attributes File Defined For the Identity Capture Example

The attributes file may look familiar. This is due to the fact that the same source records were used for both the merge-purge example and this identity capture example. Due to this, both attributes files are defined identically.

As with the merge-purge example, the last file that needs to be created is the RunScript for this example. For this identity capture example, no input identity file should be specified in the Run Script but both the output identity file and the link files should be specified. The Run Script should again be stored in the root OYSTER folder as this is where the OYSTER program is expecting the file to reside. The file for this sample is named 'IdentityCaptureRunScript.xml' and is shown in Figure 50.

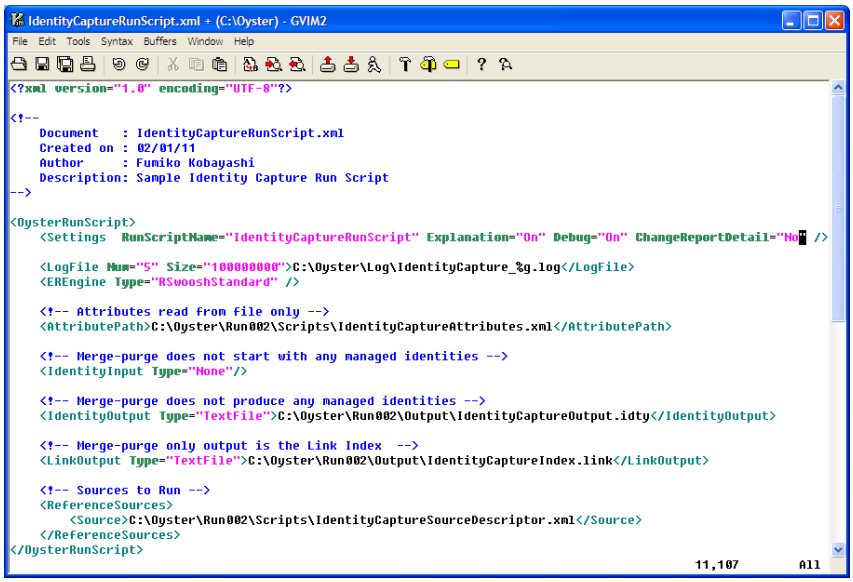


Figure 50: RunScript for Identity Capture Example

Now that all the scripts for the Identity capture example have been created we can run OYSTER. This process is depicted in Figure 37, Figure 38, and Figure 39 and described in their surrounding text in the Example section.

Once the run is complete the output for the run will be written to the command box by OYSTER. This output is shown in Figure 51 and Figure 52:

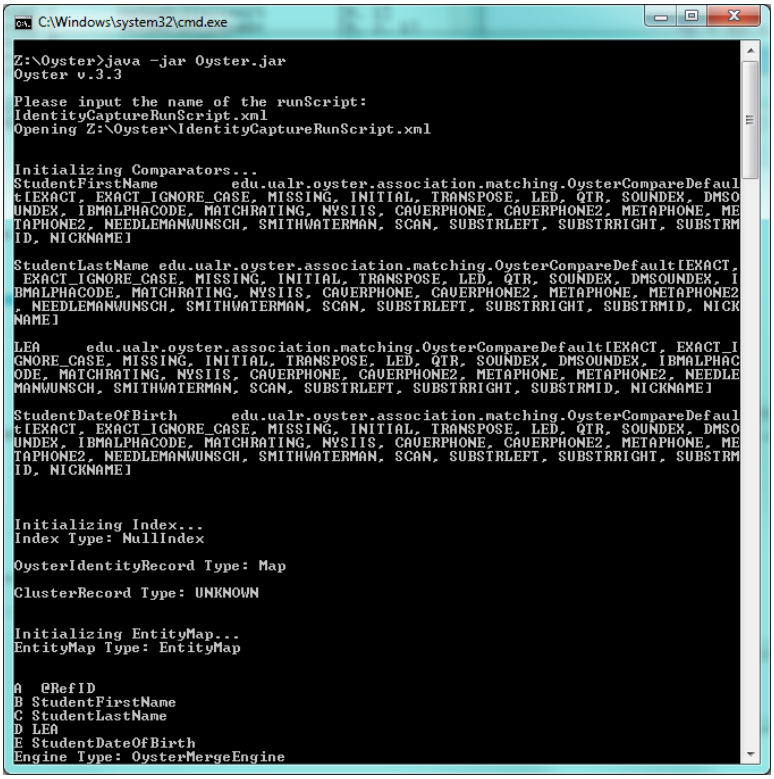


Figure 51: Output written to command box by OYSTER run - 1

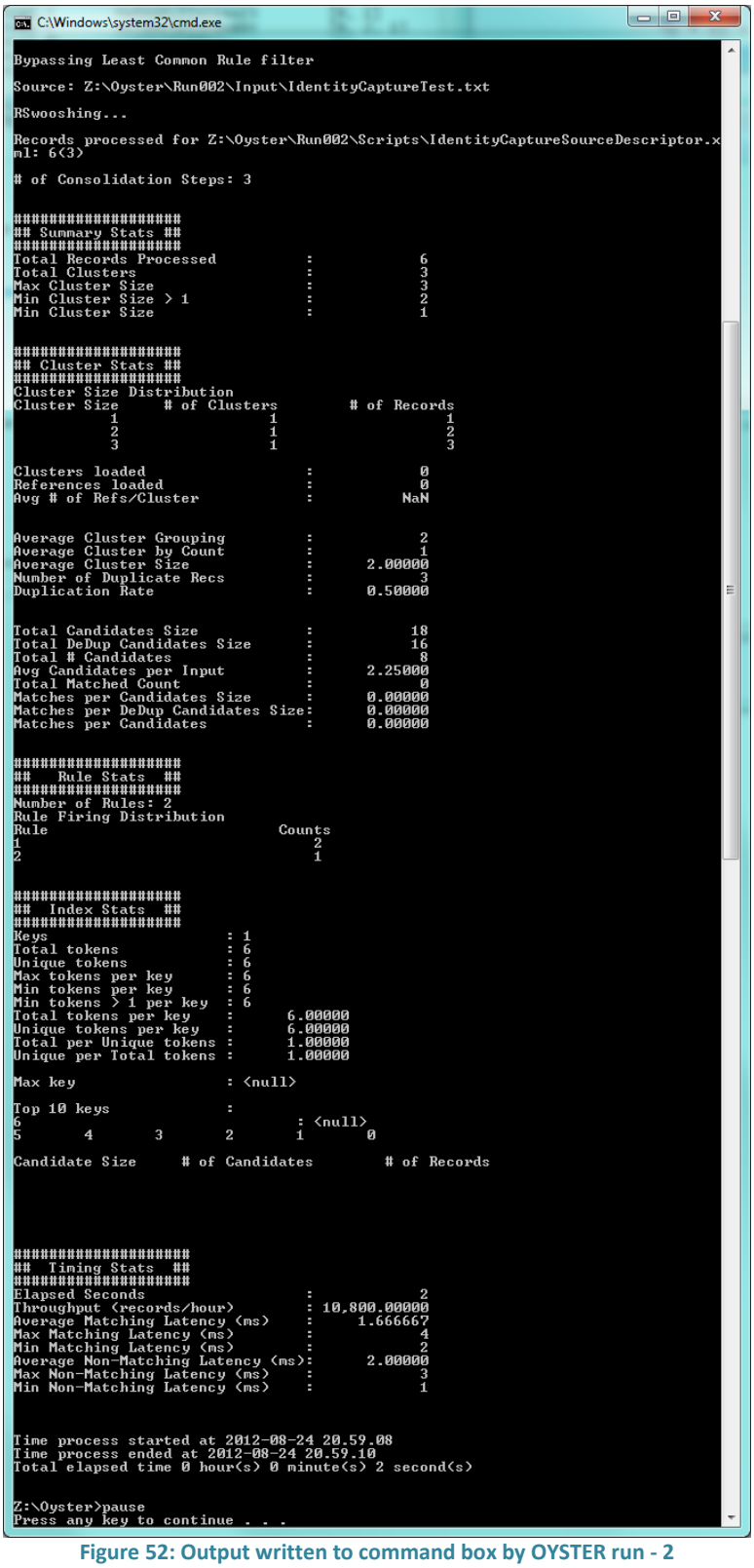
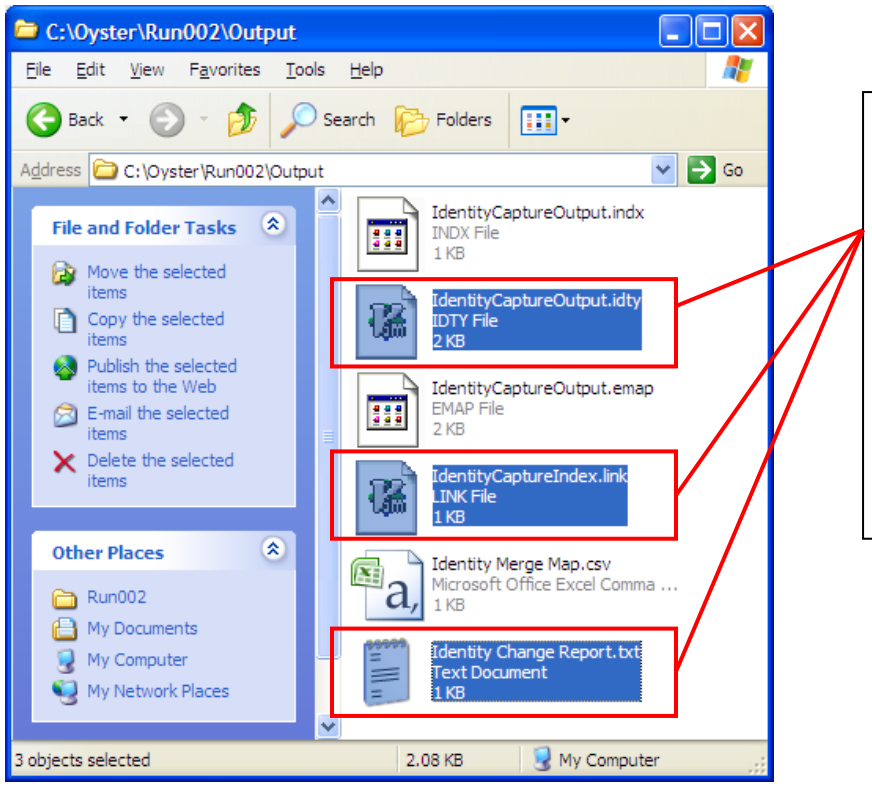


Figure 52: Output written to command box by OYSTER run - 2

By examining the output you can see that OYSTER processed 6 references and found that these 6 references belong to 3 real-world identities (groups). These results are identical to

the results from the merge-purge example. This was expected since the same source records were used in both examples and the same rules were used to match those records. The difference between the merge-purge and the identity capture is that the identity capture example stored the resulting identities of the ER into an output file in addition to creating an identical link file as was seen in the merge-purge example. Both of these output files can be seen in the Z:\Oyster\Run002\Output folder, as shown in Figure 53.



The .link, .idty, and Identity Change Report.txt files are the output files that matter for this run. The other files are generated by assigning the Debug attribute in the Settings tag of the run script the value of "On".

Figure 53: Contents of the Output Folder for Identity Capture Example

The link file, shown in Figure 54, lists all the references that were read in from the source by their source ID, it also lists the OYSTER ID that was assigned to each reference and what rule it used, if any, to match the reference to another reference. The references that share the same OYSTER ID compose the linked records meaning they are the same real-world entity.

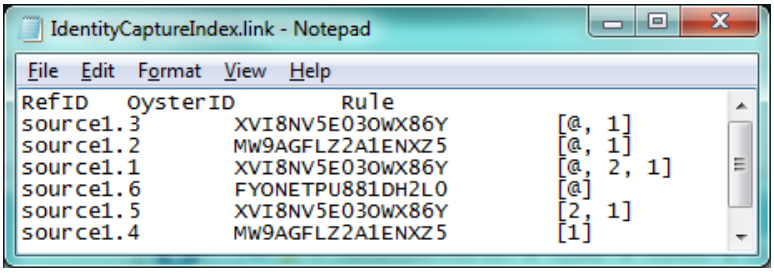


Figure 54: Link File Generated by Identity Capture Example

As discussed previously, identity capture is a form of entity resolution in which the system builds (learns) a set of identities from the references it processes rather than starting with

a known set of identities. This set of identities is stored in the IdentityCaptureOutput.idty file, shown in Figure 55, which was generated by this run preserving the information derived from the three clusters of references.

