



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

OYSTER v3.3 Reference Guide

Document Version: 1.11, Date: 18 March 2013

Copyright © 2012 ERIQ
University of Arkansas at Little Rock

Edited by:

Fumiko Kobayashi

Authors:

Fumiko Kobayashi and John Talburt

Revision History

Version	Date	Prepared By	Position	Reason for Update
1.1	11/17/2010	Fumiko Kobayashi	RA	Added Output section Modified the section heading to contain defining numbers Removed redundant information Added Introduction
1.2	12/01/10	Fumiko Kobayashi	RA	Modified to reflect new XML requirements for OYSTER v2.4 Expanded Log section of the Output section
1.3	2/4/2011	Fumiko Kobayashi	RA	Modified document to reflect new changes to Oyster V2.6
1.4	2/27/2011 3/1/2011	Fumiko Kobayashi	RA	Added Comparator section Modified document to reflect new changes to OYSTER V3.0
1.5	5/2/2011	Fumiko Kobayashi	RA	Modified document to reflect new changes to OYSTER V3.1
1.6	6/14/2011	Fumiko Kobayashi	RA	Updated Output section and RunScript section
1.7	6/22/2011	Fumiko Kobayashi	RA	Removed References to RunScriptName
1.8	10/28/2011 02/01/2012	Fumiko Kobayashi	RA	Modified document to reflect new changes to OYSTER V3.2
1.9	02/24/2012 04/18/2012	Fumiko Kobayashi	RA	Added clarification regarding discrepancies in Change Report between engines.
1.10	08/24/2012	Fumiko Kobayashi	RA	Added UDI, CAC, KILL, and new comparators.
1.11	12/02/2012 03/18/2013	Fumiko Kobayashi	RA	Updated UDI and Log File sections Update to reflect OYSTER v3.3 change

Table of Contents

I. Introduction.....	6
II. Scripts.....	7
1. OysterRunScript.....	7
1.1 XML Declaration and Comments.....	7
1.2 <OysterRunScript> Tag.....	8
1.3 <Settings> Tag.....	8
1.4 <LogFile> Tag.....	11
1.5 <RunMode> Tag.....	12
1.6 <EREngine> Tag.....	12
1.7 <AttributePath> Tag.....	13
1.8 <IdentityInput> Tag.....	13
1.8.1 Type="None"	14
1.8.2 Type="TextFile"	14
1.8.3 Type="Database"	14
1.9 <IdentityOutput>Tag.....	15
1.9.1 Type="None"	15
1.9.2 Type="TextFile"	15
1.9.3 Type="Database"	15
1.10 <LinkOutput> Tag.....	16
1.10.1 Type="TextFile"	16
1.10.2 Type="Database"	17
1.11 <AssertionInput> Tag.....	17
1.12 <ReferenceSources> Tag	18
1.13 <Source> Tag.....	18
2. OysterAttributes	18
2.1 XML Declaration and Comments.....	19
2.2 <OysterAttributes> Tag	19

2.3 <Attribute> Tag.....	20
2.4 <IdentityRules> Tag.....	20
2.5 <Rule> Tag	20
2.6 <Term> Tag	21
2.7 <Indices> Tag	26
2.8 <Index> Tag.....	26
2.9 <Segment> Tag	26
2.10 User Defined Index (UDI) Example.....	27
2.11 Scan Hash Function.....	27
2.12 Cross-Attribute Compare (CAC)	29
3. OysterSourceDescriptor	31
3.1 XML Declaration and Comments.....	31
3.2 <OysterSourceDescriptor> Tag.....	31
3.3 <Source> Tag.....	32
3.3.1 Type="FileDelim"	32
3.3.2 Type="FileFixed"	33
3.3.3 Type="Database"	33
3.4 <ReferenceItems> Tag.....	34
3.5 <Item> Tag	35
3.5.1 Delimited Text Items	35
3.5.2 Fixed Width Items	38
3.5.3 Database Items.....	39
III. Attribute-Based vs. Record-Based Matching for Identities.....	40
1. Example.....	40
1.1 Attribute-Based.....	40
1.2 Record-Based.....	41
IV. Reference Sources	44
1. Example from Text Files	44
1.1 From Delimited Files.....	44
1.2 From Fixed Width Files.....	45

2. Example from Database Tables.....	46
2.1 From Open Database Connectivity (ODBC)	46
2.1.1 Creating ODBC Connections	46
2.1.2 Example from ODBC	48
2.2 From MySQL.....	49
2.3 From Microsoft SQLServer	50
V. Outputs.....	51
1. Identity Document	51
1.1 <root> Tag.....	51
1.2 <Metadata> Tag	51
1.2.1 <Modifications> Tag	52
1.2.1.1 <Modification> Tag.....	52
1.2.2 <Attributes> Tag	53
1.2.2.1 <Attribute> Tag.....	53
1.3 <Identities> Tag.....	53
1.3.1 <Identity> Tag.....	54
1.3.2 <References> Tag	54
1.3.2.1 <Reference> Tag.....	55
1.3.2.1.1 <Value> Tag	55
1.4 Example.....	55
2. Link Index.....	56
3. Identity Change Report	57
3.1 Sections	57
3.1.1 Metadata Section.....	58
3.1.2 Summary	58
3.1.3 Change Detail.....	58
3.4 ChangeReportDetail="Off"	59
3.3 ChangeReportDetail="On"	60
4. Identity Merge Map.....	61
5. Log File.....	61

5.1 Summary Stats.....	62
5.2 Cluster Stats.....	62
5.3 Rule Stats.....	65
5.4 Index Stats.....	65
5.5 Resolution Stats	68
5.6 Timing Stats.....	68
6. Extended Log File	70
VI. Other OYSTER Functionality	71
1. KILL Thread	71
2. 90% memory Warning	71
VII. OYSTER User-Defined Inverted Index	72
1. Inverted Index.....	72
2. Match Key	72
2.1 Match Key Example 1.....	73
3. Index Operation.....	73
4. Alignment of Index with Match Rules.....	75
5. Index Recall and Precision	75
5.1 Alignment Scenarios	76
5.2 Index Fewer Attributes Strategy	77
5.3 Multiple Index Strategy.....	78
6. The Alignment Process.....	79
6.1 Rule Analysis.....	79
6.2 Index Design.....	80
6.2.1 Match Key Comparators versus Similarity Comparators.....	82
6.2.2 Balancing Alignment with Reduction	83
6.3 Alignment Validation	85
VIII. Error Messages	86
Common Errors.....	86
Figures.....	87

I. Introduction

This OYSTER reference guide is intended for use by OYSTER users who already have an understanding of the basics of both OYSTER and Entity Resolution as introduced in the “Introduction to Entity Resolution with OYSTER” document bundled with OYSTER. This document is divided into seven sections. The first section is the introduction which you are reading now. The second section is designed to answer configuration questions regarding available options in the OYSTER XML files. The third section provides an explanation and example of Record-based and Attribute based matching. The fourth section provides examples of how to establish connections to different types of source inputs. The fifth section details how to read and understand the various output files that can be generated by an OYSTER run. The sixth section describes other features of OYSTER that fall outside the realm of any other section. The last section provides a list of common errors that can be encountered during the configuration of OYSTER and how to resolve the errors.

II. Scripts

This reference contains an exhaustive explanation of all components that make up the XML files required by OYSTER. Each file is broken down into sections by their tags and each tag section explains the possible attributes and values associated with each tag,

1. OysterRunScript

This section describes all the different tags and attributes available in the OysterRunScript.

This table outlines the available tags and the RunModes they are required for:

Run Script Element	MergePurge	IdentityCapture	IdentityResolution	IdentityUpdate	AssertRefToRef	AssertRefToStr	AssertStrToStr	AssertSplitStr
<Settings>	Req	Req	Req	Req	Req	Req	Req	Req
<LogFile>	-	-	-	-	-	-	-	-
<RunMode>	Req	Req	Req	Req	Req	Req	Req	Req
<EREngine>	-	-	-	-	-	-	-	-
<AttributePath>	Req	Req	Req	Req	Req	Req	Req	Req
<IdentityInput>	None	None	Req	Req	None	Req	Req	Req
<IdentityOutput>	None	Req	None	Req	Req	Req	Req	Req
<LinkOutput>	Req	Req	Req	Req	Req	Req	None	None
<AssertionInput>	None	None	None	None	Req	Req	Req	Req
<ReferenceSources>	Req	Req	Req	Req	None	None	None	None
<Source>	Req	Req	Req	Req	None	None	None	None

Please Note: the “-” indicates these tags are optional for the corresponding RunMode. “Req” specifies a tag must be defined for the corresponding RunMode of it can cause an error. “None” means the tag must not be specified for the corresponding RunMode of it may cause an error.

1.1 XML Declaration and Comments

All the OYSTER XML documents begin with an XML declaration and a comment section. The XML declaration is required and must be included exactly as follows:

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

The comments section is optional but should be included in the file. The following is the suggested format of the comments section:

```
<!--  
    Document    : RunScript.xml  
    Created on  :  
    Author      :  
    Description:  
        Purpose of the document follows.  
-->
```

1.2 <OysterRunScript> Tag

```
<OysterRunScript>  
    .  
    .  
    <child_elements>...</child_elements>  
    .  
    .  
</OysterRunScript>
```

The `OysterRunScript` tag is the root element and thus the start and end tag encloses all other elements in the document. All other elements are considered child elements to this root element. It has no attributes.

The `OysterRunScript` tag can enclose up to 10 of the following tags:

- <Settings>
- <LogFile>
- <RunMode>
- <EREngine>
- <AttributePath>
- <IdentityInput>
- <IdentityOutput>
- <LinkOutput>
- <ReferenceSources>
- <Source>
- <AssertionInput>

1.3 <Settings> Tag

```
<Settings RunScriptName="Name" Explanation="Off" Debug="On" SS="No"  
ChangeReportDetail="No" Trace="On" />
```

The `Settings` tag directs what information is written to the `Oyster.log` file that is created during each OYSTER run. It also directs what additional output files are generated during

the run and the level of detail written to the files. Its one other function is to validate the RunScript by checking the name of the RunScript to make sure it valid.

The Setting tag has six attributes:

- RunScriptName
 - Assigned value must match the name of the RunScript .xml file without the .xml file extension.
 - Example:
 - Filename is "OysterRunScript.xml"
 - RunScriptName should be assigned the value of "OysterRunScript"
 - Function:
 - The RunScriptName attribute is simply a check that makes sure the user is not copying a RunScript without checking to make sure it does what they intend to do. It does this by forcing the user to open the RunScript to edit at least one value.
- **Explanation**
 - Acceptable values:
 - "On"
 - "Off"
 - Function:
 - The Explanation flag turns on the log file and writes out an explanation of what is happening when OYSTER runs.
- **Debug**
 - Acceptable values
 - "On"
 - "Off"
 - Function:
 - The Debug flag turns on the "IdentityCaptureOutput.idty.emap" and "IdentityCaptureOutput.idty.indx" files and it adds more detail to the log file.
 - If the Explanation flag is off and the Debug is on the log file still gets written out due to the way Java handles logging.

The following table provides an explanation of the different combinations of the Explanation and Debug attributes.

Explanation	Debug	Description
Off	Off	There is no explanation and no debug. Only significant errors are output to the log file. Best performance in speed.
On	Off	There is an explanation but no debug. Explanation and significant errors are output to the log file. Still good performance.

Off	On	There is an explanation and a debug as well significant errors are output to the log file. Performance decreased due to increased logging.
On	On	There is an explanation and a enhanced debug as well significant errors are output to the log file. Poor performance should only be used on small files for testing purposes.

- SS
 - Acceptable values:
 - "On"
 - "Off"
 - Function:
 - The SS attribute turns on the System Statistics for the run and generates statistics before and after parts of the run such as before and after reading in a new source file. These statistics include:
 - Heap Size – Total Memory
 - Max Heap Size – Max Memory allowed to be used
 - Used Heap Size – Memory being used
 - Free Heap Size – memory not being used.
 - Defaults to "Off" if not included in the RunScript
- ChangeReportDetail
 - Acceptable values:
 - "Yes"
 - "No"
 - Function:
 - The ChangeReportDetail attribute dictates the level of detail that will be written to the Change Report output file.
 - When it is set to "Yes", OYSTER will produce detailed change report.
 - When it is set to "No", OYSTER will produce a brief change report that does not contain a comprehensive list of OYSTER IDs.
 - Causes OYSTER to generate two reports stored in same directory as the output .idty file:
 - "Identity Change Report"
 - "Identity Merge Map"
 - Defaults to "No" if not included in the RunScript
- Trace
 - Acceptable values:
 - "On"
 - "Off"
 - Function:
 - The Trace attribute turns off and on the Trace functionality of the OYSTER .idty file.
 - Trace provides the user the capability to trace the progress of a reference from the current run all the way back to its origin (when enabled since the first run). 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.

- Defaults to "Off" if not included in the RunScript.
- If Trace is on, the performance of OYSTER is impacted.

The `Settings` tag is not optional meaning OYSTER will not run successfully if the run script does not contain this element. All attributes can be omitted from the `Settings` tag except for the `RunScriptName` attribute. OYSTER will default all missing attribute values to "Off" or "No".

NOTE: `Explanation` and `Debug` should only be set to "On" when testing or trying to resolve an issue with a Run. By setting these attributes to "On" there can be a significant performance impact for a run on the scale of the run taking hours instead of minutes.

1.4 <LogFile> Tag

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

The `LogFile` tag directs where the log file for the run is stored, the number of log files that can be created before OYSTER round-robins and overwrites the first file, and the allowed size of the log file. The start and end tag enclose character data that represents the absolute path to the generated log files.

The `LogFile` tag has two attributes:

- `Num`
 - Accepts any integer as a value.
 - Function:
 - Defines the number of log files that can be created for the run.
 - Defaults to 10
- `Size`
 - Accepts any integer as a value
 - The integer represents the max size of the file in bits. i.e. 100000000 bits \approx 11.9 megabytes
 - Function:
 - Defines the max size of each generated log file.
 - Defaults to 100000bits

The Absolute path to the log files must contain the `_%g` as this tells OYSTER where to include the numeric increment in the name of the log files.

1.5 <RunMode> Tag

```
<RunMode>Mode</RunMode>
```

The `RunMode` tag is used to define which mode the OYSTER run is configured to run in. This dictates what tags should be included in the `RunScript` for it to validate correctly.

The `RunMode` tag has no attributes. The start and end tags enclose character data that represents a valid `RunMode`. Valid `RunModes` are:

- MergePurge
- IdentityCapture
- IdentityResolution
- IdentityUpdate
- AssertRefToRef
- AssertRefToStr
- AssertStrToStr
- AssertSplitStr

1.6 <REngine> Tag

```
<REngine Type="RSwooshStandard" />
```

The `REngine` tag is used to define which entity resolution engine should be used for the defined OYSTER run.

The `REngine` tag has one attributes:

- Type
 - Acceptable values:
 - "RSwooshStandard"
 - Tells OYSTER to use the original R-Swoosh engine to perform ER. Faster than the `RSwooshEnhanced` engine when a low percentage of duplicate records are expected to be processed.
 - Uses Attribute-Based matching
 - "RSwooshEnhanced"
 - Tells OYSTER to use an enhanced R-Swoosh engine to perform ER. Faster than the `RSwooshStandard` engine when a high percentage of duplicate records are expected to be processed across multiple sources.
 - Uses Attribute-Based matching
 - "FScluster"

- Tells OYSTER to use the Fellegi-Sunter record-linking model to perform ER. This is not a modification of the R-Swoosh engine. FScluster is a completely different method of performing ER and may produce different results when compared with the results of RSwooshStandard or RSwooshEnhanced.
- Uses Record-Based matching.