IdentityCaptureOutput.idty - Notepad

File Edit Format View Help

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
        <Metadata>
                <Modifications>
                        <Modification ID="1" OysterVersion="3.3" Date="2012-08-24 20.59.09"
RunScript="IdentityCaptureRunScript" />
                </Modifications>
                <Attributes>
                        <Attribute Name="@RefID" Tag="A"/>
                        <Attribute Name="StudentFirstName" Tag="B"/>
                        <Attribute Name="StudentLastName" Tag="C"/>
                        <Attribute Name="LEA" Tag="D"/>
                        <Attribute Name="StudentDateOfBirth" Tag="E"/>
                </Attributes>
        </Metadata>
        <Identities>
        <Identity Identifier="FYONETPU881DH2L0" CDate="2012-08-24">
                <References>
                        <Reference>
                                <Value>A^source1.6|B^Super|C^Man|D^G19|E^20011104</Value>
                                <Traces/>
                        </Reference>
                </References>
        </Identity>
        <Identity Identifier="MW9AGFLZ2A1ENXZ5" CDate="2012-08-24">
                <References>
                        <Reference>
                                <Value>A^source1.2|B^Mary|C^Smith|D^G55|E^19990921</Value>
                                <Traces/>
                        </Reference>
                        <Reference>
                                <Value>A^source1.4|B^Mary|C^Smith|D^H17|E^19990921</Value>
                                <Traces/>
                        </Reference>
                </References>
        </Identity>
        <Identity Identifier="XVI8NV5EO3OWX86Y" CDate="2012-08-24">
                <References>
                        <Reference>
                                <Value>A^source1.1|B^Edgar|C^Jones|D^G34|E^20001104</Value>
                                <Traces/>
                        </Reference>
                        <Reference>
                                <Value>A^source1.3|B^Eddie|C^Jones|D^H15|E^20001104</Value>
                                <Traces/>
                        </Reference>
                        <Reference>
                                <Value>A^source1.5|B^Eddie|C^Jones|D^G34|E^20001104</Value>
                                <Traces/>
                        </Reference>
                </References>
        </Identity>
        </Identities>
</root>
```

Figure 55: IdentityCaptureOutput.idty File

By examining the identity structures created in the identity capture configuration you can see how each one directly corresponds to one of the Link Indexes shown in Figure 54. The OYSTER IDs in Figure 54 correspond to the Identity Identifiers in Figure 55.

As with the merge-purge example, this sample run was done using a delimited text file. Examples of how to connect to a Fixed Width text file, a Microsoft Access DB, MySQL, and Microsoft SQLServer can be seen in the OYSTER Reference Guide.

Identity Build from Assertions

Figure 56 illustrates the dataflow for an OYSTER run configured to build identities from an assertion file.

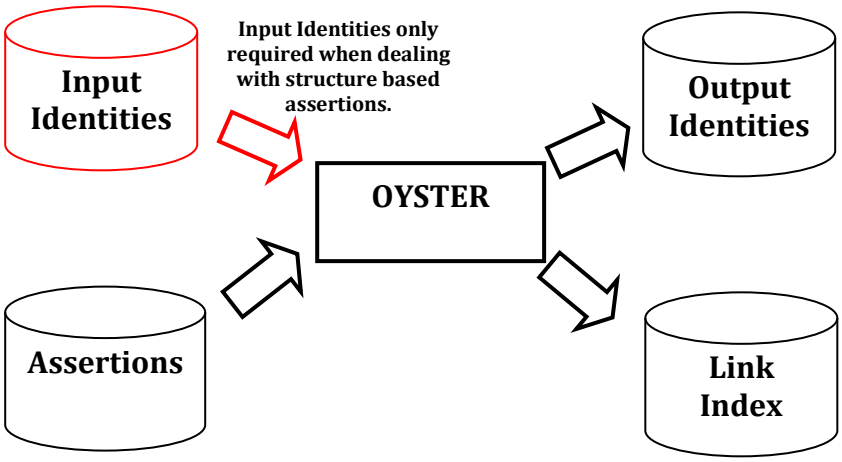


Figure 56: Dataflow to build identities from Assertions.

As mentioned previously, there are four types of Assertions:

- Reference to Reference
- Reference to Structure
- Structure To Structure
- Split Structure

Each of these is shown in the following sections.

Reference to Reference Configuration

When OYSTER is configured to build identities from an assertion file input identities are optional and if specified the new asserted identities will be added into the existing idty structure. A link file and output identity file must be specified. This is shown in Figure 57.

```
      <!-- Identity Input Selection -->
      <IdentityInput Type="None"/>

      <!-- Identity Output Selection -->
      <IdentityOutput
Type="TextFile">Z:\Oyster\Output.idty</IdentityOutput>

      <!-- Link Output Selection -->
      <LinkOutput Type="TextFile">Z:\Oyster\Index.link</LinkOutput>
```

Figure 57: Defined Input and Output for Building Assertions

OYSTER can use the identity capture architecture to build a set of identities from a set of assertions. Reference to Reference Assertions represent knowledge about one or more known entity identities. The identities built through this process can be used as an input when performing Identity Resolution or Identity Update to force a match based on the previous knowledge represented by the assertions.

Lastly, the RunMode should be set to "AssertRefToRef" as shown in Figure 58.

```
  <RunMode>AssertRefToRef</RunMode>
```

Figure 58: RunMode Set to "AssertRefToRef" in OysterRunScript for Reference to Reference Assertion Run

Example

Running OYSTER in the Reference to Reference Assertions Configuration allows identity information to be asserted, preserved, and input into later processes (OYSTER runs) that run in the Identity Resolution or Identity Update Configuration. These identities can be built from a set of assertion sources that represent knowledge about the entities.

For this example, the test source file is named 'AssertionsSource.txt', shown in Figure 59. This data consists of four references; each reference is constructed from the following attributes:

- RefID
- FirstName
- LastName
- DOB
- SchCode
- Assert

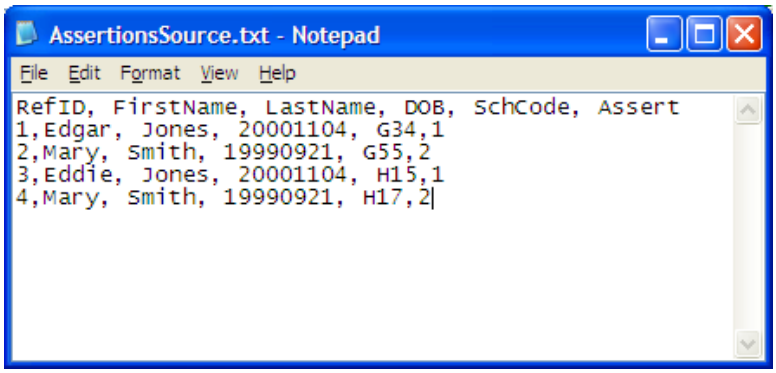


Figure 59: Source file for Identity Build from RefToRef Assertions

Note that based on previous knowledge, an Assert attribute has been added to the source records. The Assert attribute should match for records that are known to represent the same entity. Since a reference to reference assertion run is based off of previous knowledge of the references there is no need to analyze the source data. Based on the knowledge about the source references, the source descriptor file can be created. Using this source file information the source descriptor file, named "AssertionsSourceDescriptor.xml", can be created. This file is shown in Figure 60.

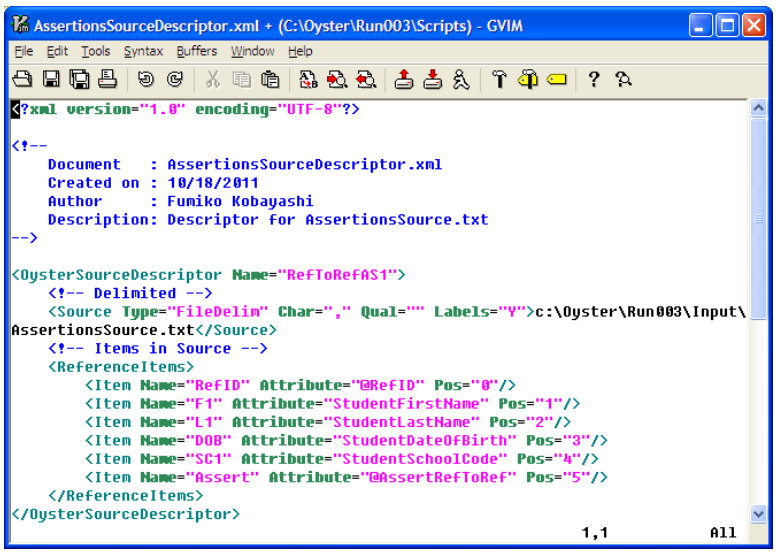


Figure 60: Source Descriptor for Identity Build from Assertions

Note that when creating the source descriptor for an assertion run, as mentioned earlier, an Assert attribute is added to each record to represent the previous knowledge. To identify to OYSTER that this is a RefToRef assertion run there is a predefined key word that must be assigned as the value of the Attribute attribute of the Assert attribute. This keyword is @AssertRefToRef. By looking at Figure 60 you can see that the @AssertRefToRef keyword was used. The @AssertRefToRef keyword forces OYSTER to use RefToRef assertion logic on the source input and to ignore any user defined matching rules. Matching will only occur if the Assert attribute in the source file are the same for multiple records.

Following the same process as was performed in the previous two examples, once the source descriptor is defined the source attributes file must also be defined. This file is stored in the Source folder along with the Source Descriptor file. The attributes file is used to define the attributes in the source along with the algorithms used to compare the attributes and the matching (Identity) rules used when performing ER. For this example run no matching rules will be identified. Instead, as mentioned earlier, the matching will depend solely on the values of the Assert attribute.

The source attribute file is named 'AssertionsAttributes.xml' and is depicted in Figure 61.

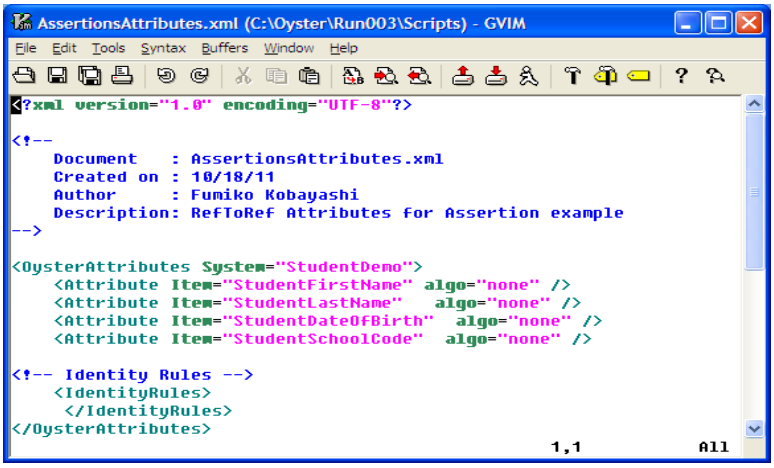


Figure 61: Attributes file for Identity Build from RefToRef Assertions

The defined attributes match the number of distinct values assigned by the Attribute value in the source descriptor. You may also note that there is no rule defined for this run as mentioned earlier but the Rule tag must still be include or the OYSTER run will fail.

As with the previous two examples, the last file that needs to be created is the RunScript for this example. For the attributes example, no input identity file should be specified in the Run Script but both the output identity file and the link files should be specified. The Run Script should again be stored in the root OYSTER folder as this is where the OYSTER program is expecting the file to reside. The file for this sample is named 'RefToRefAttributesRunScript.xml' and is shown in Figure 62.

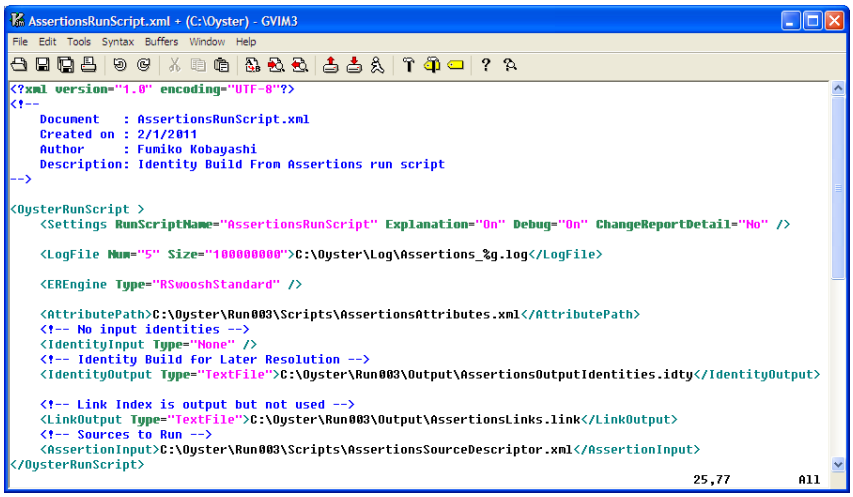


Figure 62: Run Script for Identity Build from RefToRef Assertions

Now that all the scripts for the Assertions example have been created we can run OYSTER. This process is depicted in Figure 37, Figure 38, and Figure 39 and described in their surrounding text in the Example section.

Once the run is complete the output for the run will be written to the command box by OYSTER. This output is shown in Figure 63 and Figure 64.

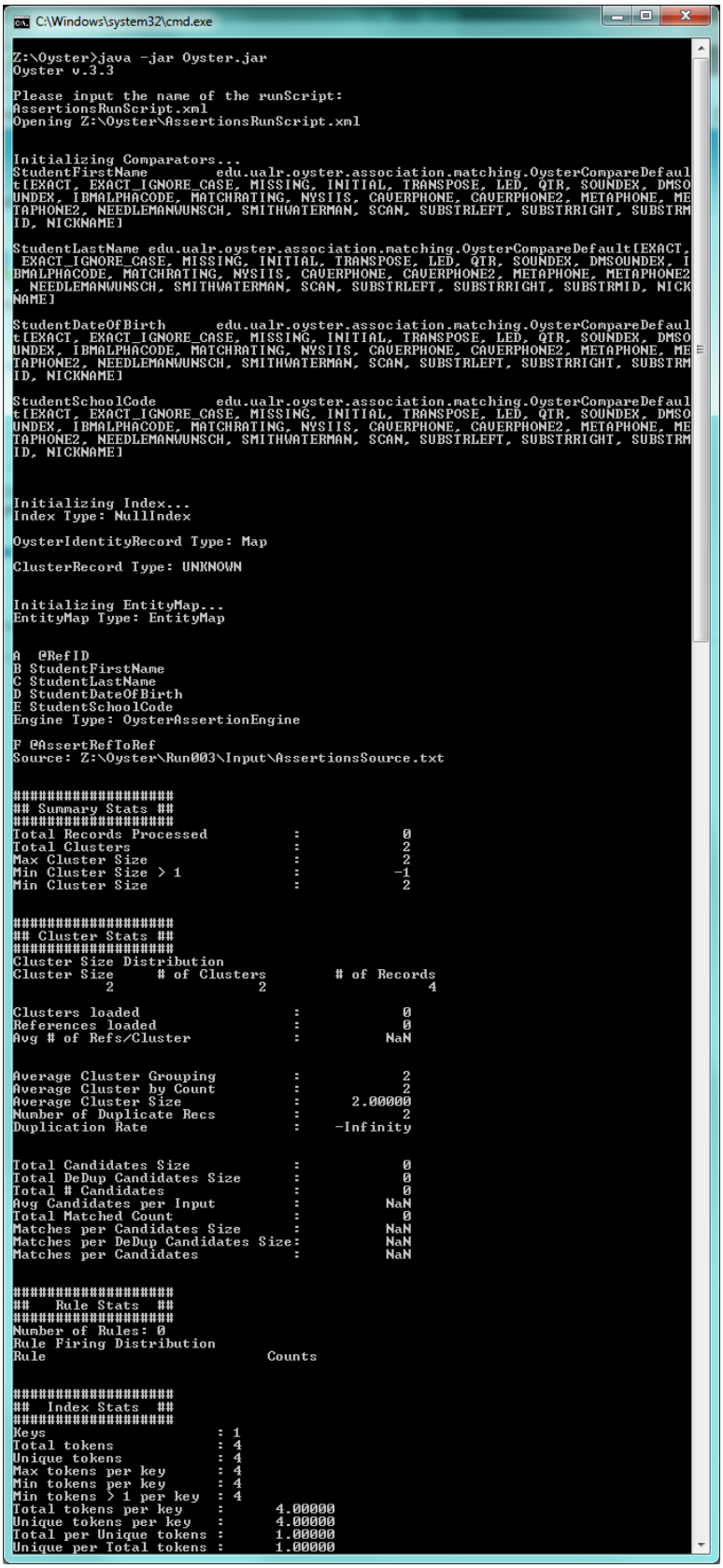


Figure 63: Output to Command Box Generated by OYSTER Run - 1

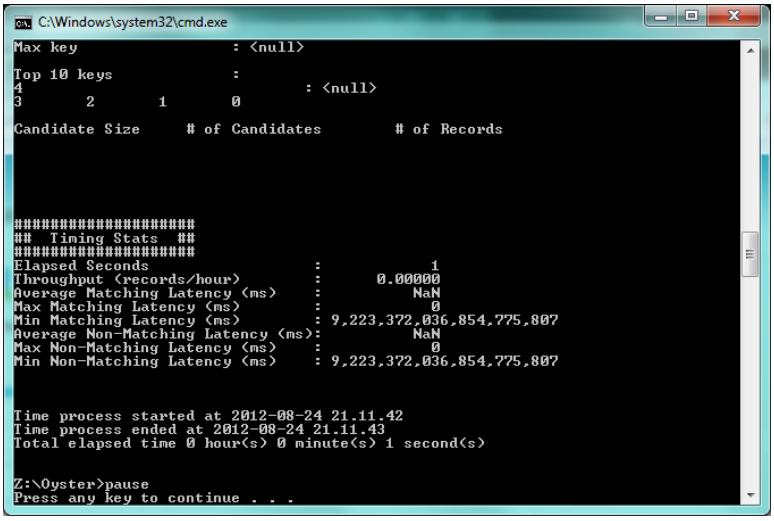


Figure 64: Output to Command Box Generated by OYSTER Run - 2

The statistics for this run may be slightly confusing. According to the statistics, OYSTER processed the 0 records and found they belong to 2 real-world identities, shown in Figure 65. This is due to this being an Assertions run and the references were asserted into equivalence, not matched. This figure also shows that no rules were used for matching and instead all matching was done through assert.

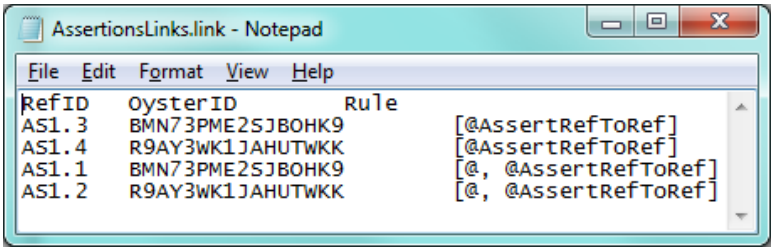


Figure 65: Link file created for Identity Build from Assertions

The entire point of this RefToRef assertion run is to build a set of identities that can be used as input when performing Identity Resolution or Identity Capture. These identities are constructed through the use of previous knowledge about the references. As shown in Figure 66 these 4 references resolve to 2 identities. By assigning the Assert attribute the @AssertRefToRef Attribute value in the source descriptor, it forced OYSTER to match the records with no regard to the other attribute values of the record.

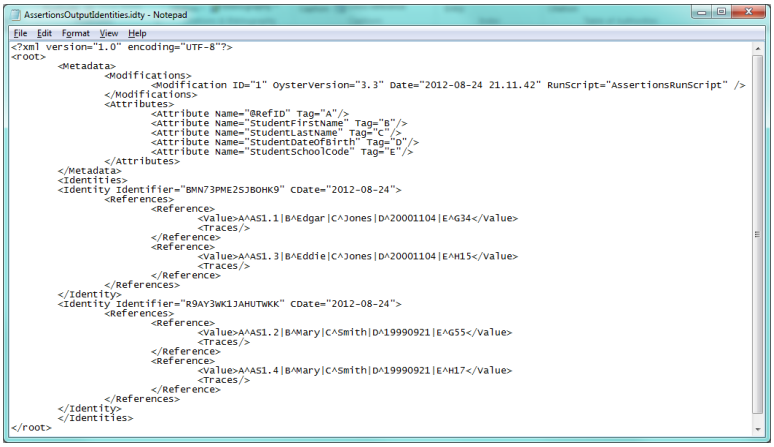


Figure 66: Identity file created for Identity Build from Assertions

As with the previous examples, this sample run was done using a delimited text file. Examples of how to connect to a Fixed Width text file, a Microsoft Access DB, MySQL, and Microsoft SQLServer can be seen in the OYSTER Reference Guide.

Reference to Structure Configuration

When OYSTER is configured for Reference to Structure Assertions, input identities are required since the purpose of this type of assertion is to force a new reference to match an existing identity. A link file and output identity file must also be specified. This is shown in Figure 67.

```
      <!-- Identity Input Selection -->
      <IdentityInput Type="TextFile">Z:\Oyster\Input.idty</IdentityInput>

      <!-- Identity Output Selection -->
      <IdentityOutput
Type="TextFile">Z:\Oyster\Output.idty</IdentityOutput>

      <!-- Link Output Selection -->
      <LinkOutput Type="TextFile">Z:\Oyster\Index.link</LinkOutput>
```

Figure 67: Defined Input and Output for Building Assertions

OYSTER can use the identity capture architecture to build a set of identities from a set of assertions. Reference to Structure Assertions represent knowledge about one or more known entity identities. The identities updated through this process can be used as an input when performing Identity Resolution or Identity Update to force a match based on the previous knowledge represented by the assertions. This type of assertion is used to force a reference to match an existing identity that would not match based on any defined rule.

Lastly, the RunMode should be set to "AssertRefToStr" as shown in Figure 68.