The EREngine tag is completely optional. When left out of the RunScript, OYSTER defaults to RSwooshStandard as the EREngine.

1.7 <AttributePath> Tag

<AttributePath>Absolute Path to Attributes.xml file</AttributePath>

The AttributePath tag has no attributes. The start and end tag enclose character data that represents the absolute path to the OYSTER Attributes script, described in the OysterAttributes section.

The AttributePath tag and the absolute path to the attributes file are required or the OYSTER run will produce the error: “##ERROR: Reference Items and Attributes do not match. ”

1.8 <IdentityInput> Tag

The IdentityInput tag specifies where the Identities to be used as input are stored. These identities are stored in an XML format generated by a previous OYSTER run. The identities XML can be stored in and read from a text file or a database table.

The IdentityInput tag must have the Type attribute that defines the three possible formats of this tag.

- Type attribute
 - This attribute is required. If it is omitted OYSTER will stop running without producing any user facing error.
 - Acceptable Values:
 - “None”
 - “TextFile”
 - “Database”

1.8.1 Type="None"

```
<IdentityInput Type="None" />
```

When Type is assigned the value of "None" IdentityInput has only the Type attribute.

The IdentityInput statement cannot be left out of the run script or OYSTER will stop running once it reaches this section of the run script. To prevent this OYSTER must be notified that there is no identity input by assigning a Type value of "None".

1.8.2 Type="TextFile"

```
<IdentityInput Type="TextFile">Absolute Path</IdentityInput>
```

When Type is assigned the value of "TextFile" the IdentityInput tag has only the Type attribute. The start and end tag enclose character data that represents the absolute path to the text file containing the XML formatted identities.

1.8.3 Type="Database"

```
<IdentityInput Type="Database" Server="123.123.123.123" Port="1433"  
SID="PROD" UserID="UserName" Passwd="Password">Table  
Name</IdentityInput>
```

When Type is assigned the value of "Database" the IdentityInput tag may have up to five additional attributes.

- Type – Must be assigned the value of "Database"
- Server – Must be assigned the address of the database server to be used.
 - DNS Name
 - IP address
- Port – Must be assigned the port number on which the database server will respond to requests.
- SID – Must be assigned the name of the database in which the XML identities are stored on the server.
- UserID – Must be assigned the name of the user with access to the required data.
- Passwd – Must be assigned the password associated to the user given as the values of the UserID attribute.

The start and end tag enclose character data that represents the table name in which the identity input can be located.

Please note that when connecting to Microsoft SQLServer or MySQL, the table name must be fully qualified or OYSTER will produce a connection error. An example of a fully

qualified table name is "DatabaseName.dbo.TableName" in SQL Server or "DatabaseName.TableName" in MySQL.

1.9 <IdentityOutput>Tag

The IdentityOutput tag specifies where the Identities defined during the OYSTER run will be stored.

The IdentityOutput tag must have the Type attribute that defines the three possible formats.

- Type attribute
 - Acceptable Values:
 - "None"
 - "TextFile"
 - "Database"

1.9.1 Type="None"

```
<IdentityOutput Type="None" />
```

When Type is assigned the value of "None" IdentityOutput has only the Type attribute.

The IdentityOutput statement cannot be left out of the run script or OYSTER will stop running once it reaches this section of the run script. To prevent this OYSTER must be notified that there is no identity output by assigning a Type value of "None".

1.9.2 Type="TextFile"

```
<IdentityOutput Type="TextFile">Absolute Path</IdentityOutput>
```

When Type is assigned the value of "TextFile" the IdentityOutput tag has only the Type attribute. The start and end tag enclose character data that represents the absolute path to the text file that will be created to contain the XML formatted identities for the OYSTER run.

1.9.3 Type="Database"

```
<IdentityOutput Type="Database" Server="123.123.123.123" Port="1433"  
SID="PROD" UserID="UserName" Passwd="Password">Table  
Name</IdentityOutput>
```

NOTE: as of OYSTER v3.3 the Output to a DBMS is not implemented but will be supported in the future with the following implementation.

When `Type` is assigned the value of "Database" the `IdentityOutput` tag may have up to five additional attributes .

- `Type` – Must be assigned the value of "Database"
- `Server` – Must be assigned the address of the database server to be used.
 - DNS Name
 - IP address
- `Port` – Must be assigned the port number on which the database server will respond to requests.
- `SID` – Must be assigned the name of the database in which the XML identities are to be stored on the server.
- `UserID` – Must be assigned the name of the user with access to write to the desired table.
- `Passwd` – Must be assigned the password associated to the user given as the values of the `UserID` attribute.

The start and end tag enclose character data that represents the table name in which the identity output will be written.

Please note that when connecting to SQLServer 2005 or MySQL, the table name must be fully qualified or OYSTER will produce a connection error. An example of a fully qualified table name is "DatabaseName.dbo.TableName" in SQL Server or "DatabaseName.TableName" in MySQL.

1.10 <LinkOutput> Tag

The `LinkOutput` tag specifies where the link index created during the OYSTER run will be stored. A link output is required for every OYSTER run.

The `LinkOutput` tag must have the `Type` attribute that defines the two possible formats.

- `Type` attribute
 - Acceptable Values
 - "TextFile"
 - "Database"

1.10.1 Type="TextFile"

```
<LinkOutput Type="TextFile">Absolute Path</LinkOutput>
```

When `Type` is assigned the value of "TextFile" the `LinkOutput` tag has only the `Type` attribute. The start and end tag enclose character data that represents the absolute path to the text file that will be created to contain the link index created for the OYSTER run.

1.10.2 `Type="Database"`

```
<LinkOutput Type="Database" Server="123.123.123.123" Port="1433"
SID="PROD" UserID="UserName" Passwd="Password">Table Name</LinkOutput>
```

NOTE: As of OYSTER v3.3 the output to a DBMS is not implemented but will be supported in the future with the following implementation.

When `Type` is assigned the value of "Database" the `LinkOutput` tag may have up to five additional attributes.

- `Type` – Must be assigned the value of "Database"
- `Server` – Must be assigned the address of the database server to be used.
 - DNS Name
 - IP address
- `Port` – Must be assigned the port number on which the database server will respond to requests.
- `SID` – Must be assigned the name of the database in which the link index will be stored on the server.
- `UserID` – Must be assigned the name of the user with access to write to the desired table.
- `Passwd` – Must be assigned the password associated to the user given as the values of the `UserID` attribute.

The start and end tag enclose character data that represents the table name in which the link index will be written.

Please note that when connecting to SQLServer 2005 or MySQL, the table name must be fully qualified or OYSTER will produce a connection error. An example of a fully qualified table name is "DatabaseName.dbo.TableName" in SQL Server or "DatabaseName.TableName" in MySQL.

1.11 `<AssertionInput>` Tag

```
<AssertionInput>Absolute Path to Source Descriptor XML</AssertionInput>
```

The `AssertionInput` tag has no attributes. The start and end tag enclose character data that represents the absolute path to the OYSTER SourceDescriptor XML script, described in the `OysterSourceDescriptor` section.

This tag is only used when running OYSTER in one of the four assertion modes.

NOTE: Assertion runs can only accept a single source unlike the other OYSTER `RunModes`.

1.12 `<ReferenceSources>` Tag

```
<ReferenceSources>
  .
  <Source>...</Source>
  .
</ReferenceSources>
```

The `ReferenceSources` tag has no attributes. The start and end tag enclose a series of `Source` tags, discussed in the next section, that are children of the `ReferenceSources` tag.

This tag is only used when running OYSTER in one of the four none-assertion modes.

1.13 `<Source>` Tag

```
<Source>Absolute Path to Source Descriptor XML</Source>
```

The `Source` tag contains no attributes.

The start and end tag enclose character data that represents the absolute path to the OYSTER SourceDescriptor XML script, described in the `OysterSourceDescriptor` section.

Multiple `Source` tags can be included in the `RunScript`. There should be one `Source` tag for every `SourceDescriptor` defined for the OYSTER run.

This tag is only used when running OYSTER in one of the four none-assertion modes.

2. `OysterAttributes`

This section describes all the different tags and attributes available in the `OysterAttributes` file.

2.1 XML Declaration and Comments

All the OYSTER XML documents begin with an XML declaration and a comment section. The XML declaration is required and must be included exactly as follows:

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

The comments section is optional but should be included in the file. The following is the suggested format of the comments section:

```
<!--  
    Document    : Attributes.xml  
    Created on  :  
    Author      :  
    Description:  
        Purpose of the document follows.  
-->
```

2.2 <OysterAttributes> Tag

```
<OysterAttributes System="System_Name">  
    .  
    <child_elements>...</child_elements>  
    .  
</OysterAttributes>
```

The `OysterAttributes` tag has a single attribute.

- `System` – Assigned the name of the system for which the OYSTER run was configured. This value is created and defined by the user.
 - Has no impact of the outcome of the OYSTER run.
 - Possible to completely remove this attribute from the `OysterAttributes` file with no impact to the run

`OysterAttributes` is the root element of this file and thus the start and end tag encloses all other elements in the document. All other elements are considered child elements to this root element. The `OysterAttributes` tag can enclose seven tags:

- `<Attribute>`
- `<IdentityRules>`
- `<Rule>`
- `<Term>`
- `<Indices>`
- `<Index>`
- `<Segment>`

2.3 <Attribute> Tag

```
<Attribute Item="Name" Algo="matchAlgoName" />
```

The `Attribute` tag has two attributes.

- `Item`
 - Must be assigned a name of an attribute that is used to define a reference in the source file.
 - The value must match the value assigned to `Attribute` in the `<Item>` tag of the `OysterSourceDescriptor` file for the same attribute or OYSTER will produce the error: “##ERROR: Reference Items and Attributes do not match.”
- `Algo`
 - Assign a pre-defined matching algorithm that is to be used for comparing attribute values 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.
 - The default matching algorithm supports many different comparators which are defined in detail in the `<Term>` Tag section of this document.

There must be one `Attribute` statement defined for each distinct `Attribute` value defined in the `OysterSourceDescriptor` file, discussed in the `OysterSourceDescriptor` section excluding any attribute assigned an OYSTER reserved word such as `@RefID`.

2.4 <IdentityRules> Tag

```
<IdentityRules>
  .
  <child_elements>...</child_elements>
  .
</IdentityRules>
```

The `IdentityRules` tag has no attributes. The start and end tag enclose the `Rule` and `Term` tags described in the next two section. The `Rule` elements are considered child elements to this element and the `Term` elements are considered grandchildren of this element.

2.5 <Rule> Tag

```
<Rule Ident="rule_name">
  .
  .
```

```

        <child_elements>...</child_elements>
        .
        .
    </IdentityRules>

```

The `Rule` tag has one attribute.

- `Ident` - assigned value represents a unique name for the matching rule.

The start and end tag enclose the `Term` tags described in the next section. The `Term` elements are considered child elements to this element.

Any number of matching rules can be defined for any given source input and are determined by the user. Each rule must be provided a unique `Ident` value.

2.6 `<Term>` Tag

```

<Term Item="name" MatchResult="code"/>

```

The `Term` tag has up to three attributes.

- `Item` - Assigned value must correspond to the value of one of the `Item` attributes defined in the `Attributes` section.
- `MatchResult` - Match codes used to determine equivalence for the `Item`
 - Available default True/False match codes:
 - `EXACT` - used if the two values being compared must be EXACT matches. This function is case sensitive.
 - `EXACT_IGNORE_CASE` - Functions exactly like `Exact` except this version is not case sensitive. Example: "Sam" matches "SAM"
 - `TRANSDPOSE` - used to allow for a difference between a source and a target of 2 adjacent characters in one of the values being in reverse order. This is case sensitive. Example: Value1 = "12345" and Value2 = "12435". In this case OYSTER's matching algorithm would return `TRANSDPOSE`.
 - `INITIAL` - used to allow two values being compared to be equivalent if one value is an initial and it matches the first character of the other value. This function is case sensitive. Example: Value1 = "J" and Value2 = "John". In this case OYSTERs matching algorithm would return `INITIAL`.
 - `NICKNAME` - used to allow for two values to be equivalent if both values can be found in and correspond to one another in the lookup table (see the `alias.dat` file). This function is not case sensitive. Example: Value1 = "Robert" and Value2 = "Bob". In this case OYSTERs

matching algorithm would check the lookup table, find the values are valid, and return `NICKNAME`.

- `SOUNDEX` – used to allow a match if two values produce the same hashing code according to the Soundex algorithm (a phonetic algorithm for indexing names by sound, as pronounced in English). This function is not case sensitive. Example: Using this algorithm, both "Robert" and "Rupert" return the same hash and `OYSTER` would return `SOUNDEX` when these 2 values are compared.
- `DMSOUNDEX` – used to allow a match if two values produce the same hashing code according to the Daitch-Mokotoff Soundex algorithm (a refinement of the Russell and American Soundex algorithms designed to allow greater accuracy in matching of Slavic and Yiddish surnames with similar pronunciation but differences in spelling). This function is not case sensitive. Example: Using this algorithm, both "Moskowitz" and "Moskovitz" return the same hash and `OYSTER` will return `DMSOUNDEX` when these 2 values are compared. Note that these 2 values actually produce different hash results using the `SOUNDEX` algorithm and would not match.
- `IBMALPHACODE` – used to allow a match if two values produce the same hashing code according to the IBM Alpha Code algorithm. This function is not case sensitive. Example: Using this algorithm, both "Rodgers" and "Rogers" produce the same hash and `OYSTER` will return `IBMALPHACODE` when these 2 values are compared.
- `NYSIIS` (New York State Identification and Intelligence System) – used to allow a match if two values produce the same hashing code according to the `NYSIIS` Code algorithm (is a phonetic algorithm devised in 1970 as part of the New York State Identification and Intelligence System and features an accuracy increase of 2.7% over the traditional Soundex algorithm). This function is not case sensitive. Example: Using this algorithm, both "McKee" and "Mackie" produce the same hash and `OYSTER` will return `NYSIIS` when these 2 values are compared.
- `MATCHRATING` – used to allow a match if two values produce a similar hashing code that is then compared by a similarity algorithm according to the Western Airlines Match Rating Approach algorithm. This function is not case sensitive. Example: "Byrne" produces `BYRN` and "Boern" produces `BRN` which both produce a similarity rating of 5 with a minimum rating of 4 meaning that `OYSTER` will find them to match and return `MATCHRATING`.
- `CAVERPHONE` – this is a phonetic matching hash algorithm similar to `SOUNDEX` but is optimized for accents present in the southern part of New Zealand. Example: Old and Hold both produce a hash value of `AT11111111` so are considered a match.
- `METAPHONE` – this is a phonetic matching hash algorithm similar to `SOUNDEX` that hashes words by their English pronunciation. Example:

Franky and Frankie both generate the hash FRNK so they are considered a match.