```
  <RunMode>AssertRefToStr</RunMode>
```

Figure 68: RunMode Set to "AssertRefToRef" in OysterRunScript for Reference to Reference Assertion Run

Example

Running OYSTER in the Reference to Structure Assertion configuration allows reference information to be injected into an existing identity, preserved, and input into later processes (OYSTER runs) that run in the Identity Resolution or Identity Update Configuration. These identities can be built from a set of assertion sources that run against a previous set of identities and represent knowledge about the entities.

For this example, the test source file is named 'AssertionsSource.txt', shown in Figure 69. This data consists of one reference; the reference is constructed from the following attributes:

- RefID
- FirstName
- LastName
- DOB
- SchCode
- @AssertRefToStr

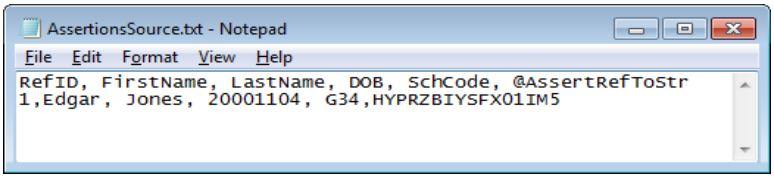


Figure 69: Source file for Identity Build from RefToStr Assertions

Note that based on previous knowledge, an @AssertRefToStr attribute has been added to the source records. The @AssertRefToStr attribute should contain the OYSTER ID of an identity in the input idty file in which the source reference is to be inserted into. Since a reference to structure assertion run is based off of previous knowledge of the references there is no need to analyze the source data. Based on the knowledge about the source references, the source descriptor file can be created. Using this source file information the source descriptor file, named "AssertionsSourceDescriptor.xml", can be created. This file is shown in Figure 70.

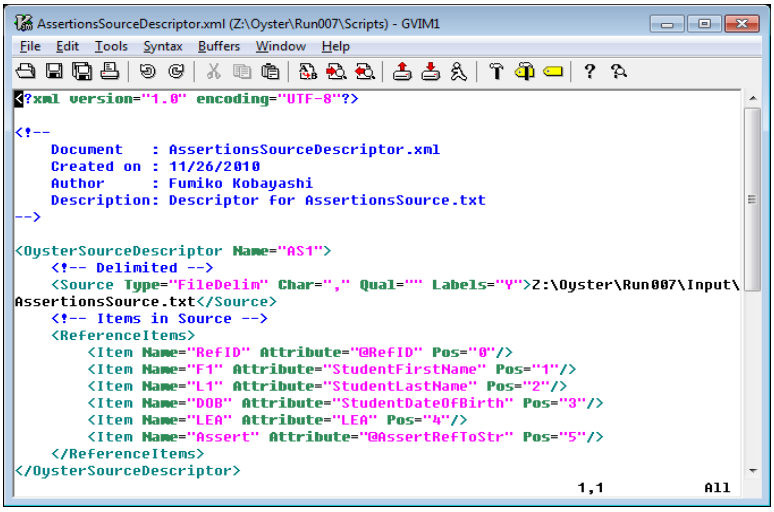


Figure 70: Source Descriptor for Identity Build from Assertions

Note that when creating the source descriptor for a RefToStr assertion run, as mentioned earlier, an @AssertRefToStr attribute is added to each record to represent the previous knowledge. To identify to OYSTER that this is a RefToStr assertion run there is a predefined key word that must be assigned as the value of the Attribute attribute of the @AssertRefToStr attribute. This keyword is @AssertRefToStr. By looking at Figure 70 you can see that the @AssertRefToStr keyword was used. The @AssertRefToStr keyword forces OYSTER to use RefToStr assertion logic on the source input and to ignore any user defined matching rules. Matching will only occur if the @AssertRefToStr attribute in the source file matches existing OYSTER IDs in the idty input file.

Following the same process as was performed in the previous two examples, once the source descriptor is defined the source attributes file must also be defined. This file is stored in the Source folder along with the Source Descriptor file. The attributes file is used to define the attributes in the source along with the algorithms used to compare the attributes and the matching (Identity) rules used when performing ER. For this example run no matching rules will be identified. Instead, as mentioned earlier, the matching will depend solely on the values of the @AssertRefToStr attribute.

The source attribute file is named 'AssertionsAttributes.xml' and is depicted in Figure 71.

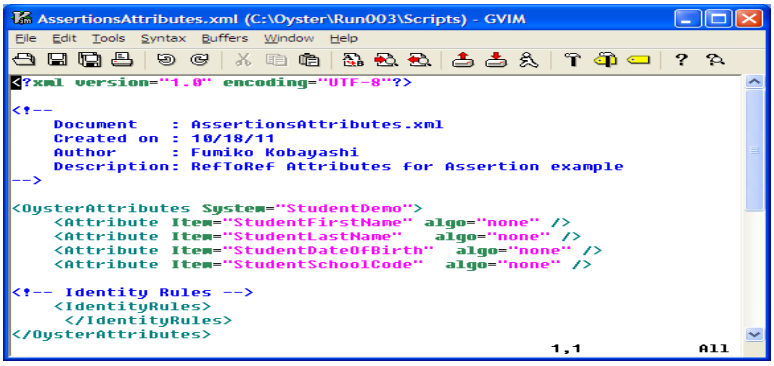


Figure 71: Attributes file for Identity Build from RefToStr Assertions

The defined attributes match the number of distinct values assigned by the Attribute value in the source descriptor. You may also note that there is no rule defined for this run as mentioned earlier but the Rule tag must still be include or the OYSTER run will fail.

As with the previous two examples, the last file that needs to be created is the RunScript for this example. For the RefToStr example, the input identity file, output identity file, and the link files should be specified. The Run Script should again be stored in the root OYSTER folder as this is where the OYSTER program is expecting the file to reside. The file for this sample is named 'RefToRefAttributesRunScript.xml' and is shown in Figure 72.

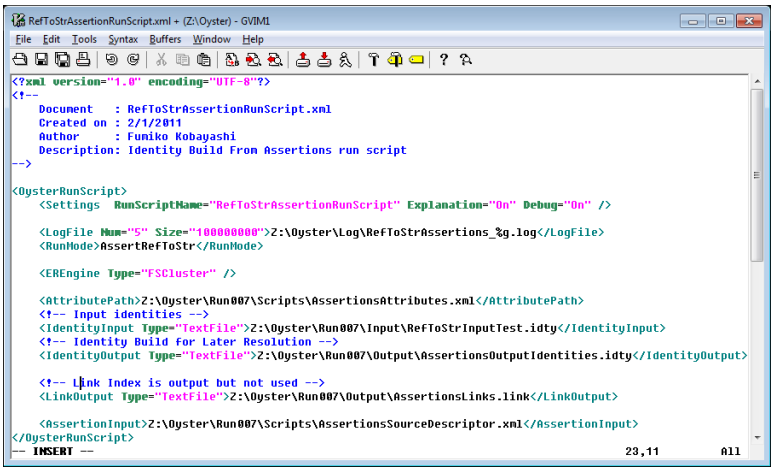


Figure 72: Run Script for Identity Build from RefToRef Assertions

Now that all the scripts for the RefToStr Assertions example have been created we can run OYSTER. This process is depicted in Figure 37, Figure 38, and Figure 39 and described in their surrounding text in the Example section.

Once the run is complete the output for the run will be written to the command box by OYSTER. This output is shown in Figure 73 and Figure 74.

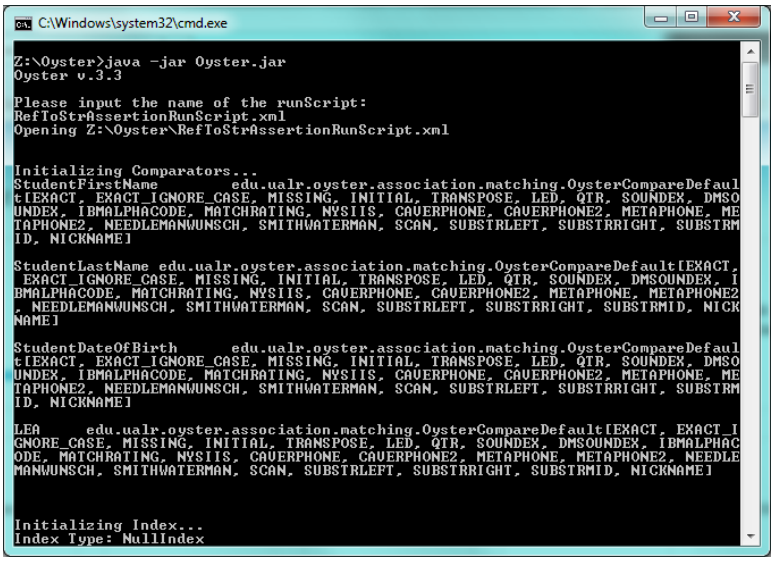


Figure 73: Output to Command Box Generated by OYSTER Run - 1

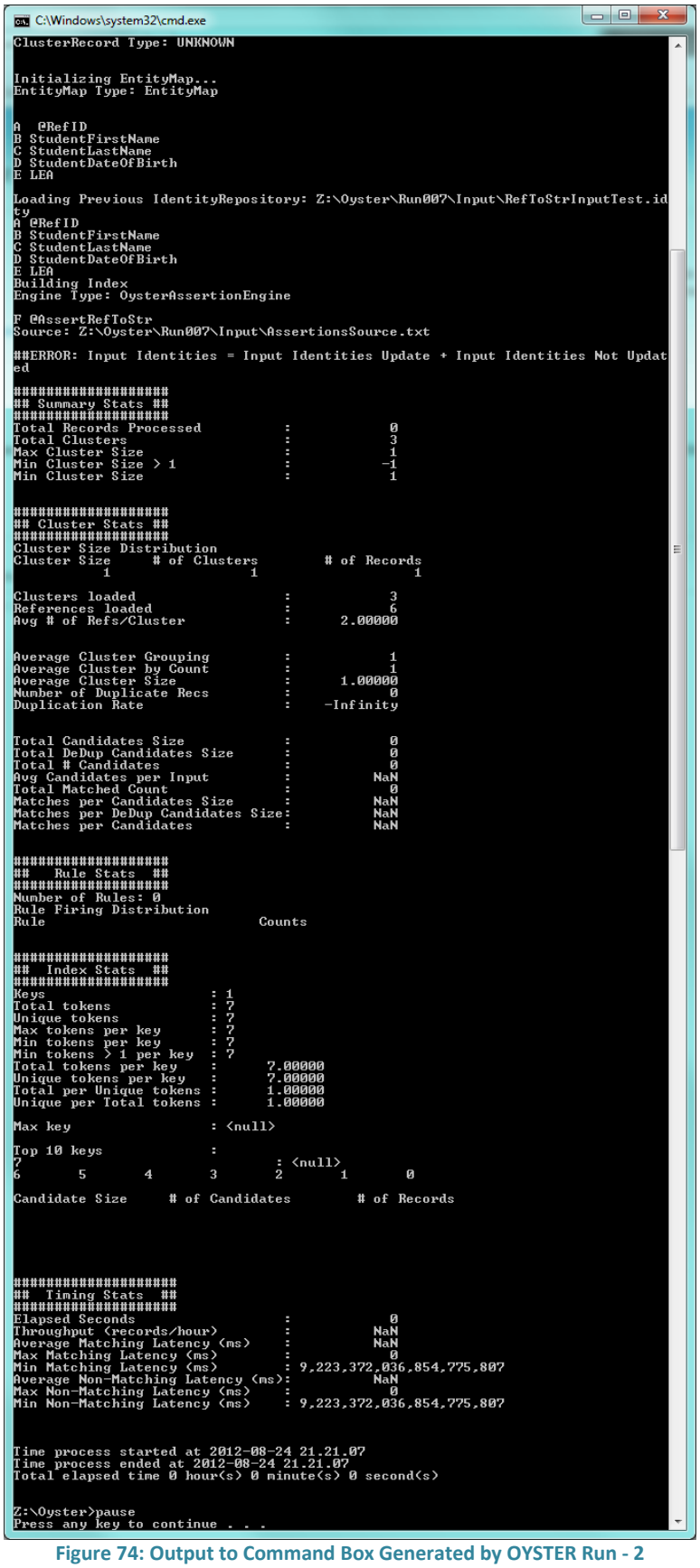


Figure 74: Output to Command Box Generated by OYSTER Run - 2

The statistics for this run may be slightly confusing. According to the statistics, OYSTER processed the 0 records and found they belong to 3 real-world identities, shown inFigure 74. This is due to this being an Assertions run and the references were asserted into equivalence, not matched. Figure 75 shows the Link file with shows the reference AS1.1 was added to the specified Identity, it also shows that no rules were used for matching, all matching was done through assert.

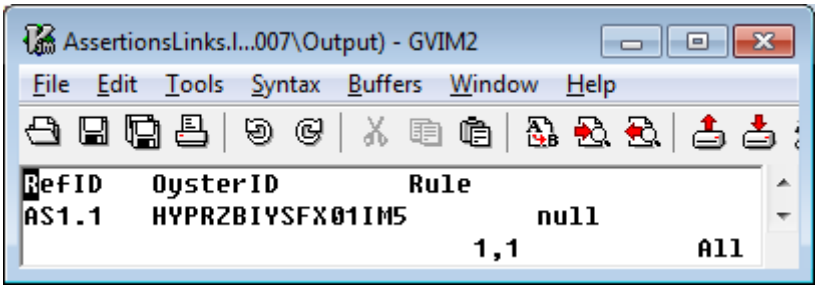


Figure 75: Link file created for Identity Build from Assertions

The entire point of this RefToStr assertion run is to update a set of identities that can be used as input when performing Identity Resolution or Identity Update. These identities are updated through the use of previous knowledge about the references. Figure 76 shows the reference was asserted into the correct identity. By assigning the Assert attribute the @AssertRefToStr Attribute value in the source descriptor, it forced OYSTER to match the records with no regard to the other attribute values of the record.

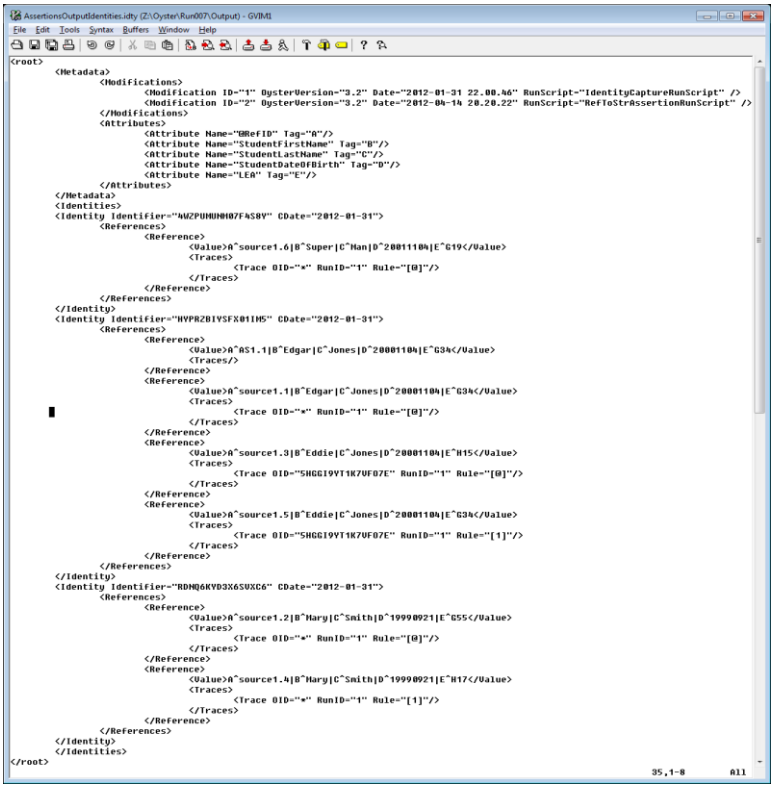


Figure 76: Identity file created for Identity Build from RefToStrAssertions

As with the previous examples, this sample run was done using a delimited text file. Examples of how to connect to a Fixed Width text file, a Microsoft Access DB, MySQL, and Microsoft SQLServer can be seen in the OYSTER Reference Guide.

Structure to Structure Configuration

When OYSTER is configured to build identities from a structure to structure assertion run, the input identities are required since asserted identities will be pulled from the existing idty structure. A link file and output identity file must also be specified. This is shown in Figure 77Figure 57.

```
      <!-- Identity Input Selection -->
      <IdentityInput Type="TextFile">Z:\Oyster\Input.idty</IdentityInput>

      <!-- Identity Output Selection -->
      <IdentityOutput
Type="TextFile">Z:\Oyster\Output.idty</IdentityOutput>

      <!-- Link Output Selection -->
      <LinkOutput Type="TextFile">Z:\Oyster\Index.link</LinkOutput>
```

Figure 77: Defined Input and Output for Building Assertions

OYSTER can use the identity update architecture to build a set of identities from a set of assertions. Structure to Structure Assertions represent knowledge about two or more known entity identities. The identities built through this process can be used as an input when performing Identity Resolution or Identity Update to force a match based on the previous knowledge represented by the assertions.

Lastly, the RunMode should be set to "AssertStrToStr" as shown in Figure 78.

```
  <RunMode>AssertStrToStr</RunMode>
```

Figure 78: RunMode Set to "AssertStrToStr" in OysterRunScript for Structure to Structure Assertion Run

Example

Running OYSTER in the Structure To Structure Assertions configuration allows identity information to be asserted, preserved, and input into later processes (OYSTER runs) that run in the Identity Resolution or Identity Update Configuration. These identities can be built from a set of assertion sources that represent knowledge about the existing entities.

For this example, the test source file is named 'AssertionsSource.txt', shown in Figure 79. This data consists of 2 references; each reference is constructed from the following attributes:

- RefID
- OID
- AssertStrToStrLastName

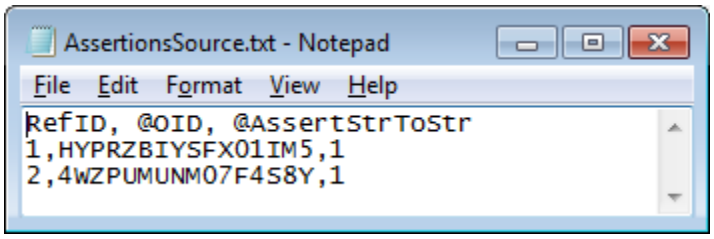


Figure 79: Source file for Identity Build from StrToStr Assertions

Note that based on previous knowledge, an OID and Assert attribute has been configured to force two identities in the identity source to merge. The Assert attribute should match for identities that are known to represent the same entity. Since a Structure to Structure assertion run is based off of previous knowledge of the identities there is no need to analyze the source data. Based on the knowledge about the source references, the source descriptor file can be created. Using this source file information the source descriptor file, named "AssertionsSourceDescriptor.xml", can be created. This file is shown in Figure 80.

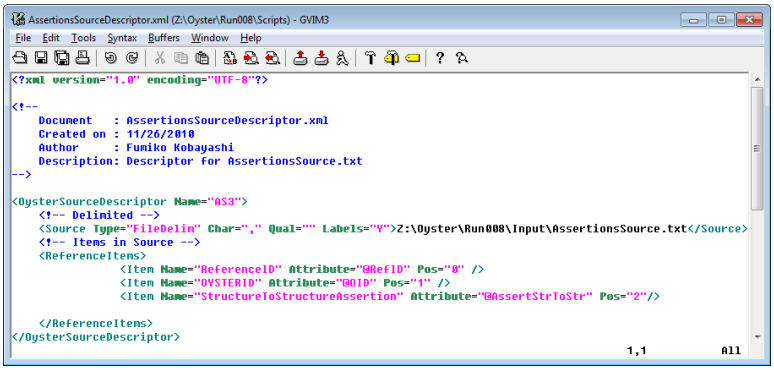


Figure 80: Source Descriptor for Identity Build from Assertions

Note that when creating the source descriptor for a Structure to Structure assertion run, as mentioned earlier, an AssertStrToStr and OID attribute is configured for each record to

represent the previous knowledge. To identify to OYSTER that this is a StrToStr assertion run there is a predefined key word that must be assigned as the value of the Attribute attribute of the Assert attribute. This keyword is @AssertStrToStr. The Keyword @OID must also be specified and correspond to the attribute that represents the OYSTER IDs of the identities to be merged. The @AssertStrToStr keyword forces OYSTER to use StrToStr assertion logic on the source input and to ignore any user defined matching rules. Matching will only occur if the Assert attribute in the source file are the same for multiple records.

Following the same process as was performed in the previous two examples, once the source descriptor is defined the source attributes file must also be defined. This file is stored in the Source folder along with the Source Descriptor file. The attributes file is used to define the attributes in the source along with the algorithms used to compare the attributes and the matching (Identity) rules used when performing ER. For this example run no matching rules will be identified. Instead, as mentioned earlier, the matching will depend solely on the values of the Assert attribute.

The source attribute file is named 'AssertionsAttributes.xml' and is depicted in Figure 81.

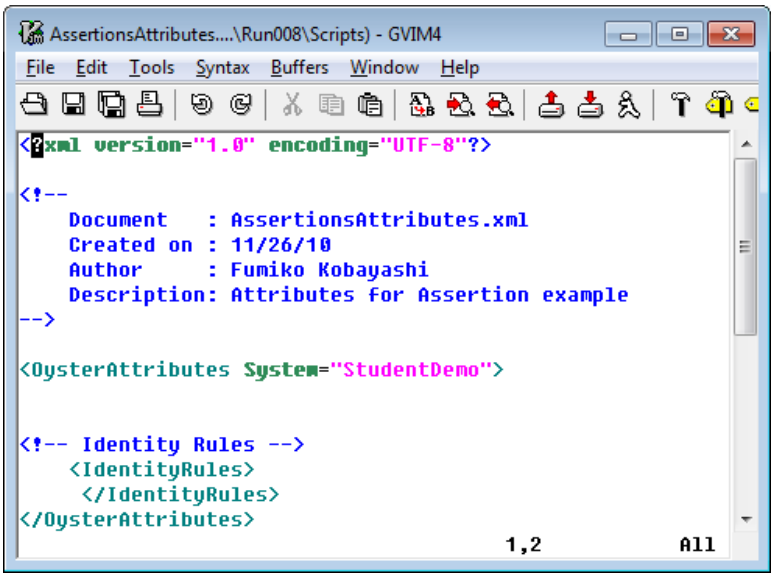


Figure 81: Attributes file for Identity Build from StrToStr Assertions

Since a StrToStr assertion is based only on existing identities in the idty file and the attributes are specified by OYSTER keywords, the attribute file requires no attributes to be defined. You may also note that there is no rule defined for this run as mentioned earlier but the Rule tag must still be include or the OYSTER run will fail.

As with the previous two examples, the last file that needs to be created is the RunScript for this example. For the attributes example, no input identity file should be specified in the Run Script but both the output identity file and the link files should be specified. The Run Script should again be stored in the root OYSTER folder as this is where the OYSTER

program is expecting the file to reside. The file for this sample is named 'StrToStrAttributeRunScript.xml' and is shown in Figure 82.

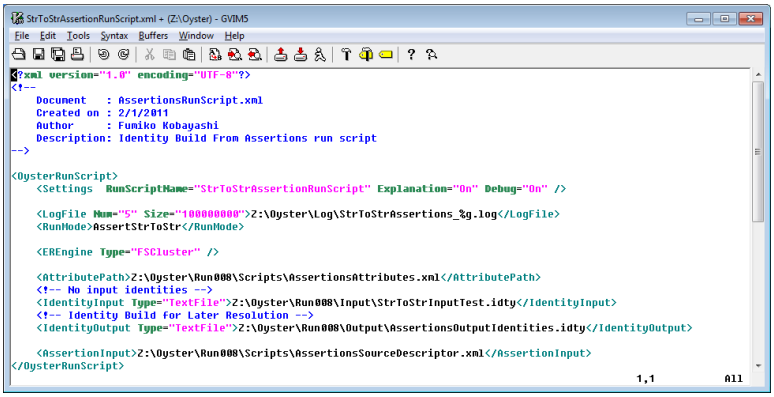


Figure 82: Run Script for Identity Build from StrToStr Assertions

Now that all the scripts for the StrToStr Assertions example have been created we can run OYSTER. This process is depicted in Figure 37, Figure 38, and Figure 39 and described in their surrounding text in the Example section.

Once the run is complete the output for the run will be written to the command box by OYSTER. This output is shown in Figure 83 and Figure 84.

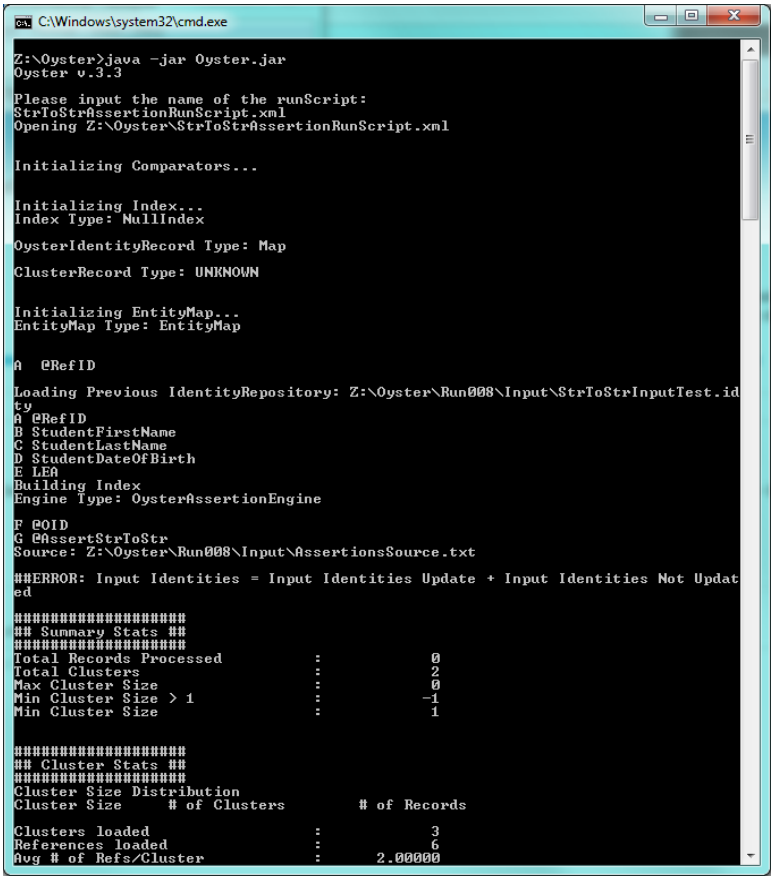


Figure 83: Output to Command Box Generated by OYSTER Run - 1

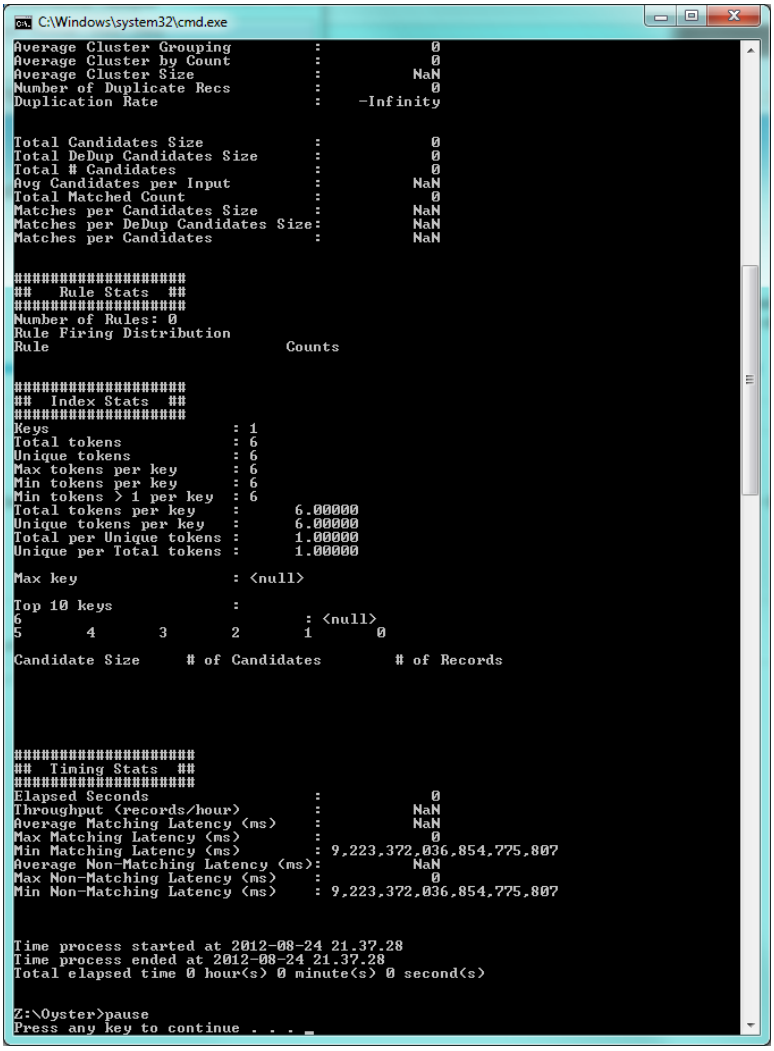


Figure 84: Output to Command Box Generated by OYSTER Run - 2

The statistics for this run may be slightly confusing. According to the statistics, OYSTER processed the 0 records and found they belong to 2 real-world identities. This is due to this being an Assertions run and the Identities were asserted into equivalence, not matched. This figure also shows that no rules were used for matching and instead all matching was done through assert. StrToStr assertion runs do not generate a link output file.

The entire point of this StrToStr assertion run is to merge a set of identities. These identities are constructed through the use of previous knowledge about the references contained in the Identities. As shown in Figure 85 these 4 references resolve to 2 identities. By assigning the Assert attribute the @AssertStrToStr Attribute value in the source descriptor, it forced OYSTER to match the records with no regard to the other attribute values of the record.

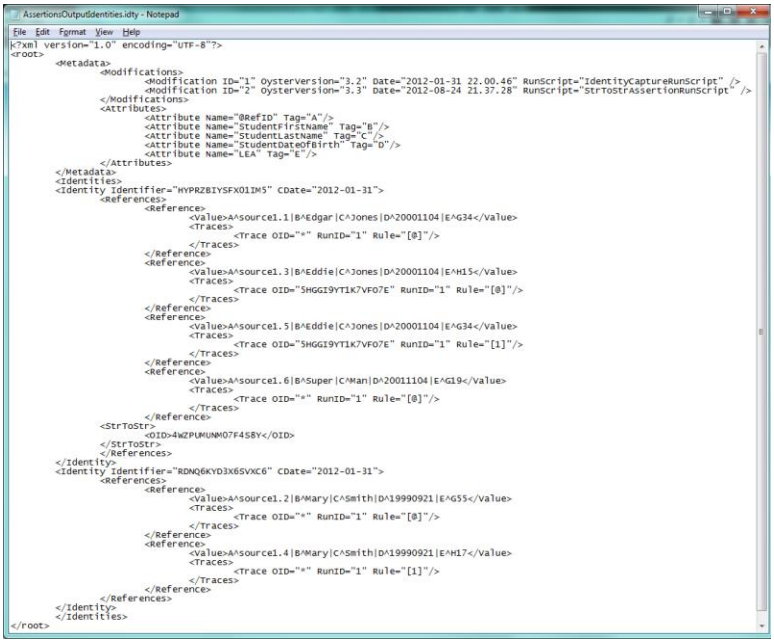


Figure 85: Identity file created for Identity Build from Assertions

As with the previous examples, this sample run was done using a delimited text file. Examples of how to connect to a Fixed Width text file, a Microsoft Access DB, MySQL, and Microsoft SQLServer can be seen in the OYSTER Reference Guide.

Structure Split Configuration

When OYSTER is configured to build identities from a Structure Split assertion, the input identities are required. A link file and output identity file must also be specified. This is shown in Figure 86.