- Available default functionalized match codes:
 - LED (Levenshtein Edit Distance) – used if wanting to allow for matches based on a normalized edit distance score when comparing 2 attributes. The LED function is not case sensitive. Default threshold is 0.8 if LED match is used without a threshold assigned. The user defined threshold take this form: `"LED(threshold)"` where the threshold must be between 0 and 1.
 - QTR (Q-Gram Tetrahedral Ratio) – Default threshold is 0.25 if QTR match is used without a threshold assigned. QTR is not case sensitive and the user defined threshold take the form of: `"QTR(threshold)"` where the threshold must fall between 0 and 1.
 - SUBSTRLEFT(length) – used to allow a match on 2 values if the first x characters starting from the left are the same. This is not case sensitive. Example: `SubStrLeft(3)` makes "Samual" match "Sam".
 - SUBSTRRIGHT(length) – used to allow a match on 2 values if the last x characters starting from the right most character are the same. This is not case sensitive. Example: `SubStrRight(4)` makes "JeanAnne" match "Anne".
 - SUBSTRMID(start, length) – used to allow a match on 2 values if the middle characters starting from position x for a specified length are the same. This is not case sensitive. Example: `SubStrMid(2, 6)` makes "Krystal" match "Crystalline".
 - `SmithWaterman(Match, Mismatch, Gap, Threshold)` – used to find local sequence alignment. Match, Mismatch, and Gap can be integers or float values. The threshold must be a value between 0 and 1 where 1 indicates an exact match.
 - `SCAN(Direction, CharType, Length, Casing, Order)` – a new hash algorithm that was created for OYSTER and introduced in v3.3 along with the UDI. It examines the value string either from left-to-right or from right-to-left (`Direction`) searching for the type of characters specified (`CharType`). As characters are found they are extracted to form the hash value until the requested number of characters (`Length`) has been found, or until the end of the value string is encountered. If the number of characters found is less than the number requested, the hash value is padded with asterisk (*) to meet the Length request. The Casing and Order parameters designate operations to be performed on the hash value after the characters have been extracted. The Casing parameter allows the user to specify whether the letters in the hash value are to be converted to uppercase or left as is. The Order parameter allows the user to specify the reordering of the characters in the hash value.
 - Parameters
 - `Direction`

- Acceptable Values
 - LR – scan string left-to-right
 - RL – scan string right-to-left
 - CharType
 - Acceptable Values
 - ALL – extract all characters
 - NONBLANK – extract only non-blank characters
 - ALPHA – extract only letter and digit characters
 - LETTER – extract only letters of the English alphabet
 - DIGIT – extract only digits 0-9
 - Length
 - Acceptable Values
 - An integer between 1 and 30
 - Casing
 - Acceptable Values
 - ToUpper – all letters in the hash should be converted to upper casing.
 - KeepCase – all letters in the hash should keep original casing.
 - Order
 - Acceptable Values
 - SameOrder – keep original order in which the characters were extracted.
 - L2HKeepDup – reorder the hash characters from lowest to highest values. Keep any dup characters.
 - L2HDropDup – reorder the hash characters from lowest to highest values. Drop any duplicate characters from the hash.
- Examples: the following table shows some example uses of the SCAN algorithm.

Scan Configuration	String Value	Hash Value
Scan(LR, ALPHA, 8, ToUpper, SameOrder)	"123 N. Oak St, Apt #5"	"123NOAKS"
Scan(RL, ALPHA, 8, ToUpper, SameOrder)	"123 N. Oak St, Apt #5"	"AKSTAPT5"
Scan(LR, DIGIT, 6, KeepCase, SameOrder)	"123 N. Oak St, Apt #5"	"1235**"
Scan(RL, DIGIT, 6, KeepCase, SameOrder)	"123 N. Oak St, Apt #5"	"**1235"
Scan(LR, NONBLANK, 20, KeepCase, SameOrder)	"123 N. Oak St, Apt #5"	"123N.OakSt,Apt#5*****"
Scan(LR, ALL, 10, ToUpper, SameOrder)	"123 N. Oak St, Apt #5"	123 N. OAK"
Scan(LR, DIGIT, 9, KeepCase, SameOrder)	"412-67-1784"	"412671784"
Scan(LR, DIGIT, 9, KeepCase, L2HKeepDup)	"412-67-1784"	"112446778"
Scan(LR, DIGIT, 9, KeepCase, L2HDropDup)	"412-67-1784"	"124678****"
Scan(RL, DIGIT, 7, KeepCase, SameOrder)	" +501-555-1234"	"5551234"
Scan(RL, DIGIT, 7, KeepCase, L2HKeepDup)	" +501-555-1234"	"1234555"
Scan(RL, DIGIT, 7, KeepCase, L2HDropDup)	" +501-555-1234"	"**12345"

- CompareTo - When the <Term> element contains the CompareTo attribute, the <Term> will define a cross-attribute comparison (see section "2.12 Cross-Attribute Compare (CAC)" for a full example)
 - Valid values consist of either be a single OYSTER attribute identifier defined in the same Attributes script defining the <Term> element or a list of valid OYSTER attribute identifiers separated with a semi-colon (;)
 - e.g. CompareTo="Item_A; Item_B; Item_C".
 - The combined list of OYSTER attribute identifiers defined by both the Item and the CompareTo attributes must all be distinct (not repeated).

Multiple Term tags are used to build a single matching rule.

The <Term> element of an identity rule must enclose only one of the following combinations of attributes

1. Item and MatchResult
2. Item, MatchResult, CompareTo

OYSTERs default matching algorithm supports the above matching codes 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) .

2.7 <Indices> Tag

```
<Indices>
  .
  <child_elements>...</child_elements>
  .
</Indices>
```

The `Indices` tag has no attributes. The start and end tag enclose the `Index` and `Segment` tags described in the next two section. The `Index` elements are considered child elements to this element and the `Segment` elements are considered grandchildren of this element.

2.8 <Index> Tag

```
<Index Ident="ID">
  .
  <child_elements>...</child_elements>
  .
</Index>
```

The `Index` tag has one attribute.

- `Ident` - assigned value represents a unique name for the `Index` being defined.

The start and end tag enclose the `Segment` tags described in the next section. The `Segment` elements are considered child elements to this element.

Any number of `Indexes` can be defined for any given run and are defined by the user. Each `Index` must be provided a unique `Ident` value.

2.9 <Segment> Tag

```
<Segment Item="Name" Hash="Hash " />
```

The `Segment` tag has two attributes.

- `Item`
 - The value must be a valid OYSTER attribute defined by an `<Attribute>` element in the same `OysterAttribute` script containing the `<Index>` element.
- `Hash`
 - The value must be a valid `Index Hash` algorithm defined by an OYSTER Utility Class.
 - `Scan`

- Soundex
- DMSoundex
- IBMAlphaCode
- NYSIIS
- The Hash algorithm designated by the value of a Hash attribute will transform the values of the attribute identified by the Item attribute of the same <Segment> element into a fixed-length character string according to the hash algorithm.

Note that the final index value produced by an <Index> statement will be a concatenation of the individual hash values produced that are defined by each <Segment> element. The hash values will be concatenated in the same order as <Segment> elements occur in the <Index> statement.

Each <Segment> element must have an Item attribute and a Hash attribute or OYSTER will generate an error.

2.10 User Defined Index (UDI) Example

The following is an example of how to build two custom indices:

```
<Indices>
  <Index Ident="X1">
    <Segment Item="FirstName" Hash="Soundex"/>
    <Segment Item="LastName" Hash="Scan(LR, LETTER, 7, Y,
SameOrder)" />
    <Segment Item="HomePhone" Hash="Scan(RL, DIGIT, 7, N,
SameOrder)" />
  </Index>
  <Index Ident="X2">
    <Segment Item="LastName" Hash="NYSIIS(6)"/>
    <Segment Item="SchoolCode" Hash="Scan(LR, ALPHA, 6, Y, Same)" />
    <Segment Item="SocialSecNbr" Hash="Scan(LR, DIGIT, 9, N,
L2HKeepDup)" />
  </Index>
</Indices>
```

The example shows the instructions for creating two indices "X1" and "X2". Both indices are created by hashing three attributes. For a detailed explanation of the principals involved with designing properly aligned UDI, see section "VII. OYSTER User-Defined Inverted Index" on page 70 of this Reference Guide.

2.11 Scan Hash Function

The Scan function is a new hash algorithm that was created for OYSTER and introduced in v3.3 to complement the new UDI functionality (discussed in the previous few sections). The algorithm examines the string value either from left-to-right or from right-to-left (Direction), depending on which is specified, searching for the type of characters

specified (`CharType`). As valid characters that match the `CharType` are found they are extracted to form the hash value until the requested number of characters (`Length`) has been found, or until the end of the value string is encountered. If the number of characters found is less than the number requested, the hash value is either left or right padded with asterisk (*), depending on the specified `Direction`, to meet the `Length` request. The algorithm also accepts `Casing` and `Order` parameters to designate operations to be performed on the hash value after the characters have been extracted. The `Casing` parameter allows the user to specify whether the letters in the hash value are to be converted to uppercase or left as is. The `Order` parameter allows the user to specify the reordering of the characters in the hash value. The syntax of the Scan algorithm is as follows:

SCAN(`Direction`, `CharType`, `Length`, `Casing`, `Order`)

As described above, the scan algorithm has the following five (5) parameters:

- Parameters
 - `Direction`
 - Acceptable Values
 - LR – scan string left-to-right
 - RL – scan string right-to-left
 - `CharType`
 - Acceptable Values
 - ALL – extract all characters
 - NONBLANK – extract only non-blank characters
 - ALPHA – extract only letter and digit characters
 - LETTER – extract only letters of the English alphabet
 - DIGIT – extract only digits 0-9
 - `Length`
 - Acceptable Values
 - An integer between 1 and 30
 - `Casing`
 - Acceptable Values
 - ToUpper – all letters in the hash should be converted to upper casing.
 - KeepCase – all letters in the hash should keep original casing.
 - `Order`
 - Acceptable Values
 - SameOrder – keep original order in which the characters were extracted.
 - L2HKeepDup – reorder the hash characters from lowest to highest values. Keep any dup characters.
 - L2HDropDup – reorder the hash characters from lowest to highest values. Drop any duplicate characters from the hash.

Examples: the Table 1 shows some example uses of the SCAN algorithm along with how the various options affect the produced Hash value.

Table 1: Scan Algorithm Examples

Scan Configuration	String Value	Hash Value
Scan(LR, ALPHA, 8, ToUpper, SameOrder)	"123 N. Oak St, Apt #5"	"123NOAKS"
Scan(RL, ALPHA, 8, ToUpper, SameOrder)	"123 N. Oak St, Apt #5"	"AKSTAPT5"
Scan(LR, DIGIT, 6, KeepCase, SameOrder)	"123 N. Oak St, Apt #5"	"1235**"
Scan(RL, DIGIT, 6, KeepCase, SameOrder)	"123 N. Oak St, Apt #5"	"**1235"
Scan(LR, NONBLANK, 20, KeepCase, SameOrder)	"123 N. Oak St, Apt #5"	"123N.OakSt,Apt#5*****"
Scan(LR, ALL, 10, ToUpper, SameOrder)	"123 N. Oak St, Apt #5"	123 N. OAK"
Scan(LR, DIGIT, 9, KeepCase, SameOrder)	"412-67-1784"	"412671784"
Scan(LR, DIGIT, 9, KeepCase, L2HKeepDup)	"412-67-1784"	"112446778"
Scan(LR, DIGIT, 9, KeepCase, L2HDropDup)	"412-67-1784"	"124678***"
Scan(RL, DIGIT, 7, KeepCase, SameOrder)	"412-67-1784"	"412671784"
Scan(RL, DIGIT, 7, KeepCase, L2HKeepDup)	"412-67-1784"	"112446778"
Scan(RL, DIGIT, 7, KeepCase, L2HDropDup)	"412-67-1784"	"124678***"
Scan(LR, DIGIT, 9, KeepCase, SameOrder)	"123 N. Oak St, Apt #5"	"123NOAKS"
Scan(RL, DIGIT, 7, KeepCase, L2HKeepDup)	"123 N. Oak St, Apt #5"	"AKSTAPT5"
Scan(RL, DIGIT, 7, KeepCase, L2HDropDup)	"123 N. Oak St, Apt #5"	"**1235"
Scan(LR, NONBLANK, 20, KeepCase, SameOrder)	"123 N. Oak St, Apt #5"	"123N.OakSt,Apt#5*****"
Scan(LR, ALL, 10, ToUpper, SameOrder)	"123 N. Oak St, Apt #5"	123 N. OAK"
Scan(LR, DIGIT, 9, KeepCase, SameOrder)	"412-67-1784"	"412671784"
Scan(LR, DIGIT, 9, KeepCase, L2HKeepDup)	"412-67-1784"	"112446778"
Scan(LR, DIGIT, 9, KeepCase, L2HDropDup)	"412-67-1784"	"124678***"
Scan(RL, DIGIT, 7, KeepCase, SameOrder)	"412-67-1784"	"412671784"
Scan(RL, DIGIT, 7, KeepCase, L2HKeepDup)	"412-67-1784"	"112446778"
Scan(RL, DIGIT, 7, KeepCase, L2HDropDup)	"412-67-1784"	"124678***"
Scan(LR, DIGIT, 9, KeepCase, SameOrder)	"123 N. Oak St, Apt #5"	"123NOAKS"
Scan(RL, DIGIT, 7, KeepCase, L2HKeepDup)	"123 N. Oak St, Apt #5"	"AKSTAPT5"
Scan(RL, DIGIT, 7, KeepCase, L2HDropDup)	"123 N. Oak St, Apt #5"	"**1235"
Scan(LR, NONBLANK, 20, KeepCase, SameOrder)	"123 N. Oak St, Apt #5"	"123N.OakSt,Apt#5*****"
Scan(LR, ALL, 10, ToUpper, SameOrder)	"123 N. Oak St, Apt #5"	123 N. OAK"
Scan(LR, DIGIT, 9, KeepCase, SameOrder)	"412-67-1784"	"412671784"
Scan(LR, DIGIT, 9, KeepCase, L2HKeepDup)	"412-67-1784"	"112446778"
Scan(LR, DIGIT, 9, KeepCase, L2HDropDup)	"412-67-1784"	"124678***"
Scan(RL, DIGIT, 7, KeepCase, SameOrder)	"412-67-1784"	"412671784"
Scan(RL, DIGIT, 7, KeepCase, L2HKeepDup)	"412-67-1784"	"112446778"
Scan(RL, DIGIT, 7, KeepCase, L2HDropDup)	"412-67-1784"	"124678***"

2.12 Cross-Attribute Compare (CAC)

OYSTER provides the functionality for the match rules to compare values between references that are logically stored different attributes. This is known as Cross-Attribute Compare. This functionality is best shown in the below example.

Suppose that two valid references are provided in Table 2:

Table 2: Valid References for CAC

	FName	LName	SSN	LEA
Ref1	Bill	Doe	123456789	K45
Ref2	Doe	William	(null)	123456789

Ref1 represents the Input Reference and Ref2 is the Candidate Reference.

If the cross-attribute comparison rule is given as:

```
<Rule Ident="1">
  <Term> Item="FName" CompareTo="LName" MatchResult="Nickname" />
  <Term> Item="LName" CompareTo="FName" MatchResult="Exact" />
  <Term> Item="LEA" CompareTo="SSN" MatchResult="Exact" />
</Rule Ident="1">
```

`</Rule>`

Based on these rules, the two references would be found to match.

3. OysterSourceDescriptor

This section describes all the different tags and attributes available in the OysterSourceDescriptor.

3.1 XML Declaration and Comments

All the OYSTER XML documents begin with an XML declaration and a comment section. The XML declaration is required and must be included exactly as follows:

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

The comments section is optional but should be included in the file. The following is the suggested format of the comments section:

```
<!--  
    Document    : SourceDescriptor.xml  
    Created on  :  
    Author      :  
    Description:  
        Purpose of the document follows.  
-->
```

3.2 <OysterSourceDescriptor> Tag

```
<OysterSourceDescriptor Name="source name">  
    .  
    <child_elements>...</child_elements>  
    .  
</OysterSourceDescriptor>
```

The OysterSourceDescriptor tag has a single attribute.