```
      <!-- Identity Input Selection -->
      <IdentityInput Type="TextFile">Z:\Oyster\Input.idty</IdentityInput>

      <!-- Identity Output Selection -->
      <IdentityOutput
Type="TextFile">Z:\Oyster\Output.idty</IdentityOutput>

      <!-- Link Output Selection -->
      <LinkOutput Type="TextFile">Z:\Oyster\Index.link</LinkOutput>
```

Figure 86: Defined Input and Output for Building StrSplit Assertions

OYSTER can use the identity update architecture to split identities based on the assertion information. Structure Split Assertions represent knowledge about false positive resolutions made to one or more known entity identities. The identities built through this process can be used as an input when performing Identity Resolution or Identity Update to force a match based on the previous knowledge represented by the assertions.

Lastly, the RunMode should be set to "AssertSplitStr" as shown in Figure 87.

```
  <RunMode> AssertSplitStr</RunMode>
```

Figure 87: RunMode Set to "AssertSplitStr" in OysterRunScript

Example

Running OYSTER in the Split Structure configuration allows identity information to be asserted, preserved, and input into later processes (OYSTER runs) that run in the Identity Resolution or Identity Update Configuration. These identities can be built from a set of assertion sources that represent knowledge about the entities. This can be used to fix False Positive resolutions from previous runs.

For this example, the test source file is named 'AssertionsSource.txt', shown in Figure 88. This data consists of four references; each reference is constructed from the following attributes:

- RefID
- @RID
- @OID
- @AssertSplitStr

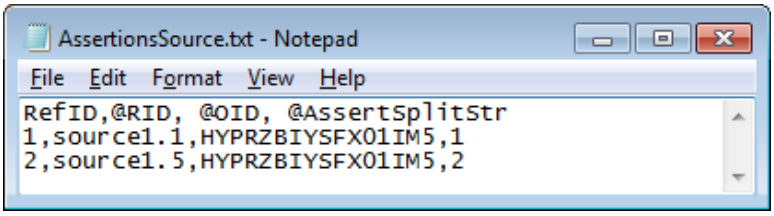


Figure 88: Source file for Identity Build from SplitStr Assertions

Note that based on previous knowledge, an Assert, RID, and OID attribute has been configured as source references. See the reference guide for details on the function of each. Since a Structure Split assertion run is based off of previous knowledge of the references there is no need to analyze the source data. Based on the knowledge about the source references, the source descriptor file can be created. Using this source file information the source descriptor file, named "AssertionsSourceDescriptor.xml", can be created. This file is shown in Figure 89.

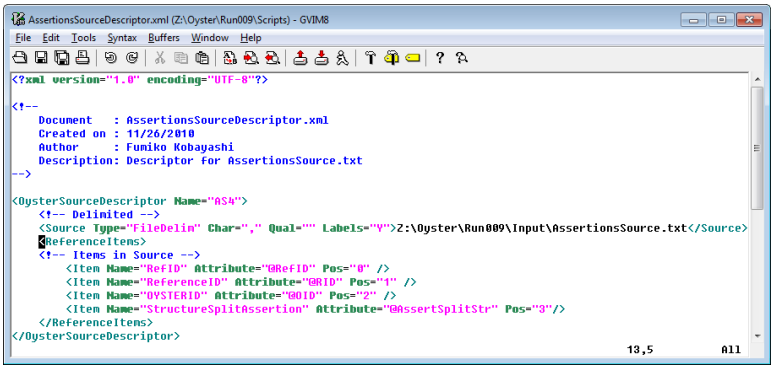


Figure 89: Source Descriptor for Identity Build from Structure Split Assertions

Note that when creating the source descriptor for a SplitStr assertion run, as mentioned earlier, an Assert, OID, and RID attribute are added to each record to represent the previous knowledge. To identify to OYSTER that this is a SplitStr assertion run there is a predefined key word that must be assigned as the value of the Attribute attribute of the Assert attribute. This keyword is @AssertSplitStr. The @AssertSplitStr keyword forces OYSTER to use SplitStr assertion logic on the source input and to ignore any user defined matching rules.

Following the same process as was performed in the previous two examples, once the source descriptor is defined the source attributes file must also be defined. This file is stored in the Source folder along with the Source Descriptor file. The attributes file is used to define the attributes in the source along with the algorithms used to compare the attributes and the matching (Identity) rules used when performing ER. For this example run no matching rules will be identified. Instead, as mentioned earlier, the matching will depend solely on the values of the Assert attribute.

The source attribute file is named 'AssertionsAttributes.xml' and is depicted in Figure 90.

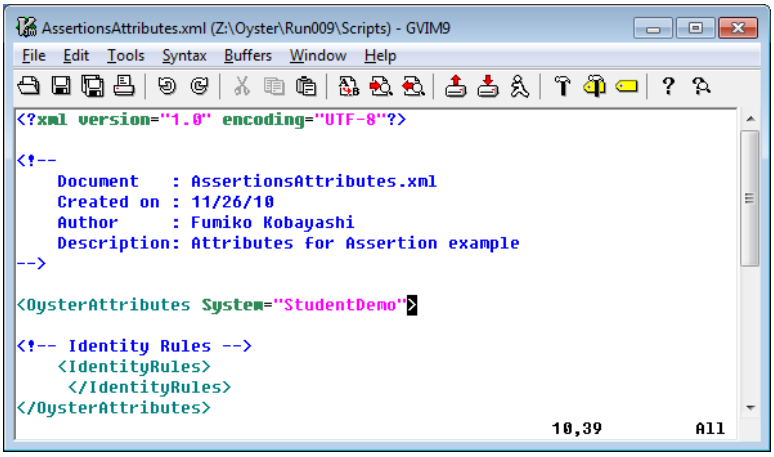


Figure 90: Attributes file for Identity Build from SplitStr Assertions

Since the SplitStr attributes consist of only attributes specified by OYSTER keyword, no attributes need to be configured in the attribute file. You may also note that there is no rule defined for this run as mentioned earlier but the Rule tag must still be include or the OYSTER run will fail.

As with the previous two examples, the last file that needs to be created is the RunScript for this example. For the attributes example, no input identity file should be specified in the Run Script but both the output identity file and the link files should be specified. The Run Script should again be stored in the root OYSTER folder as this is where the OYSTER program is expecting the file to reside. The file for this sample is named 'StrSplitAttributesRunScript.xml' and is shown in Figure 91.