- Name
 - Should be assigned a unique value that identifies the source described in the OysterSourceDescriptor.
 - A unique Name value must be set in each OysterSourceDescriptor for each source to be used in the OYSTER run.
 - When performing Identity Resolution or Identity Update, the Name value must be different from the name values used on any of the previous runs that contributed to the identity input file.
 - This attribute is used in both the link output and the identity output files to specify which source an identity originated from.

This is the root element and thus the start and end tag encloses all other elements in the document. All other elements are considered child elements to this root element. The

OysterSourceDescriptor tag can enclose three tags:

- <Source>
- <ReferenceItems>
- <Item>

3.3 <Source> Tag

The `Source` tag defines the type of source and all the connection information required for the source. There must be exactly one source defined via the `Source` tag in every OysterSourceDescriptor XML file.

The `Source` tag can define connections to sources stored in three different formats which are defined by the value of the `Type` attribute.

- `Type`
 - Acceptable Values
 - "FileDelim"
 - "FileFixed"
 - "Database"

3.3.1 `Type="FileDelim"`

```
<Source Type="FileDelim" Char="|" Qual="" Labels="Y">Absolute  
Path</Source>
```

The `Source` tag has four attributes, including the `Type` attribute, when `Type` is assigned the value of "FileDelim".

- `Type` - must be assigned the value of "FileDelim"
- `Char` - the value assigned must identify the character used as the delimiter for the attributes that form a reference in the source file.
 - Special characters used as delimiters:
 - Tab - specified as "\t", "[t]", or "\u0008"
 - Quotation - specified as """;
- `Qual` - the value assigned must identify the character used to qualify attributes in the source file.
- `Labels` - identifies if the source file contains labels as the first line of the document.
 - Acceptable Values
 - "Y" - Causes OYSTER to ignore the first line in the source file.
 - "N" - Causes OYSTER to include the first line in the source file.

The start and end tag enclose character data that represents the absolute path to the delimited text file that contains the source data.

3.3.2 Type="FileFixed"

```
<Source Type="FileFixed">Absolute Path</Source>
```

The `Source` tag has no additional attributes when the `Type` attribute is assigned the value of "FileFixed".

The start and end tag enclose character data that represents the absolute path to the fixed width text file that contains the source data.

3.3.3 Type="Database"

When the `Type` attribute is assigned the value "Database" there is three different formats available for the `Source` tag.

3.3.3a ODBC Connections

```
<Source Type="Database" SID="Database_Name" UserID="User" Passwd="Pass"  
CType="odbc">Table Name</Source>
```

The `Source` tags that define ODBC connections have either three or five attributes including the `Type` attribute.

- `Type` – Must be assigned the value of "Database"
- `SID` – Must be assigned the name of the database in which the source data is stored on the server.
- `UserID` – Must be assigned the name of the user with access to write to the desired table.
 - Optional - Only required if the `UserID` and `Password` are not defined during the creation of the actual ODBC connection
- `Passwd` – Must be assigned the password associated to the user given as the values of the `UserID` attribute.
 - Optional - Only required if the `UserID` and `Password` are not defined during the creation of the actual ODBC connection
- `CType` – Must be assigned the value of "odbc"

The start and end tag enclose character data that represents the name of the table that contains the source data.

3.3.3b Standard Database Connections

```
<Source Type="Database" Server="123.123.123.123" Port="####"  
SID="Database_Name" UserID="User" Passwd="Pass"  
CType="Type_of_Database">Table Name</Source>
```

The `Source` tags that define standard database connections have seven attributes including the `Type` attribute.

- `Type` – Must be assigned the value of “Database”
- `Server` – Must be assigned the address of the database server to be used.
 - DNS Name
 - IP address
- `Port` – Must be assigned the port number on which the database server will respond to requests.
- `SID` – Must be assigned the name of the database in which the source data is stored on the server.
- `UserID` – Must be assigned the name of the user with access to the desired table.
- `Passwd` – Must be assigned the password associated to the user given as the value of the `UserID` attribute.
- `CType` – Value is based on the type of database to which a connection is being made.
 - Possible values
 - “mysql”
 - “oracle”
 - “postgresql”
 - “sqlserver”

The start and end tag enclose character data that represents the table name in which the source data is located. This character data must represent a fully qualified table name. An example of a fully qualified table name is “DatabaseName.dbo.TableName” in SQL Server or “DatabaseName.TableName” in MySQL.

3.4 <ReferenceItems> Tag

```
<ReferenceItems>  
.  
  <child_elements>...</child_elements>  
.  
</ReferenceItems>
```

The `ReferenceItems` tag has no attributes. The start and end tag enclose the `Item` tags described in the next section. The `Item` elements are considered child elements to this element.

3.5 <Item> Tag

The `Item` tags are used to define each attribute of the source data. There must be one defined `Item` tag for each attribute used to define a reference located in the source data including one for the reference identifier.

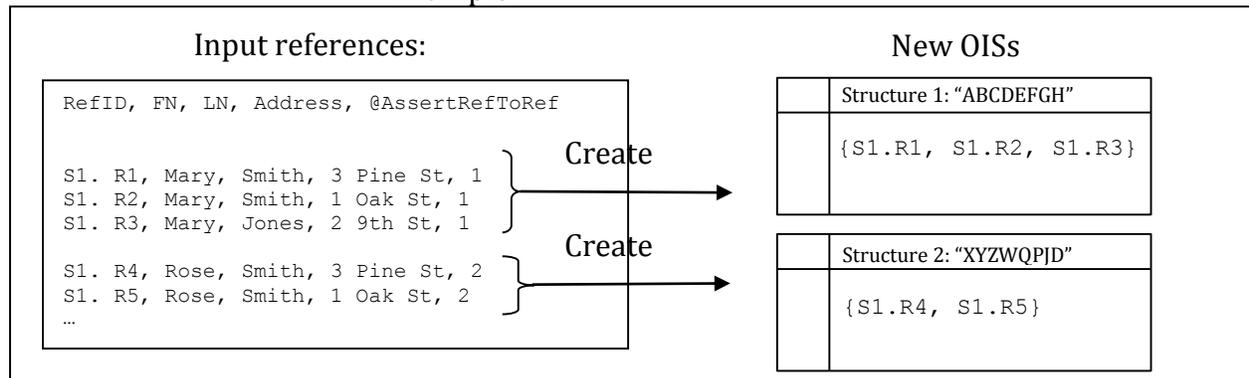
The `Item` tag has three different formats. These formats are directed by the value assigned to the `Type` attribute discussed previously in the `<Source>` Tag section.

3.5.1 Delimited Text Items

```
<Item Name="item 1" Attribute="RID" Pos="0"/>
```

The `Item` tag has three attributes when defining an attribute in a delimited text file.

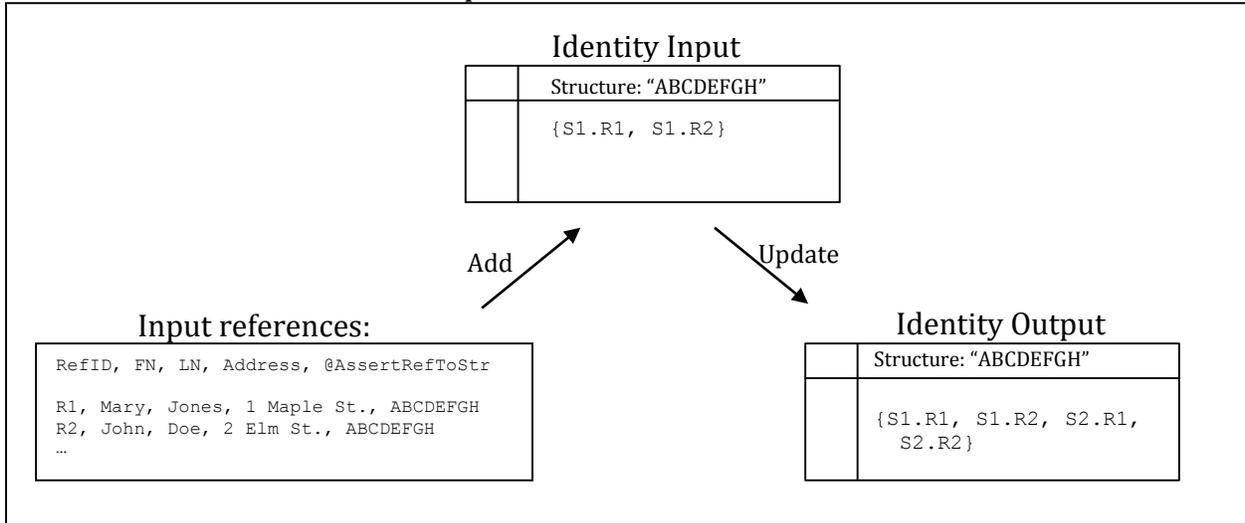
- Name – User defined value that names the attribute
- Attribute - values must correspond directly to the `Item` values in the `OysterAttributes.xml` file
 - OYSTER Keywords
 - `@RefID` - is used to inform OYSTER what field in the data is the unique keyed field
 - **NOTE: All RefIDs in the source file MUST be unique.**
 - `@Skip` - is used to skip data that may appear in the file but will not be used for integration.
 - `@AssertRefToRef` - is used when performing a Reference to Reference assertion run. This keyword implies that the input references which have same value of `@AssertRefToRef` must be grouped together to a new output identity with a new OYSTER ID
 - Example:



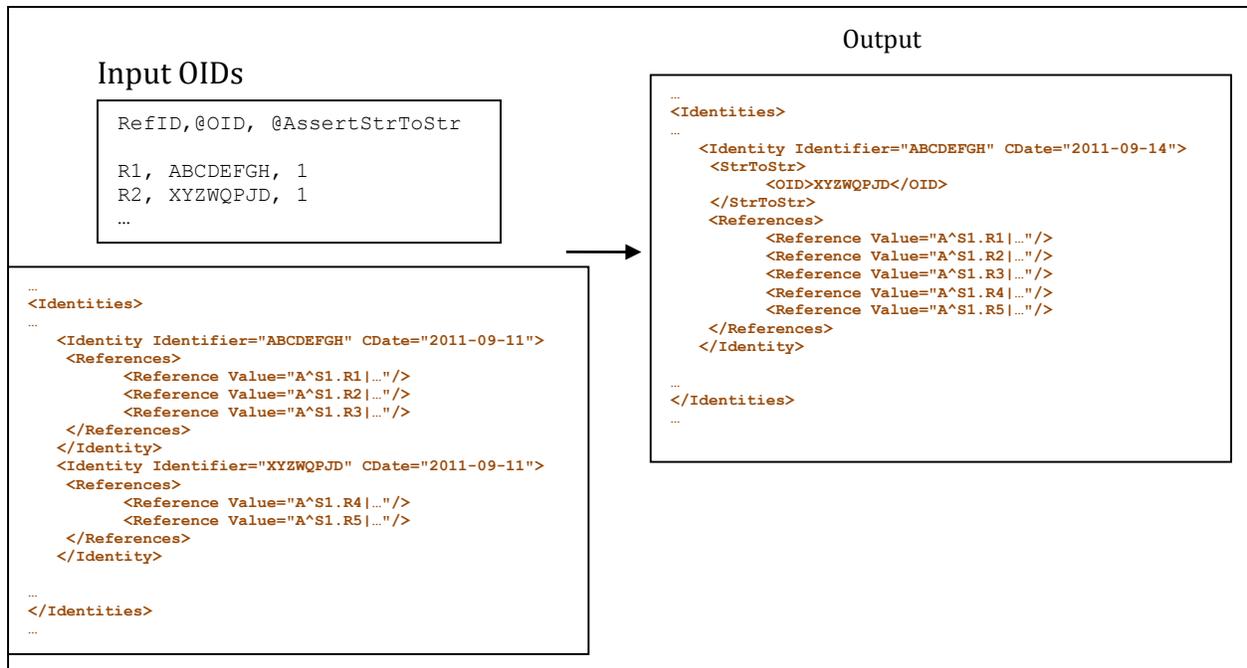
- `@AssertRefToStr` - is used when performing a Reference to Structure assertion run. This keyword informs OYSTER that the reference should be merged into the structure that matches the

OYSTER ID identified by the @AssertRefToStr tag. OYSTER will ignore all defined rules.

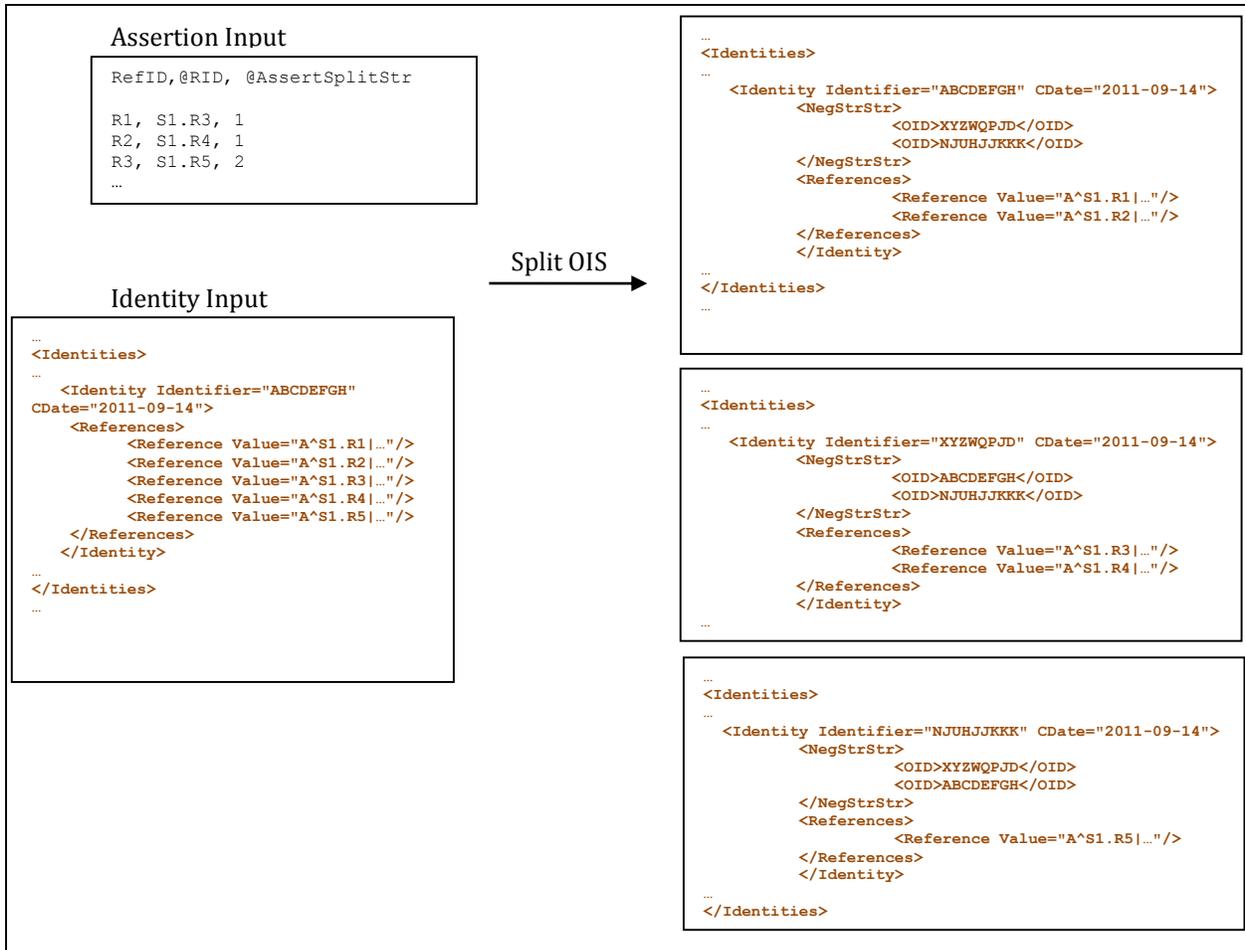
- Example:



- @OID – Identifies valid OYSTER IDs in an input file that will, when used in conjunction with @AsserStrToStr, cause the OYSTER IDs to consolidate into a single OYSTER ID.
- @AssertStrToStr – is used when performing a Structure to Structure assertion run. This keyword informs OYSTER that it should ignore all defined rules and force matches on structures (Previously resolved Identities) that have been assigned like numbers for this attribute.
 - Must be used in conjunction with @OID
 - Example:



- @RID - The value of @RID must be a valid reference ID in an existing OYSTER identity structure. The Reference IDs of @RID which have same value of @AssertSplitStr must be split from original identity into a new OYSTER identity with a new OYSTER ID.
- @AssertSplitStr - is used when performing a Split Structure assertion run. This keyword informs OYSTER that it should ignore all defined rules and force a split on structures that have been assigned like numbers for this attribute.
 - Must be used in conjunction with @RID
 - Example:



- POS - value is used to specify the location of the item in the source file
 - Pos counting starts from the left most delimited attribute.
 - Pos numbering starts at 0

3.5.2 Fixed Width Items

```
<Item Name="item 2" Attribute="Attr 1" Start="10" End="11"/>
```

The Item tag has four attributes when defining an attribute in a fixed width text file.

- Name - User defined value that names the attribute
- Attribute - values must correspond directly to the Item values in the OysterAttributes.xml file
 - See previous section for full list of OYSTER Keywords
- Start - used to specify the location of the first character in the fixed width item in the source file
- End - used to specify the location of the last character in the fixed width item in the source file

3.5.3 Database Items

```
<Item Name="item 3" Attribute="Attr 2" />
```

The `Item` tag has two attributes when defining an attribute in a database table.

- Name
 - Must be assigned a value equal to the column name in the table in which the attribute is stored
 - Used to build the SQL Select statement used by OYSTER to fetch the references from the source database.
 - Used in the Matching Rules section when building the rules
- Attribute - value must correspond directly to the corresponding `Item` value in the `OysterAttributes.xml` file.
 - See previous section for full list of OYSTER Keywords

III. Attribute-Based vs. Record-Based Matching for Identities

There are two types of matching that can be done when matching records in an input source against a previously defined Knowledge Base (KB).

- Attribute-Based Matching - when the matching depends on any combination of attributes from any records contained in a single identity.
- Record-Based Matching - when the matching depends completely on the attributes in a single record contained in a single identity.

1. Example

This example illustrates how matching is done via Attribute-Based and Record-Based. The sample input records are:

```
1 | Sam | Jacobs | 05071982 | SC11
2 | Samuel | Jacobs | 05071982 | SC21
```

The example identity that has already been defined is:

```
<Identity Identifier="VUKPECCU0KHBJUDO" CDate="2011-06-14">
  <References>
    <Reference Value="A^source1.3|B^Sam|C^Jacobs|D^05071983|E^SC21"/>
    <Reference Value="A^source1.4|B^Sam|C^Jacobs|D^05071982|E^SC11"/>
    <Reference Value="A^source1.5|B^Samual|C^Jacobs|D^05071982|E^SC11"/>
  </References>
</Identity>
```

The match rules used to determine a match state that first name, last name, and school code (SC##) must match exactly.

1.1 Attribute-Based

As defined above, Attribute-Based matching depends on any combination of attributes from any records contained in a previously defined identity. This means that the first name, last name, and school code attribute for each record in the source must have a matching attribute in any record in the identity it is being compared against.

This is how the matching takes place:

1. Record "1 | Sam | Jacobs | 05071982 | SC11" is selected.
2. The first name, last name and school code are noted:

- a. Sam, Jacobs, SC11
3. All records are selected from the identity:
 - <Reference Value="A^source1.3|B^Sam|C^Jacobs|D^05071983|E^SC21"/>
 - <Reference Value="A^source1.4|B^Sam|C^Jacobs|D^05071982|E^SC11"/>
 - <Reference Value="A^source1.5|B^Samual|C^Jacobs|D^05071982|E^SC11"/>
4. The first name, last name, and school code are noted:
 - a. Sam & Samual, Jacobs, SC21 & SC11
5. The attributes from the source and the identity record are compared
 - a. Does Sam = Sam or Samual?
 - i. Yes
 - b. Does Jacobs = Jacobs?
 - i. Yes
 - c. Does SC11 = SC21 or SC11?
 - i. Yes
6. Since all the matching rules are satisfied, it is determined that the source record is a match and it is now part of the identity.
7. The new identity would look like this:

```
<Identity Identifier="VUKPECCU0KHBJUDO" CDate="2011-06-14">
  <References>
    <Reference Value="A^source1.3|B^Sam|C^Jacobs|D^05071983|E^SC21"/>
    <Reference Value="A^source1.4|B^Sam|C^Jacobs|D^05071982|E^SC11"/>
    <Reference Value="A^source1.5|B^Samual|C^Jacobs|D^05071982|E^SC11"/>
    <Reference Value="A^source2.1|B^Sam|C^Jacobs|D^05071982|E^SC11"/>
  </References>
</Identity>
```

8. Once the first record is matched, the process aborts for this identity and selects the next source record:
 - a. 2|Samual|Jacobs|05071982|SC21
9. If you apply the same process to this source record you will see that it too matches and is part of the identity.
10. The final identity structure looks like:

```
<Identity Identifier="VUKPECCU0KHBJUDO" CDate="2011-06-14">
  <References>
    <Reference Value="A^source1.3|B^Sam|C^Jacobs|D^05071983|E^SC21"/>
    <Reference Value="A^source1.4|B^Sam|C^Jacobs|D^05071982|E^SC11"/>
    <Reference Value="A^source1.5|B^Samual|C^Jacobs|D^05071982|E^SC11"/>
    <Reference Value="A^source2.1|B^Sam|C^Jacobs|D^05071982|E^SC11"/>
    <Reference Value="A^source2.2|B^Samual|C^Jacobs|D^05071982|E^SC21"/>
  </References>
</Identity>
```

1.2 Record-Based

As defined above, Record-Based matching is depended on all the attributes in each individual record of the previously defined identity. This means that the first name, last

name, and school code attribute for each record in the source must have an exactly matching record in the identity it is being compared against.

This is how the matching takes place:

1. Record "1|Sam|Jacobs|05071982|SC11" is selected.
2. The first name, last name and school code are noted:
 - a. Sam, Jacobs, SC11
3. Record "A^source1.3|B^Sam|C^Jacobs|D^05071983|E^SC21" is selected from the identity.
4. The first name, last name, and school code are noted:
 - a. Sam, Jacobs, SC21
5. The attributes from the source and the identity record are compared
 - a. Does Sam = Sam?
 - i. Yes
 - b. Does Jacobs = Jacobs?
 - i. Yes
 - c. Does SC11 = SC21?
 - i. No
6. Since all 3 attributes do not match this record from the identity is discarded and the next record from the identity is selected:
 - a. A^source1.4|B^Sam|C^Jacobs|D^05071982|E^SC11"/>
7. The attributes from the source and the identity record are compared
 - a. Does Sam = Sam?
 - i. Yes
 - b. Does Jacobs = Jacobs?
 - i. Yes
 - c. Does SC11 = SC11?
 - i. Yes
8. Since all the matching rules are satisfied, it is determined that the source record is a match and it is now part of the identity.
9. The new identity would look like this:

```
<Identity Identifier="VUKPECCU0KHBJUDO" CDate="2011-06-14">
  <References>
    <Reference Value="A^source1.3|B^Sam|C^Jacobs|D^05071983|E^SC21"/>
    <Reference Value="A^source1.4|B^Sam|C^Jacobs|D^05071982|E^SC11"/>
    <Reference Value="A^source1.5|B^Samual|C^Jacobs|D^05071982|E^SC11"/>
    <Reference Value="A^source2.1|B^Sam|C^Jacobs|D^05071982|E^SC11"/>
  </References>
</Identity>
```

10. Once the first record is matched, the process aborts for this identity and selects the next source record:
 - a. 2|Samual|Jacobs|05071982|SC21

11. If you apply the same process to this source record you will see that the records do not match so a new identity will be created. The final identities will look like this:

```
<Identity Identifier="VUKPECCU0KHBJUDO" CDate="2011-06-14">
  <References>
    <Reference Value="A^source1.3|B^Sam|C^Jacobs|D^05071983|E^SC21"/>
    <Reference Value="A^source1.4|B^Sam|C^Jacobs|D^05071982|E^SC11"/>
    <Reference Value="A^source1.5|B^Samual|C^Jacobs|D^05071982|E^SC11"/>
    <Reference Value="A^source2.1|B^Sam|C^Jacobs|D^05071982|E^SC11"/>
  </References>
</Identity>
<Identity Identifier="SUGPQC4UBKPBAU90" CDate="2011-06-14">
  <References>
    <Reference Value="A^source2.2|B^Samual|C^Jacobs|D^05071982|E^SC21"/>
  </References>
</Identity>
```

Please note that the Record-Based matching will produce fewer consolidations in general but requires less processing and memory which speeds up the overall processing time.

IV. Reference Sources

Appendix B shows the modified SourceDescriptor that is required for each of the four examples in this document to be run against a fixed width text file instead of the delimited text files that were used in the main body.

For OYSTER to use different types of sources as input only the SourceDescriptor file needs to be modified. This is due to the fact that the connection string for the data source is defined inside of the SourceDescriptor along with the fields used for attributes that are stored in the defined source.

1. Example from Text Files

This section defines example OysterSourceDescriptor files that can be used to read reference sources that are store in text files.

1.1 From Delimited Files

Figure 1 illustrates a pipe (|) delimited text file that can be used as input for an OYSTER run. Any text character can be used as the delaminating character so long as it is specifically defined in the SourceDescriptor.

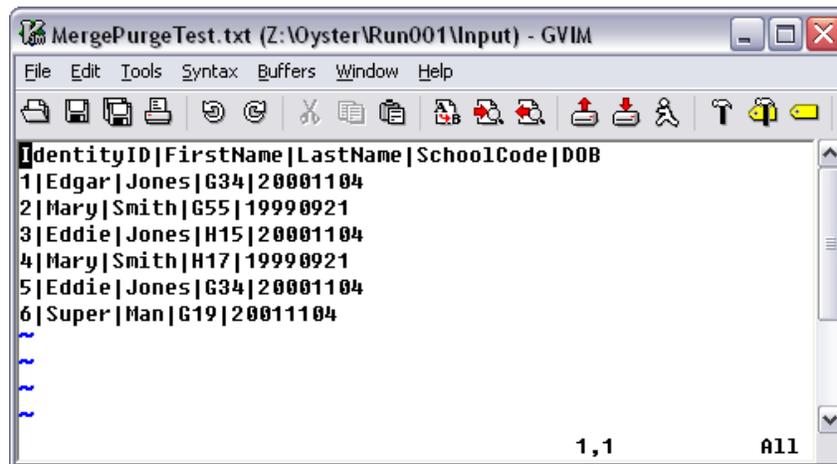
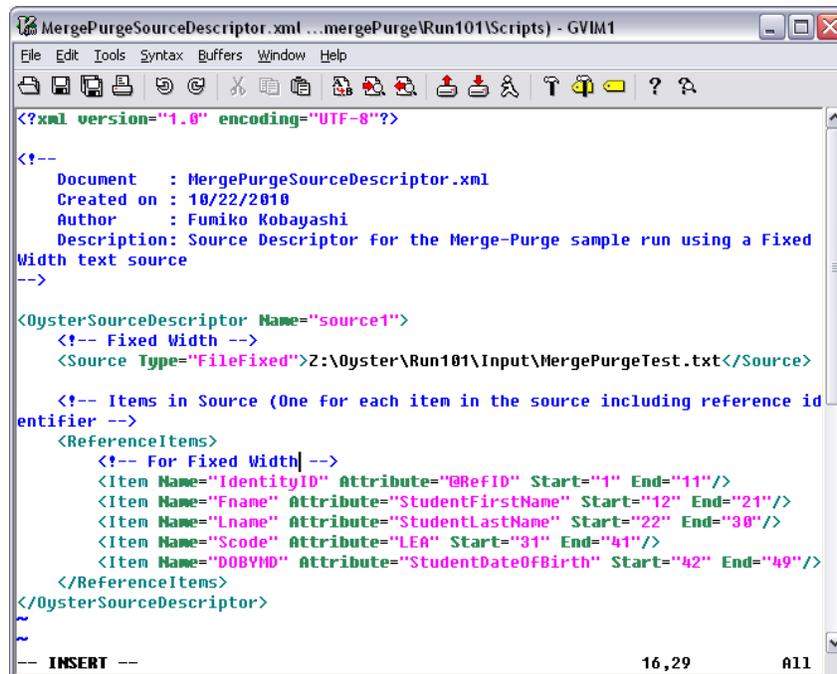


Figure 1: Delimited Text Input

The SourceDescriptor illustrated in Figure 2 can be used in an OYSTER run if the input source is stored in a delimited text file like the one shown in Figure 1.



```
<?xml version="1.0" encoding="UTF-8"?>
<!--
  Document   : MergePurgeSourceDescriptor.xml
  Created on : 10/22/2010
  Author    : Fumiko Kobayashi
  Description: Source Descriptor for the Merge-Purge sample run using a Fixed
  Width text source
-->

<OysterSourceDescriptor Name="source1">
  <!-- Fixed Width -->
  <Source Type="FileFixed">Z:\Oyster\Run101\Input\MergePurgeTest.txt</Source>

  <!-- Items in Source (One for each item in the source including reference id
  entifier -->
  <ReferenceItems>
    <!-- For Fixed Width -->
    <Item Name="IdentityID" Attribute="@RefID" Start="1" End="11"/>
    <Item Name="Fname" Attribute="StudentFirstName" Start="12" End="21"/>
    <Item Name="Lname" Attribute="StudentLastName" Start="22" End="30"/>
    <Item Name="Scode" Attribute="LEA" Start="31" End="41"/>
    <Item Name="DOBYMD" Attribute="StudentDateOfBirth" Start="42" End="49"/>
  </ReferenceItems>
</OysterSourceDescriptor>
~
~
-- INSERT --                               16,29      All
```

Figure 4: Source Descriptor for a Fixed Width Text File

2. Example from Database Tables

This section defines example OysterSourceDescriptor files that can be used to read reference sources that are store in various database tables. OYSTER can connect to data base tables through ODBC connections and also has the functionality to allow for connection strings to be defined inside the SourceDescriptor file to form direct connections to other DBMSs.

2.1 From Open Database Connectivity (ODBC)

OYSTER has the ability to make use of existing ODBC connections to read reference sources from tables stored in databases systems that can be connected to via ODBC; Microsoft Access is one such database system.

2.1.1 Creating ODBC Connections

ODBC connections are simple to create with Windows XP or a newer Windows OS. The following instructions will guide you through creating an ODBC connection to an Access Database that can be used as an OYSTER source.