```
StrSplitAssertionRunScript.xml (Z:\Oyster) - GVIM7
File  Edit  Tools  Syntax  Buffers  Window  Help

<?xml version="1.0" encoding="UTF-8"?>
<!--
    Document   : AssertionsRunScript.xml
    Created on : 2/1/2011
    Author     : Fumiko Kobayashi
    Description: Identity Build From Assertions run script
-->

<OysterRunScript>
    <Settings  RunScriptName="StrSplitAssertionRunScript" Explanation="On" Debug="On" />

    <LogFile Num="5" Size="100000000">Z:\Oyster\Log\StrSplitAssertions_%g.log</LogFile>
    <RunMode>AssertSplitStr</RunMode>

    <EREngine Type="FSCluster" />

    <AttributePath>Z:\Oyster\Run009\Scripts\AssertionsAttributes.xml</AttributePath>
    <!-- No input identities -->
    <IdentityInput Type="TextFile">Z:\Oyster\Run009\Input\StrSplitInputTest.idty</IdentityInput>
    <!-- Identity Build for Later Resolution -->
    <IdentityOutput Type="TextFile">Z:\Oyster\Run009\Output\AssertionsOutputIdentities.idty</IdentityOutput>

    <AssertionInput>Z:\Oyster\Run009\Scripts\AssertionsSourceDescriptor.xml</AssertionInput>
</OysterRunScript>
                                                                    17,84         All
```

Figure 91: Run Script for Identity Build from SplitStr Assertions

Now that all the scripts for the Assertions example have been created we can run OYSTER. This process is depicted in Figure 37, Figure 38, and Figure 39 and described in their surrounding text in the Example section.

Once the run is complete the output for the run will be written to the command box by OYSTER. This output is shown in Figure 92 and Figure 93.

```
C:\Windows\system32\cmd.exe

Z:\Oyster>java -jar Oyster.jar
Oyster v.3.3

Please input the name of the runScript:
StrSplitAssertionRunScript.xml
Opening Z:\Oyster\StrSplitAssertionRunScript.xml


Initializing Comparators...


Initializing Index...
Index Type: NullIndex

OysterIdentityRecord Type: Map

ClusterRecord Type: UNKNOWN


Initializing EntityMap...
EntityMap Type: EntityMap


A  @RefID

Loading Previous IdentityRepository: Z:\Oyster\Run009\Input\StrSplitInputTest.id
ty
A @RefID
B StudentFirstName
C StudentLastName
D StudentDateOfBirth
E LEA
Building Index
Engine Type: OysterAssertionEngine

F @RID
G @QID
H @AssertSplitStr
Source: Z:\Oyster\Run009\Input\AssertionsSource.txt


####################
## Summary Stats ##
####################
Total Records Processed          :            0
Total Clusters                   :            5
Max Cluster Size                 :            0
Min Cluster Size > 1             :           -1
Min Cluster Size                 :            1


####################
## Cluster Stats ##
####################
Cluster Size Distribution
Cluster Size      # of Clusters        # of Records

Clusters loaded                  :            3
References loaded                :            6
Avg # of Refs/Cluster            :      2.00000
```

Figure 92: Output to Command Box Generated by OYSTER Run - 1

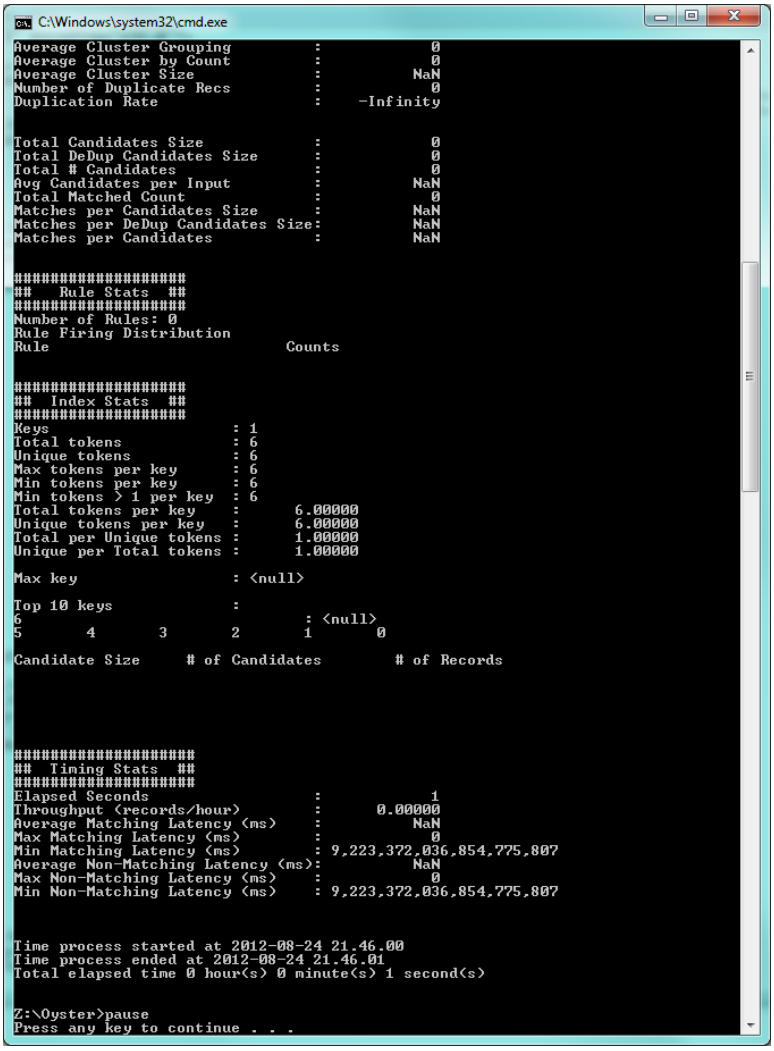


Figure 93: Output to Command Box Generated by OYSTER Run - 2

The statistics for this run may be slightly confusing. According to the statistics, OYSTER processed the 0 records and found they belong to 5 real-world identities. This is due to this being a SplitStr Assertions run and the references were asserted to split, not matched. SplitStr runs generate no link output file.

The entire point of this SplitStr assertion run is to fix false positive resolutions made by previous OYSTER runs. As shown in Figure 94 this caused the specified identity to split into two separate identities.

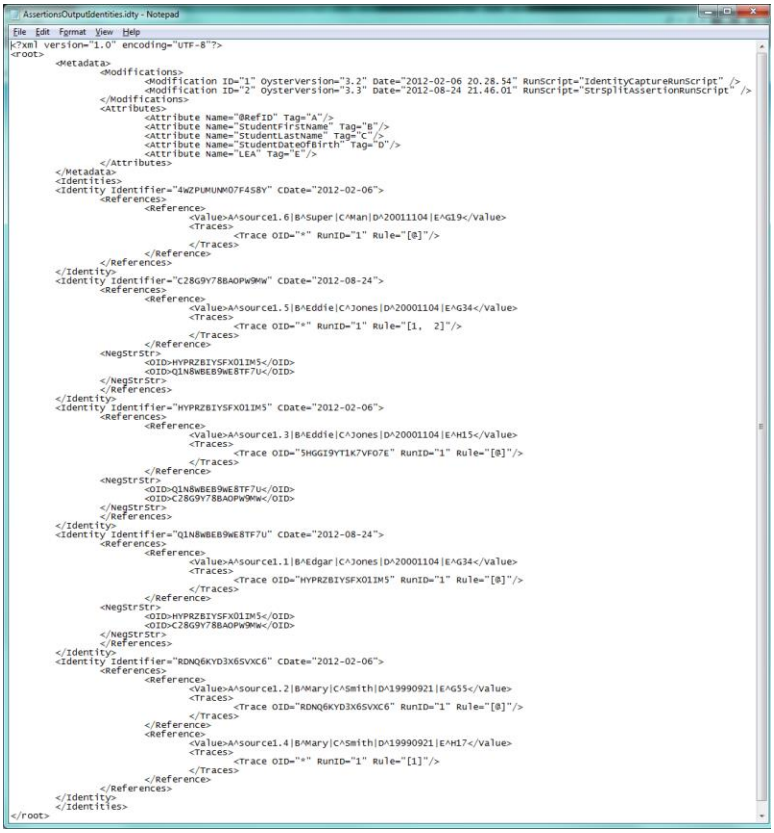


Figure 94: Identity file created for Identity Build from SplitStr Assertions

Note that the split identities were assigned a NegStrStr value which keeps these references from ever matching on following runs.

As with the previous examples, this sample run was done using a delimited text file. Examples of how to connect to a Fixed Width text file, a Microsoft Access DB, MySQL, and Microsoft SQLServer can be seen in the OYSTER Reference Guide.

Identity Resolution

Figure 95 illustrates the dataflow for an OYSTER run configured to perform identity resolution.

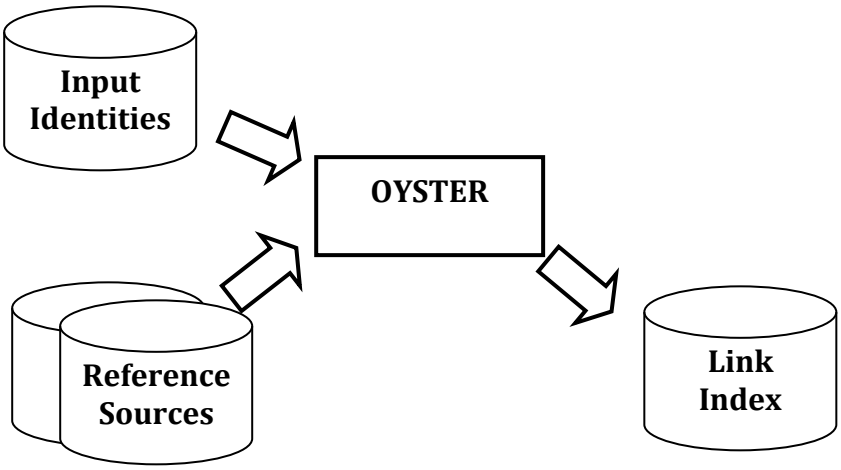


Figure 95: Identity Resolution dataflow

Configuration

When OYSTER is configured to perform identity resolution there should be no output identity file specified in the OysterRunScript but a link file and input identity file must be specified. The configuration is shown in Figure 96.

```
<!-- Identity Input Selection -->
<IdentityInput Type="TextFile">Z:\Oyster\Input.idty</IdentityInput>

<!-- Identity Output Selection -->
<IdentityOutput Type="None"/>

<!-- Link Output Selection -->
<LinkOutput Type="TextFile">Z:\Oyster\Index.link</LinkOutput>
```

Figure 96: Defined Input and Output for Identity Resolution

In this configuration the process starts with a fixed set of identities, the Input.idty file defined in Figure 96. The reference sources are resolved against the input identities. References that do not resolve to any of the input identities are written to the link index, "Index.link" above, with the same special link value that indicates "no resolution".

You can use the AssertionsOutputIdentities.idty file created by doing the identity build from RefToRef Assertions earlier as the input identities for the Identity Resolution configuration. This allows for OYSTER to create matches based on user knowledge about specific entities.

Lastly, the RunMode should be set to "IdentityResolution" as shown in Figure 97.

```
<RunMode>IdentityResolution</RunMode>
```

Figure 97: RunMode Set to "IdentityResolution" in OysterRunScript

Example

This example will use the test source file named 'IdentityResolutionTest.txt', shown in Figure 98. This data consists of six references. Each reference is defined by a RefID, FirstName, LastName, DOB, and SchCode attributes.

This run also uses the identities created by the previous assertions run that are stored in the AssertionsOutputIdentities.idty file. The identity resolution run requires a set of identity inputs so that it can identify identities in the input source that resolve to one of the input identities.

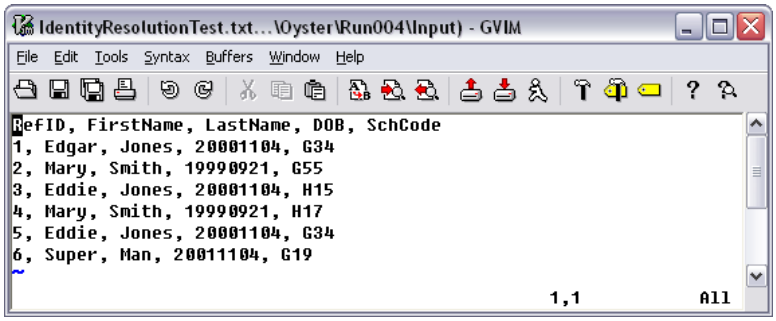


Figure 98: Source Input for Identity Resolution Example

After analyzing the source data the source descriptor file can be created.

Using this source file and these two rules, the source descriptor file, named 'IdentityResolutionSourceDescriptor.xml' can be created. This file is shown in Figure 99.

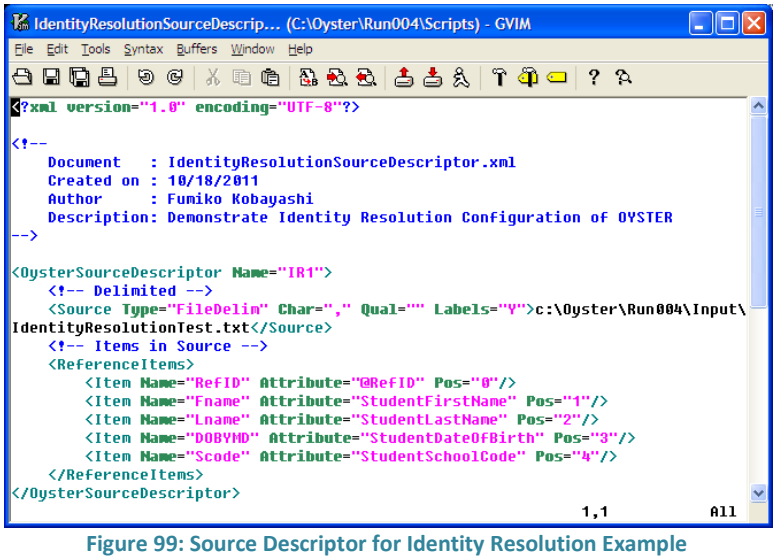


Figure 99: Source Descriptor for Identity Resolution Example

Again, by following the same process as was performed when setting up the merge-purge example, once the source descriptor is defined the source attributes file must also be defined. This file is stored in the Source folder along with the Source Descriptor file. The attributes file is used to define the attributes in the source along with the algorithm used to compare the attributes and the matching (Identity) rules used when ER is performed. For this example run two identity rules will be used. The first rule says that the reference will be considered equivalent if the FirstName, LastName, and DOB attributes match. The second rules states that the references are equivalent if the LastName, DOB, and SchoolCode match. The source attribute file is named 'IdentityResolutionAttributes.xml' and is depicted in Figure 100.

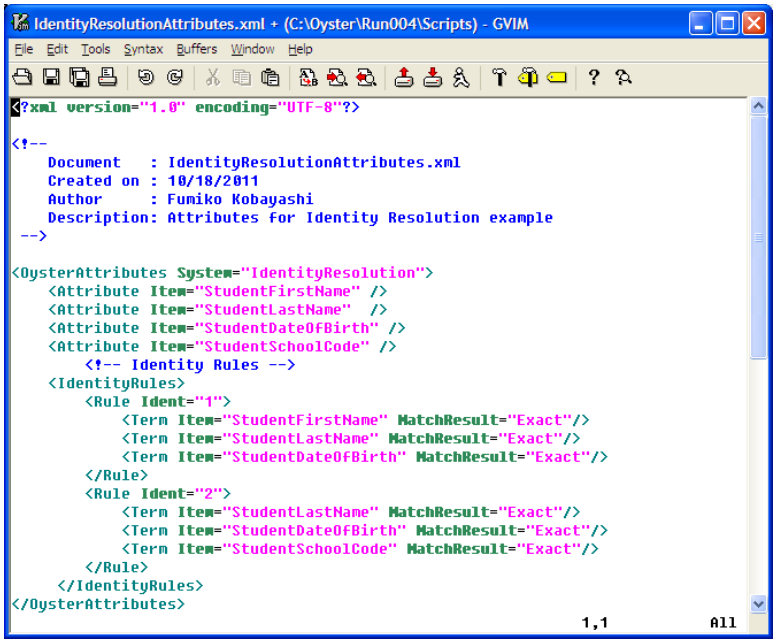


Figure 100: Attributes file for Identity Resolution Example

As with the other examples, the last file that needs to be created is the RunScript for this example. For this identity resolution example, no output identity file should be specified in the Run Script but both the input identity file and the link files should be specified. The Run Script should again be stored in the root OYSTER folder as this is where the OYSTER program is expecting the file to reside. The file for this example is named 'IdentityResolutionRunScript.xml' and is shown in Figure 101.

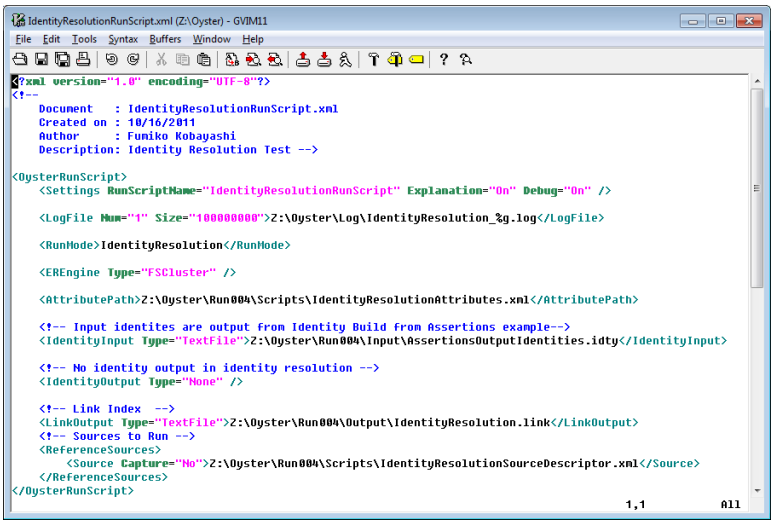


Figure 101: Run Script for Identity Resolution Example

Now that all the scripts for the Identity Resolution example have been created we can run OYSTER. This process is depicted in Figure 37, Figure 38, and Figure 39 and described in their surrounding text in the Example section.

Once the run is complete the output for the run will be written to the command box by OYSTER. This output is shown in Figure 102 and Figure 103.

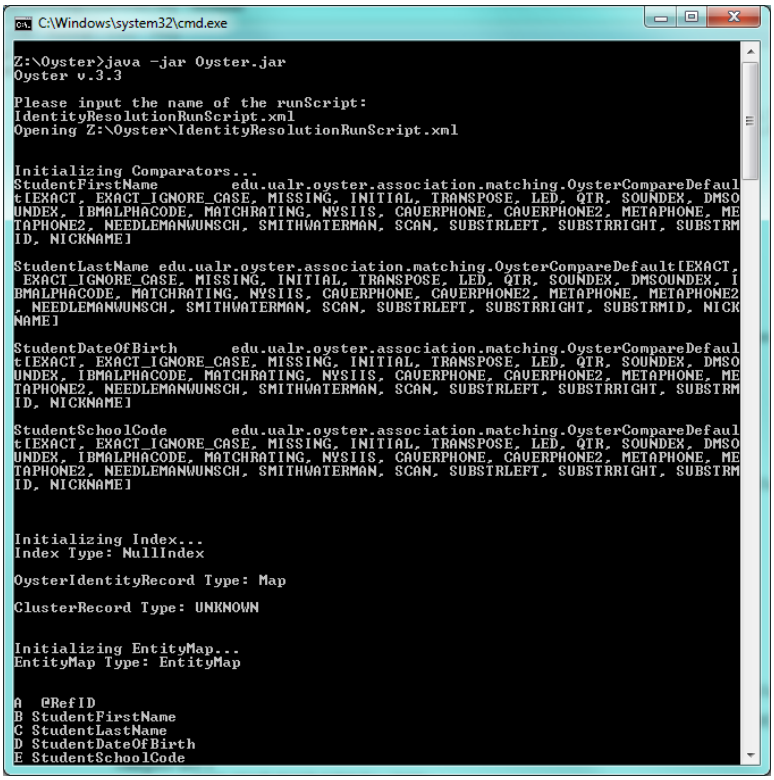


Figure 102: Screen Output Created by Identity Resolution Example Run - 1

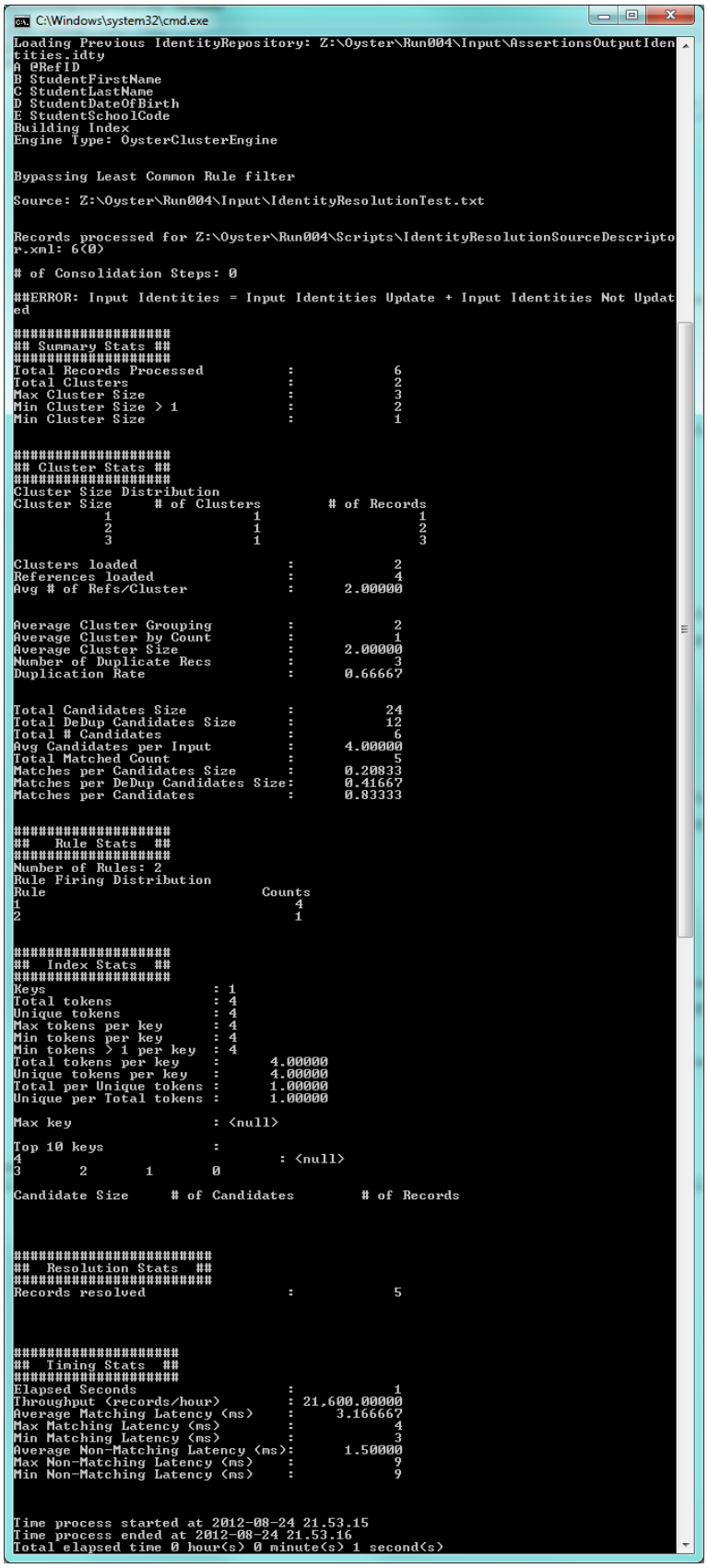


Figure 103: Screen Output Created by Identity Resolution Example Run - 2

The output in Figure 103 may seem a little confusing in that it states that OYSTER was able to process 6 records but only found 2 entities (Clusters). This is due to the way Identity Resolution works; it only creates links for records that resolve to an existing identity in the identity input file. If a reference in the source input does not resolve it is not counted.

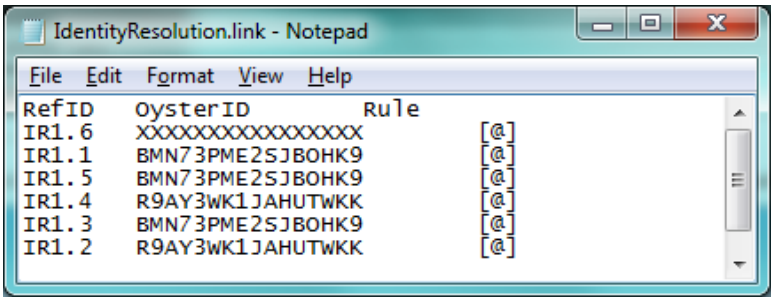


Figure 104: Link File Created by Identity Resolution Example Run

As you can see in the Link file generated by this identity resolution example, shown in Figure 104, OYSTER was able to create links for 5 of the 6 references in the source data. The sixth reference was assigned an OYSTER ID of 'XXXXXXXXXXXXXXXX' which means that reference could not be linked to any existing identity in the identity input file for this run. This is the type of data you are looking for when performing Identity Resolution.

Figures