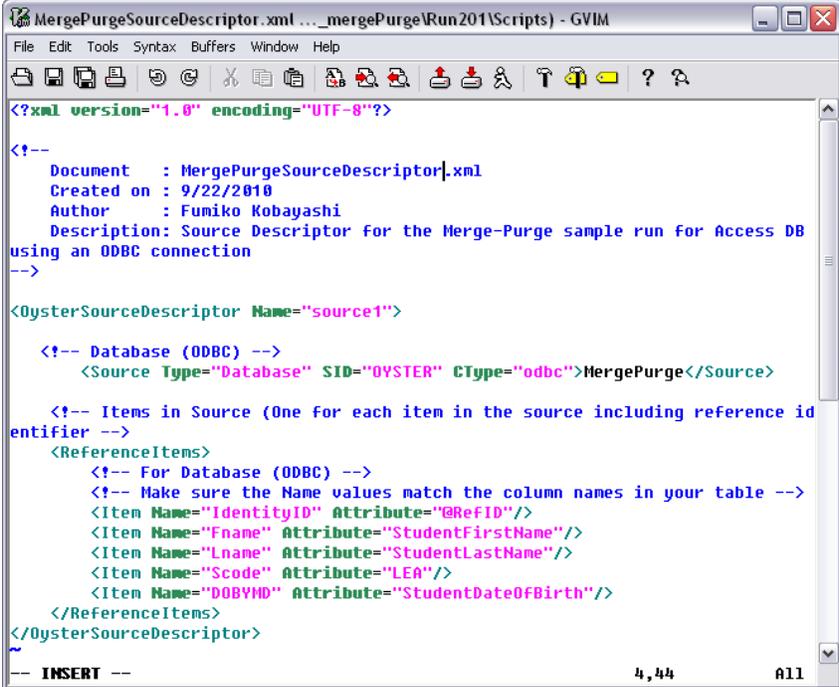
1. Open your control panel: Click on Start, then click on Control Panel.

2. Next, double click on the Administrative Tools icon.
3. Then, double click on the Data Sources (ODBC) icon.
4. In the ODBC Data Source Administrator Window, click on the ADD button on the right side of the screen.
5. In the Create New Data Source window, scroll the list and click on Microsoft Access Driver (*.mdb, *.accdb)
6. Click Finish
7. In the ODBC Microsoft Access Setup window, enter OYSTER in the Data Source Name field.
8. In the Database section click Select.
9. Browse to the folder in which your OYSTER source database is located and select the Access Database file from the list.
10. Click OK for each Window until all windows are closed. The ODBC connection is now set up.

ODBC can be used to create connection too many different types of data sources. For more information about ODBC connections please refer to this Microsoft knowledge base article: <http://support.microsoft.com/kb/110093>.

2.1.2 Example from ODBC

The SourceDescriptor illustrated in Figure 5 can be used by OYSTER if the input source is stored in a database, such as Microsoft Access, in which an ODBC connection has been established.



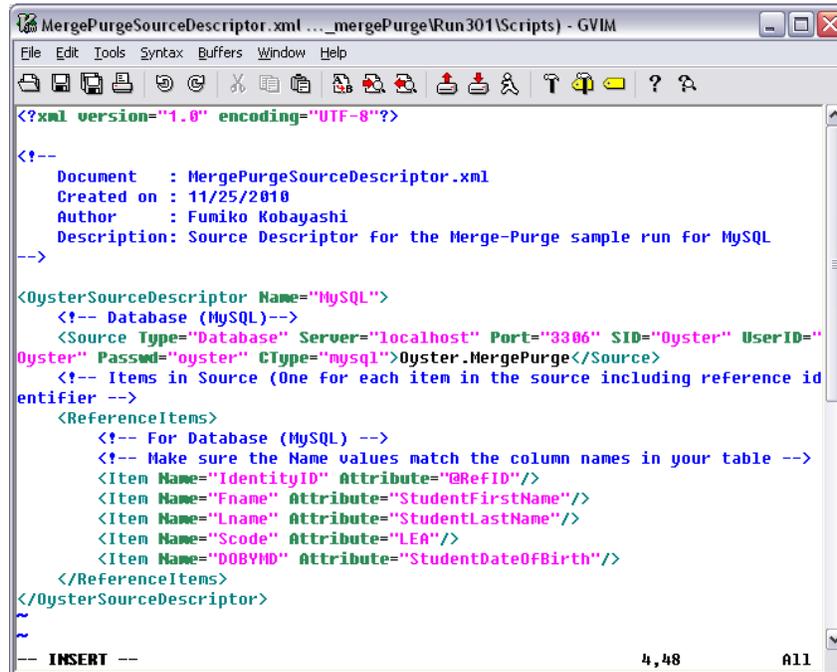
```
<?xml version="1.0" encoding="UTF-8"?>
<!--
  Document   : MergePurgeSourceDescriptor.xml
  Created on : 9/22/2010
  Author    : Fumiko Kobayashi
  Description: Source Descriptor for the Merge-Purge sample run for Access DB
              using an ODBC connection
-->
<OysterSourceDescriptor Name="source1">
  <!-- Database (ODBC) -->
  <Source Type="Database" SID="OYSTER" CType="odbc">MergePurge</Source>

  <!-- Items in Source (One for each item in the source including reference id
  entifier -->
  <ReferenceItems>
    <!-- For Database (ODBC) -->
    <!-- Make sure the Name values match the column names in your table -->
    <Item Name="IdentityID" Attribute="@RefID"/>
    <Item Name="Fname" Attribute="StudentFirstName"/>
    <Item Name="Lname" Attribute="StudentLastName"/>
    <Item Name="Scode" Attribute="LEA"/>
    <Item Name="DOBYMD" Attribute="StudentDateOfBirth"/>
  </ReferenceItems>
</OysterSourceDescriptor>
~
-- INSERT --
```

Figure 5: Source Descriptor for accessing an Access DB Table

2.2 From MySQL

The SourceDescriptor illustrated in Figure 6 can be used by OYSTER if the input source is stored in a MySQL server database table.



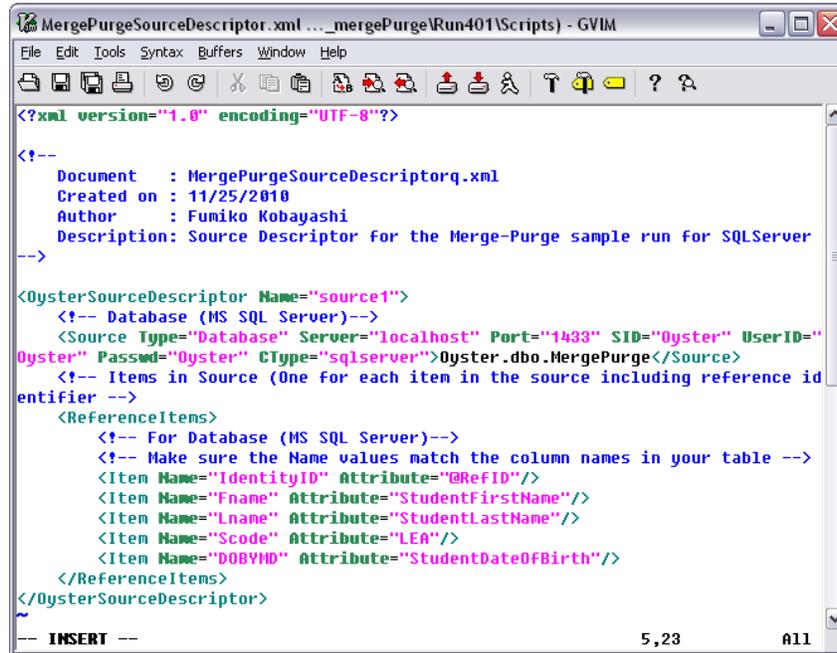
```
<?xml version="1.0" encoding="UTF-8"?>
<!--
  Document   : MergePurgeSourceDescriptor.xml
  Created on : 11/25/2010
  Author    : Fumiko Kobayashi
  Description: Source Descriptor for the Merge-Purge sample run for MySQL
-->

<OysterSourceDescriptor Name="MySQL">
  <!-- Database (MySQL)-->
  <Source Type="Database" Server="localhost" Port="3306" SID="Oyster" UserID="
Oyster" Passwd="oyster" CType="mysql">Oyster.MergePurge</Source>
  <!-- Items in Source (One for each item in the source including reference id
entifier -->
  <ReferenceItems>
    <!-- For Database (MySQL) -->
    <!-- Make sure the Name values match the column names in your table -->
    <Item Name="IdentityID" Attribute="@RefID"/>
    <Item Name="Fname" Attribute="StudentFirstName"/>
    <Item Name="Lname" Attribute="StudentLastName"/>
    <Item Name="Scode" Attribute="LEA"/>
    <Item Name="DOBYMD" Attribute="StudentDateOfBirth"/>
  </ReferenceItems>
</OysterSourceDescriptor>
~
-- INSERT --
```

Figure 6: Source Descriptor for accessing a MySQL DB Table

2.3 From Microsoft SQLServer

The SourceDescriptor illustrated in Figure 7 can be used by OYSTER if the input source is stored in a Microsoft SQLServer Database table.



```
<?xml version="1.0" encoding="UTF-8"?>
<!--
  Document   : MergePurgeSourceDescriptorq.xml
  Created on : 11/25/2010
  Author    : Fumiko Kobayashi
  Description: Source Descriptor for the Merge-Purge sample run for SQLServer
-->

<OysterSourceDescriptor Name="source1">
  <!-- Database (MS SQL Server)-->
  <Source Type="Database" Server="localhost" Port="1433" SID="Oyster" UserID="
Oyster" Password="Oyster" CType="sqlserver">Oyster.dbo.MergePurge</Source>
  <!-- Items in Source (One for each item in the source including reference id
entifier -->
  <ReferenceItems>
    <!-- For Database (MS SQL Server)-->
    <!-- Make sure the Name values match the column names in your table -->
    <Item Name="IdentityID" Attribute="@RefID"/>
    <Item Name="Fname" Attribute="StudentFirstName"/>
    <Item Name="Lname" Attribute="StudentLastName"/>
    <Item Name="Scode" Attribute="LEA"/>
    <Item Name="DOBYMD" Attribute="StudentDateOfBirth"/>
  </ReferenceItems>
</OysterSourceDescriptor>
-- INSERT --
```

Figure 7: Source Descriptor for accessing SQL Server Table

V. Outputs

1. Identity Document

The Identity document is an Output generated by OYSTER when a user defines a type and location for the `<IdentityOutput>` tag of the RunScript, discussed in section “1.9 `<IdentityOutput>`Tag” of this document.

This section describes the different tags and attributes generated by OYSTER for the Identity output document when the Identities discovered during an OYSTER run are saved.

1.1 `<root>` Tag

```
<root>
  .
  .
  <child_elements>...</child_elements>
  .
  .
</root>
```

The `<root>` tag has no generated attributes. This is the root element and thus the start and end tag encloses all other elements generated for the document. All other elements are considered child elements to this root element. The root tag will enclose 12 tags:

- `<Metadata>`
- `<Modifications>`
- `<Modification>`
- `<Attributes>`
- `<Attribute>`
- `<Identities>`
- `<Identity>`
- `<References>`
- `<Reference>`
- `<Value>`
- `<Traces>`
- `<Trace>`

1.2 `<Metadata>` Tag

```
<Metadata>
  .
  <child_elements>...</child_elements>
```

```
</Metadata>
```

The Metadata tag has no generated attributes. The start and end tags enclose the `Modifications`, `Modification`, `Attributes`, and `Attribute` tags, discussed in the next sections, which are children of the Metadata tag.

The Metadata tag occurs only once in an Identity output document and is located as the top section of the file above the defined identities.

1.2.1 <Modifications> Tag

```
<Modifications>
  .
  <child_elements>...</child_elements>
  .
</Modifications>
```

The Modifications tag has no generated attributes. The start and end tags enclose a series of `Modification` tags, discussed in the next section, that are children of the Modifications tag.

There will be a single `Modifications` tag generated by OYSTER in each Identity output file. This tag represents a history of all the runs that were performed to create the identity file up to and including the current run.

1.2.1.1 <Modification> Tag

```
<Modification ID="1" OysterVersion="3.3" Date="2012-08-24 19.21.30"
RunScript="RunScriptName" />
```

The `Modification` tag has four generated attributes.

- ID
 - This is a sequential number generated and assigned to each new `Modification` tag that is inserted into the Run Script
- `OysterVersion`
 - OYSTER derives this value from the version number embedded in the current OYSTER executable.
- `Date`
 - This is the date in which the Identity Output file was last updated.
- `RunScript`
 - This is the file name of the RunScript without the `.xml` extension.
 - Used to allow users to track run history.

A new `Modification` tag is generated each time OYSTER updates the Identity Output file

through and IdentityUpdate run. The multiple Modification tags can be used to help back track where records in an identity came from originally.

1.2.2 <Attributes> Tag

```
<Attributes>
  .
  <child_elements>...</child_elements>
  .
</Attributes>
```

The Attributes tag has no generated attributes. The start and end tags enclose a series of Attribute tags, discussed in the next section, that are children of the Attributes tag.

There will be a single Attributes tag generated by OYSTER in each Identity output file. This tag represents the Attributes used for every Identity tag defined by OYSTER.

1.2.2.1 <Attribute> Tag

```
<Attribute Name="value" Tag="value"/>
```

The Attribute tag has two generated attributes.

- Name
 - OYSTER derives this value from the Attribute value assigned by the user in the <Item> tag of the source descriptor.
- Tag
 - Unique value assigned to each attribute by OYSTER to allow for clearly defined relations to the reference assigned to the Value attribute of the <Reference> Tags, discussed in a following section, that make up this identity.

There will be one corresponding Attribute tag generated by OYSTER in the Identity output file for every unique value assigned to the Item tag in the Attributes file by the user.

1.3 <Identities> Tag

```
<Identities>
  .
  <child_elements>...</child_elements>
  .
</Identities>
```

The `Identities` tag has no generated attributes. The start and end tags enclose a series of `Identity` tags, discussed in the next section, that are children of the `Identities` tag.

There will be a single `Identities` tag generated by OYSTER in each Identity output file. This tag encloses all the identified identities defined by the `Identity` tag discussed in the next section.

1.3.1 `<Identity>` Tag

```
<Identity Identifier="Unique_Oyster_gernerated_ID" CDate="Date">
  .
  <child_elements>...</child_elements>
  .
</Identity>
```

The `Identity` tag contains two attributes. The start and end tag enclose a series of `Attribute` tags, discussed in the next section, that are children of the `Identity` tag.

- `Identifier`
 - OYSTER assigns this attribute the value of a uniquely generated OYSTER ID.
 - This ID is created to uniquely define the real world entity in which the child `Attributes` and `References` tags define.
- `CDate`
 - Denotes the date in which the identity was added to the `.idty` file.
 - Can be used in conjunction with the `Modification` tags to help backtrack where the identities originated from.

One `Identity` element will be defined for every unique real-world entity (cluster) discovered by the OYSTER runs.

1.3.2 `<References>` Tag

```
<References>
  .
  <child_elements>...</child_elements>
  .
</References>
```

The `References` tag has no generated attributes. The start and end tags enclose a series of `Reference` tags, discussed in the next section, that are children of the `References` tag.

There will be one corresponding `References` tag generated by OYSTER in the Identity output file for every unique `Identity` tag defined by OYSTER.

1.3.2.1 <Reference> Tag

```
<Reference>
  .
  <child_elements>...</child_elements>
  .
</Reference>
```

The `Reference` tag has no generated attributes. The start and end tags enclose a series of `Value` tags, discussed in the next section, that are children of the `Reference` tag.

There will be one corresponding `Reference` tag generated by OYSTER in the Identity output file for every input source reference processed by OYSTER.

1.3.2.1.1 <Value> Tag

```
<Value>A^AttributeValue1|B^AttributeValue2|...<Value/>
```

The `Value` tag has one no generated attributes. This tag contains all the attribute values that make up a reference. Each attribute is pipe (|) delimited and each attribute has a corresponding identifier concatenated to the front of the value that directly relates the part of the `Value` to an `Attribute` Tag through the `Tag` attribute.

- Example "A^ source1.2|B^Joe|C^6/4/1980|D^L902|E^010-94-8189"

There will be one corresponding `Value` tag generated by OYSTER in the Identity output file for every source reference that composes the identified cluster.

1.4 Example

OYSTER is designed to analyze every reference in a provided source and consolidate the unique values of the attributes into a unique identity that is discovered using the matching rules defined for the OYSTER run. Each discovered identity is stored using the tags discussed above. An Identity document generated by an OYSTER run is illustrated in Figure 8.

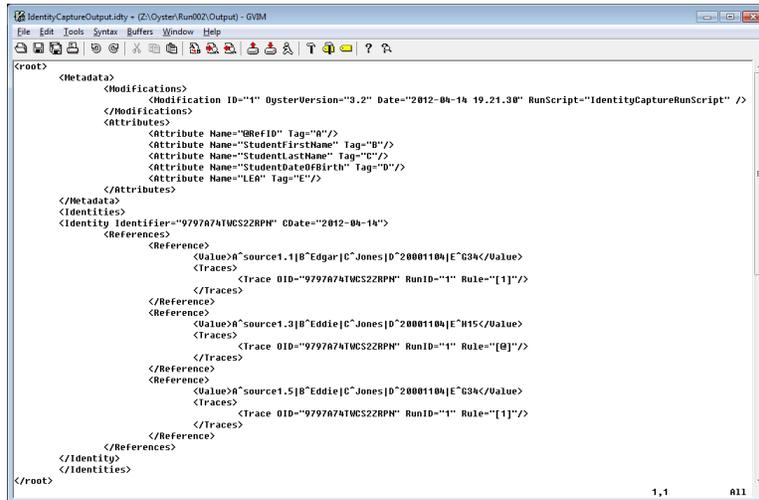


Figure 8: Example Identity Document

2. Link Index

The link index file is a file that is generated by OYSTER for every OYSTER run. This file defines the clusters by assigning each input reference a link value so that references found by OYSTER to identify the same real-world entity belong to same cluster.

This file contains three pieces of information for each reference processed during the OYSTER RUN.

- **RefID** – Identifies the reference by combining the source name and the reference ID of the reference.
 - The source name is defined by the user through the `Name` attribute of the `OysterSourceDescriptor` tag discussed in section “3.2 `<OysterSourceDescriptor>` Tag” of this document.
 - The reference ID of the reference is defined by the user through assigning the `@RefID` keyword to an `Attribute` attribute of an `Item` tag discussed in section 3.5 `<Item>` Tag of this document.
- **OysterID** – this is the unique link value defined by OYSTER and is used to identity references that belong to the same cluster.
 - Each cluster of references define the same real-world identity
 - The same OysterID is assigned to each member of a cluster.
- **Rule** – This identifies which rules were used when matching this reference to other references.
 - Rules are assigned unique identifiers by the user through the `Ident` attribute of the `Rule` tag discussed in section 2.5 `<Rule>` Tag of this document.
 - A record may have more than one rule listed if they were matched on more than one rule.

- A rule of “null” will be listed if no matches were found for the record.

An example Link Index is illustrated in Figure 9. This link Index defines three clusters, {source1.1, source1.3, source1.5}, {source1.2, source1.4}, and {source1.6} with cluster identifier (link) values of QN5Y16P5HK5JX6FA, ZKCZ2FFW4R6RICUV, and E255F96M3COGM323 respectively.

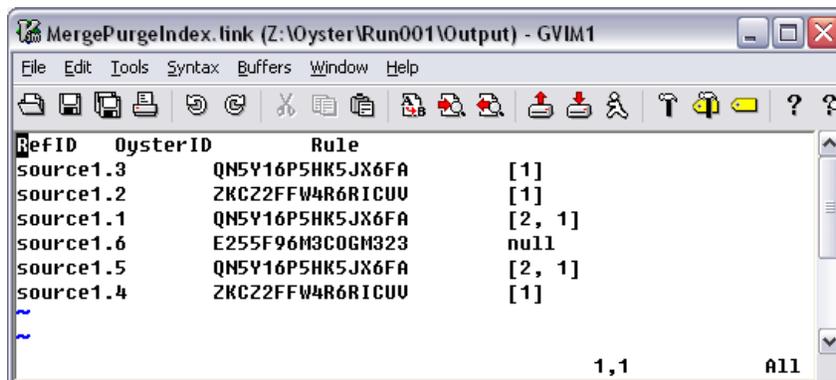


Figure 9: Example Link index

3. Identity Change Report

The Identity Change Report.txt file is created during every OYSTER run. It is stored in the directory specified for the output link index or .idty file. This file provides details pertaining to updates, merges, and new identity creations that took place during the run. It also provides information as to the number of input identities and output identities. The level of detail provided in this output document can be specified by setting the value of the “ChangeReportDetail” attribute to “On” or “Off” (see section 1.3 <Settings> Tag)

It is important to mention that due to how the RSwoosh and FScluster engines in OYSTER were initially designed to handle matching internally, the change reports may be different when running the same records on different engines. FScluster identifies some merges as updates since the update status takes precedence over merge internally whereas RSwoosh will identify the same consolidation as a merge. This is to be expected in OYSTER v3.2 and the initial release of OYSTER v3.3 but will be updated in the final release of OYSTER v3.3 such that the Change report matches between all engines.

3.1 Sections

The Identity Change Report consists of three (3) sections; Metadata, Summary, and Change Detail.

3.1.1 Metadata Section

The first section contains metadata that defines some basic information about the OYSTER run that generated the change report.

- `Date` – The date the change report is generated
- `RunScript Path` – the full file name, including extension, of the run script.
- `RunScript Name` – the name of the run script as specified in the `RunScriptName` attribute of the run script

3.1.2 Summary

The second section contains various counts that form a summary of the actions that took place during the run.

- `Count of Output Identities` – Total number of Identities written to `.idty` file.
- `Count of Input Identities` – Number of Identities read from identity input file.
- `Count of Input Identities Updated and Written to Output` – Count of Identities that existed in the input identity file that had new source references added to the identity.
- `Count of Input Identities Not Updated and Written to Output` – Count of identities from the input identity file that were not modified in any way during the run.
- `Count of Input Identities Merged` – Count of Merges that occurred during the run.
- `Count of New Identities Created` – count of new identities created from references in the input source that did not match any of the input identities on any rule.

The following relations should always be observed among the above counts:

1. `Output Identities` = `Input Identities` – `Merged Identities` + `New Identities`
2. `Input Identities` = `Input Identities Update` + `Input Identities Not Updated`

3.1.3 Change Detail

The third section contains details such as the OYSTER IDs and the Reference ID for identities and references involved in creating, merging, or updating identities.

- `New Identities Created` – list of all identities that are created during the run.
 - Contains two pieces of information
 - `Identifier` – OYSTER ID of the new identity

- Reference – list of all reference IDs that are contained in the new identity.
 - Only listed if `ChangeReportDetail="Yes"`
- Input Identities Merged – list of all identities that are merged during the run.
 - Contain two pieces of information
 - Input Identifier – The OYSTER ID that the identity had in the identity input file.
 - Output Identifier – The OYSTER ID that the identity had in the identity output file
- Input Identities Updated – list of all identities that have new references added during the run.
 - Contains three pieces of information.
 - Identifier – OYSTER ID of the identity that is getting new references added.
 - Reference before update – List of identity references that made up the identity in the identity input file.
 - Only listed if `ChangeReportDetail="Yes"`
 - References after update – List of identity references that make up the identity in the identity output file.
 - Only listed if `ChangeReportDetail="Yes"`

3.2 ChangeReportDetail="Off"

When the `ChangeReportDetail` is set to "Off", OYSTER tracks and provides a minimal level of detail regarding the merging and updating of identities in an .idty file.

An Identity change report with value set to "Off" is shown in Figure 10.

```

Identity Change Report.txt - Notepad
File Edit Format View Help
OYSTER Identity Change Report
Date : Jan 31, 2012
RunScript Path: IdentityUpdateRunScript.xml
RunScript Name: IdentityUpdateRunScript

Identity Change Summary Section
Count of Output Identities: 4
Count of Input Identities: 3
Count of Input Identities Updated and written to Output: 1
Count of Input Identities Not Updated and written to Output: 2
Count of Input Identities Merged: 0
Count of New Identities created: 1

Identity Change Detail Section
New Identities Created
Identifier
Z9CPBAHJ1Q5HZGPR

Input Identities Merged
Input Identifier output Identifier

Input Identities Updated
Identifier
RDNQ6KYD3X6SVXC6

```

Figure 10: Identity Change Report with ChangeReportDetail="Off"

3.3 ChangeReportDetail="On"

When the ChangeReportDetail is set to "On", OYSTER tracks and provides extra details regarding the merging and updating of identities in an .idty file. This extra detail consists of the reference level information involved in the merge, update, or identity creation. An Identity change report with value set to "Off" is shown in Figure 11.

```

Identity Change Report.txt - Notepad
File Edit Format View Help
OYSTER Identity Change Report
Date : Jan 31, 2012
RunScript Path: IdentityUpdateRunScript.xml
RunScript Name: IdentityUpdateRunScript

Identity Change Summary Section
Count of Output Identities: 4
Count of Input Identities: 3
Count of Input Identities Updated and written to Output: 1
Count of Input Identities Not Updated and written to Output: 2
Count of Input Identities Merged: 0
Count of New Identities Created: 1

Identity Change Detail Section
New Identities Created
Identifier References
Z9CPBAHJ1Q5HZGPR source2.1]

Input Identities Merged
Input Identifier output Identifier

Input Identities Updated
Identifier References before update References after update
RDNQ6KYD3X6SVXC6 source1.2|source1.4 source1.2|source1.4|source2.2

```

Figure 11: Identity Change Report with ChangeReportDetail="On"

4. Identity Merge Map

The Identity Merge Map.csv file is created during every OYSTER run. It is stored in the same directory as the Identity Change Report. This file provides a list of OYSTER IDs for identities in an input .idty file that have merged during the current OYSTER run. The file contains a list of the original OYSTER ID (Input Identity) and the ID that the identity was merged into (Output Identity). The file is Tab delimited. The Identity Merge Map.xls file can be seen in Figure 12.

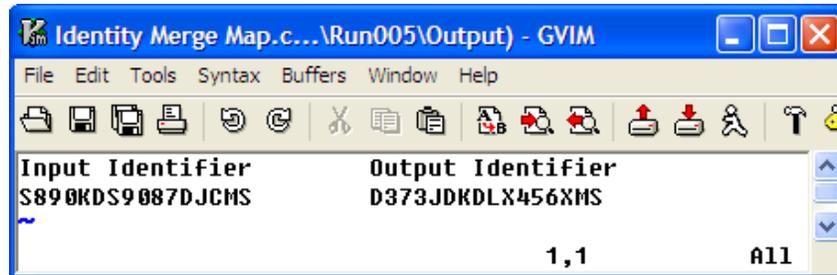


Figure 12: Identity Merge Map

5. Log File

The oyster.log file is created during every OYSTER run. The contents of the oyster.log file vary depending on the values set for the Explanation, Debug, and SS attributes of the Settings tag in the OysterRunScript.

If both the Explanation and Debug attributes are assigned the value "No" then the oyster.log file will contain only the output that was printed to the command prompt unless there are major errors in the run. This output consists of a report that is comprised of various statistics that describe the run. This report has six sections:

- Summary Stats
- Cluster Stats
- Rule Stats
- Index Stats
- Resolution Stats
- Timing Stats

Each of these sections contains various statistics. Each of these is defined in the following sections.

5.1 Summary Stats

The Summary Stats section contains five statistics:

- `Total Records Processed` - The total number of input references that were read from the input files defined in the `SourceDescriptors`.
- `Total Clusters` - The total number of resulting clusters (identities) generated by the OYATER run. (The number of resulting identities created by linking input references.)
- `Max Cluster Size` - The count of references that were linked to create the largest resulting cluster by the OYSTER run.
- `Min Clusters Size > 1` - The count of references that were linked to create the smallest resulting cluster that contains more than a single reference. (contains at least a single linked pair of references)
- `Min Cluster Size` - The count of references that were linked to create the smallest resulting cluster by the OYSTER run.

An example of this section is shown in Figure 13. This example shows that 12 references were processed into eight clusters. Three of the references were linked to create the largest cluster and the smallest cluster contains a single reference. The smallest clusters with more than one reference contain two references.

```
#####  
## Summary Stats ##  
#####  
Total Records Processed      :          12  
Total Clusters                :           8  
Max Cluster Size             :           3  
Min Cluster Size > 1         :           2  
Min Cluster Size             :           1
```

Figure 13: Summary Stats Section of Log File

5.2 Cluster Stats

The Cluster Stats section contains 17 statistics. The first statistic is:

- `Cluster Size Distribution` - Provides a Frequency listing of all the clusters in a knowledgebase grouped by the number of linked references in a cluster (identity).

This is represented by 3 columns:

- `Cluster Size` - Represents the number of linked references in the cluster
- `# of Clusters` - Represents the number of clusters which contain the number of linked references specifies in the corresponding "Cluster Size"
- `# of Records` - indicates the number of linked references that exist in clusters of a particular size. Calculated as: "Cluster Size" * "# of Clusters"

Note: Exceedingly large clusters are outliers and should be investigated as should large numbers of single clusters.

The next three statistics in this section provide information about the references and clusters located in and loaded from an identity input file.

- `Clusters loaded` - Total number of clusters loaded from the identity input file.
 - Shows a value of 0 if no identity input file was loaded.
- `References loaded` - Total number of references that comprise the loaded clusters.
 - Shows a value of 0 if no identity input file was loaded.
- `Avg # of Refs/Cluster` - This is the average number of clusters that were loaded per cluster. Calculated as "`Clusters loaded`" / "`References loaded`"
 - Shows a value of "NaN" if no identity input file was loaded. "NaN" means Not a Number and is generated when dividing 0 by 0.

The next group of statistics in this section relies on the values presented in the `Cluster Size`, `# of Clusters`, and `# of Records` columns displayed in the `Cluster Size Distribution` statistic.

- `Average Cluster Grouping` - The ACG is the average of the `Cluster Size`. This is found by summing all the unique `Cluster Size`s and then dividing by the count of the unique cluster sizes. i.e. $(1+2+3+4+5+6+7+8+9)/9 = 5$.
- `Average Cluster by Count` - The ACC is the average of the `# of Clusters` column. This is found by summing all the `# of Clusters` values and dividing by the count of the `# of Clusters` values.
- `Average Cluster Size` - The ACS is the average cluster size for the run. This is found by summing the `# of Records` values and dividing by the sum of the `# of Clusters` values.
- `Number of Duplicate Recs` - calculates the number of duplicate records found while processing the input references. This is found by taking the summation of each cluster size minus 1 times the corresponding `# of Records`.
 $\sum (\text{Cluster Size} - 1) * \# \text{ of Records}$
- `Duplication Rate` - The duplication rate is the percentage of the references that are found to be duplicates based on the identity rule set. The Calculation is:
 $1 - (\text{Total clusters} / \text{Total records})$

The remaining 8 statistics provided in this section are focused on match candidates.

- `Total Candidates Size` - The total number of Candidates that were returned by the index based on the input record set and the indexing rules.
- `Total DeDup Candidates Size` - A unique count of the `Total Candidate Size`. This is possible due to a cluster having multiple refID's (many to one relationship).

- Total # Candidates – A count of the fact that a Candidate was found for a record, i.e. input #25 returns 3 candidate records, this is counted one time.
- Avg Candidates per Input - The Avg. Candidates is the Total Candidates Size / Total # Candidates.
- Total Matched Count – Represents a count of matches that occurred between references and candidates.
- Matches per Candidates Size – Represents the percentage of matches per the full Candidates Size. Calculated as Total Match Count / Total Candidates Size
- Matches per DeDup Candidates Size - Represents the percentage of matches per the Total DeDup Candidates Size. Calculated as Total Match Count / Total DeDup Candidates Size
- Matches per Candidates - Represents the percentage of matches per the Total # Candidates. Calculated as Total Match Count / Total # Candidates

An example of the Cluster Stats section can be seen in Figure 14 and Figure 15.

```
#####
## Cluster Stats ##
#####
Cluster Size Distribution
Cluster Size      # of Clusters      # of Records
                1                5                5
                2                2                4
                3                1                3

Clusters loaded      :                0
References loaded    :                0
Avg # of Refs/Cluster :                NaN
```

Figure 14: Cluster Stats Example Part 1

```
Average Cluster Grouping      :                2
Average Cluster by Count      :                2
Average Cluster Size          :                1.50000
Number of Duplicate Recs      :                4
Duplication Rate              :                0.33333

Total Candidates Size         :                66
Total DeDup Candidates Size   :                43
Total # Candidates            :                11
Avg Candidates per Input      :                6.00000
Total Matched Count          :                4
Matches per Candidates Size   :                0.06061
Matches per DeDup Candidates Size:                0.09302
Matches per Candidates        :                0.36364
```

Figure 15: Cluster Stats Example Part 2

5.3 Rule Stats

The Rule Stats section only contains two statistics. These provide usage statistics for the rules defined in the OYSTER run.

- Number of Rules - Specifies the number of Rules defined in the OysterAttributes file.
 - Rule Firing Distribution - A frequency of the number of times a rule has fired. This is represented as corresponding columns of data.
 - Rule - the Rule number associated to the rule that fired.
 - Counts - The number of time the corresponding rule fired.
- NOTE: Only rules that fired have a count here so if a rule is never triggered it doesn't appear with a count of zero.

An example of the Rule Stats section can be seen in Figure 16.

```
#####  
## Rule Stats ##  
#####  
Number of Rules: 2  
Rule Firing Distribution  
Rule Counts  
1 1  
2 3
```

Figure 16: Rule Stats Example

5.4 Index Stats

The Index Stats section contains 17 statistics all relating to the Indexes defined by the user.

- Keys - The count of Hash Keys in the indices.
- Total tokens - The total number of RefID's held in the indices.
- Unique tokens - The unique number of RefID's held in the indices.
 - Since a single record can match multiple indexes this only counts a RefID once.
- Max tokens per key - Give the max number of RefID's (tokens) associated to a single Key.
- Min tokens per key - Give the min number of RefID's (tokens) associated to a single Key.
- Min tokens > 1 per key - Give the min number of RefID's (tokens) associated to a single Key where there is more than a single token associated to the key.
- Total tokens per key - The average number of RefID's per Hash Key.
- Unique tokens per key - The average number of unique RefID's per Hash Key.

- Total per Unique tokens - The total number of RefID's / the unique number of RefID's
- Unique per Total tokens - The unique number of RefID's / the total number of RefID's

The next three statistics are explained in detail on the following website. The concepts for each are such that it is not possible to concisely explain each. Please refer to this site for details: <http://www.tc3.edu/instruct/sbrown/stat/shape.htm>

- Skewness
 - If skewness is less than -1 or greater than +1, the distribution is highly skewed.
 - If skewness is between -1 and -½ or between +½ and +1, the distribution is moderately skewed.
 - If skewness is between -½ and +½, the distribution is approximately symmetric.
 - Bulmer, M. G., Principles of Statistics (Dover, 1979)
- Kurtosis
 - A normal distribution has kurtosis exactly 3 (excess kurtosis exactly 0). Any distribution with kurtosis $\neq 3$ (excess $\neq 0$) is called mesokurtic.
 - A distribution with kurtosis < 3 (excess kurtosis < 0) is called platykurtic. Compared to a normal distribution, its central peak is lower and broader, and its tails are shorter and thinner.
 - A distribution with kurtosis > 3 (excess kurtosis > 0) is called leptokurtic. Compared to a normal distribution, its central peak is higher and sharper, and its tails are longer and fatter.
- Excess - The Kurtosis value minus 3
 - The reference standard is a normal distribution, which has a kurtosis of 3. In token of this, often the excess kurtosis is presented: excess kurtosis is simply kurtosis-3. For example, the "kurtosis" reported by Excel is actually the excess kurtosis.
- Max key - This is the top index key in terms of size.
- Top 10 keys - The top ten index keys by descending size.
- Frequency of the Index Candidates - This is the frequency of the Index Candidates. The candidates of size zero are records that did not match anything in the index, i.e. the first record in a cluster. This statistic is represented by three corresponding columns:
 - Candidate Size
 - # of Candidates
 - # of Records

NOTE: These counts can be used to help fine tune index. If there are a large number of records in the zero bin then you are possibly missing some candidates with your rules. If there were some large groups, i.e. candidate sizes > 50 then you rules are not granular enough.

- Frequency of the Index Groups by Size – Shows the Frequency of the Index Groups sorted by size. This is represented by three corresponding columns:
 - Index Group
 - Index Size
 - # of Records

An example of the Index Stats can be seen in Figure 17 and Figure 18.

```
#####
##  Index Stats  ##
#####
Keys                :           62,403
Total tokens        :          160,696
Unique tokens       :           80,348
Max tokens per key  :              408
Min tokens per key  :                1
Min tokens > 1 per key :            2

Total tokens per key :          2.57513
Unique tokens per key :          1.28757
Total per Unique tokens :          2.00000
Unique per Total tokens :          0.50000

Skewness            :           1.07905
Kurtosis             :           1.18336
Excess               :           -1.81664

Max key              : 1234, 1235, 1236, 1237, 1238, 1239

Top 10 keys          :
408                  : 1234, 1235, 1236, 1237, 1238, 1239
396                  : 0123
296                  : 0124, 0134
288                  : 1245, 1246, 1247, 1248, [42 More]...
276                  : 0125, 0126, 0127, 0128, [13 More]...
264                  : 1123, 1223, 1233
204                  : 1124, 1125, 1126, 1127, [33 More]...
198                  : 0112, 0113, 0122, 0133, 0223, 0233
176                  : 0145, 0146
168                  : 1456, 1457, 1458, 1459, [57 More]...
```

Figure 17: Index Stats Example Part 1

Candidate Size	# of Candidates	# of Records
0	973	0
1	977	977
2	959	1,918
3	915	2,745
4	916	3,664
5	859	4,295
6	839	5,034
7	788	5,516
8	769	6,152
9	714	6,426
...		
Index Group	Index Size	# of Records
1	45,989	45,989
2	12,621	25,242
3	2,288	6,864
4	521	2,084
5	76	380
6	94	564
7	1	7
8	67	536
12	109	1,308
...		

Figure 18: Index Stats Example Part 2

5.5 Resolution Stats

The Resolution Stats section, unlike the other five sections, is conditional. It only appears if the mode of the OYSTER run is set for Identity Resolution. This section only contains a single statistic.

- `Records resolved` - This is the count of the number of records that were resolved (found) in the identity repository.

An example of the Resolution stats can be seen in Figure 19.

```
#####
## Resolution Stats ##
#####
Records resolved           : 102
```

Figure 19: Resolution Stats Example

5.6 Timing Stats

The Timing Stats section contains 11 statistics regarding the performance of the run.

- Elapsed Seconds - The elapsed time for the processing to take place for the OYSTER Run.
 - This does not include time for reading or writing identity files.
- Throughput (records/hour) - The estimated number of records that could be read in an hour if the same speed is held without change.
 - This is a realistic measure if the index is properly aligned and in Resolution or Assertion mode. For other modes, this is extremely optimistic.
- Average Matching Latency (ms) - The average amount of time it took to look up a Candidate list and check the list to find a matching record.
- Max Matching Latency (ms) - The max amount of time it took to look up a Candidate list and check the list to find a matching record.
- Min Matching Latency (ms) - The min amount of time it took to look up a Candidate list and check the list to find a matching record.
- Average Non-Matching Latency (ms) - The average amount of time it took to look up a Candidate list and check the list to find a non-matching record.
- Max Non-Matching Latency (ms) - The max amount of time it took to look up a Candidate list and check the list to find a non-matching record.
- Min Non-Matching Latency (ms) - The min amount of time it took to look up a Candidate list and check the list to find a non-matching record.
- Time process started - This is a timestamp of when OYSTER started processing
- Time process ended - This is a timestamp of when OYSTER finished processing
- Total elapsed time - This is the start time - the end time. This gives you the entire time for the OYSTER run to occur.

An example of the Timing Stats section can be seen in Figure 20.

```
#####
## Timing Stats ##
#####
Elapsed Seconds           :                7
Throughput (records/hour) : 2,215,028.57143
Average Matching Latency (ms) : 0.326910
Max Matching Latency (ms)   :                33
Min Matching Latency (ms)  :                0
Average Non-Matching Latency (ms) : 0.13838
Max Non-Matching Latency (ms) :                7
Min Non-Matching Latency (ms) :                0

Time process started at 2013-02-09 13.49.05
Time process ended at 2013-02-09 13.49.12
Total elapsed time 0 hour(s) 0 minute(s) 7 second(s)
```

Figure 20: Timing Stats Example

6. Extended Log File

The `oyster.log` file with the `Debug` and `Explanation` value is set to "Yes" displays a lot more information than when they are turned off. The extended log file contains all the information that was contained in the log file shown in the previous sections but also includes the following additional information:

- Link index
- EntityMaps
- Identity Output
- ValueIndexes
- Comparison Matrixes
- Masks

These log files are extensive as they map every comparison and it is not feasible to show a full extended log file here.

VI. Other OYSTER Functionality

This section outlines functionality that is available in OYSTER but is not integrated into one of the XML files.

1. KILL Thread

In Oyster v3.2 and earlier, when a user initiates a kill operation on an OYSTER run, the program simply terminates and all current state information is lost from any output buffers. Note: in v3.2 and earlier, kill operations are initiated via *ctrl+c* or by closing the run window. The purpose of the Kill Thread, introduced in OYSTER v3.3, is to allow OYSTER the ability to retain the current state of the run when a Kill is initiated. This will also provide the user with a direct OYSTER implemented method to Kill the processing of a run. This allows the user to gracefully exit the process and review the results of the run up till the kill point.

The Kill process depends on two things. The first is a new thread that is initiated during the start of the OYSTER run that repeatedly “polls” the directory in which the initiating RunScript resides and checks for the existence of a “kill.txt” file. The second is the kill.txt file itself. The user must create this file when they decide to kill the process.

No additional options are required in any of the XML scripts to initiate the Kill Thread. The kill.txt file that invokes the Kill process is an empty text file that is placed in the directory in which the initiating RunScript resides. The file name must be “kill.txt” as this is what the Kill Thread is checking for.

The Kill Thread polls the directory every 20 seconds so there may be a slight delay from the time the kill.txt file is placed in the directory and the current run is stopped.

2. 90% memory Warning

The purpose of the 90% Memory Warning is to alert users that OYSTER has been over 90% memory capacity for more than 10 minutes. This warning is for informational purposes only and does not pause or interrupt the current OYSTER run.

The 90% Memory Warning is a separate thread that is initiated with an OYSTER run. It checks the current allocated Heap size and the current Used Heap size. It calculates “Used Heap size”/“Allocated Heap size” and if the resulting percentage is larger than 90% for 10 minutes it will throw a warning. Java always keeps a larger Allocated Heap then the Used Heap and if the Used Heap needs more space Java will increase the size of the Allocated Heap to compensate for the required space. If the Used Heap size is greater than 90% of the Allocated Heap for longer than 10 minutes, this means that Java no longer has the ability to inflate the Allocated Heap since it is already inflated to the Max Heap size allowed. No additional options are required in any of the XML scripts to initiate the thread that handles the memory monitoring or the Memory Warning.

VII. OYSTER User-Defined Inverted Index

A new feature has been added to OYSTER Version 3.3 that allows the user to define an inverted, match-key index. The User-Defined Index (UDI) feature can significantly reduce run-time. However, a poorly designed UDI can significantly increase false negative error rates. This section defines the concepts involved for UDI and details the efforts what must be used to properly align the UDI with the defined match rules.

1. Inverted Index

An inverted index is a lookup table that allows a user to quickly find all of the records that have a common value. For example, all of the records that the value “JOHN” for the first name. The principle of the inverted index can be applied to entity resolution (ER) algorithms as technique for reducing the number of comparisons needed to resolve a set of input references.

In the case of ER, the common value is a match-key. When an input reference is read into the OYSTER engine for processing, the index quickly locates all previously processed records that share the same match key as the input reference

2. Match Key

A match key is a string value generated by applying algorithms (hash functions) to the attribute values in a reference. In the OYSTER engine, algorithms are first applied to selected attribute values then the final match key is formed by concatenating the algorithm outputs in a fixed order. A particular configuration of attributes and hash algorithms comprises a hash-key generator. Figure 21 shows a hash-key generator that selects attributes A, B, and D of an input reference.

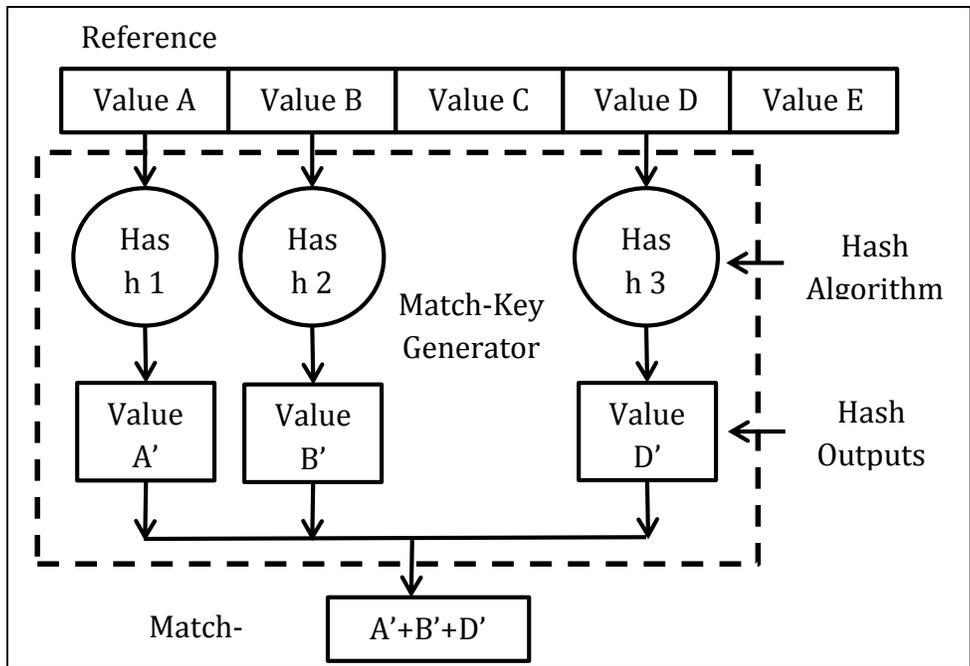


Figure 21: Schematic of Match-Key Generator

2.1 Match Key Example 1

Attributes - FirstName, LastName

Hash Algorithms

First Name: LeftSubstring(1)

Last Name: LeftSubstring(5)

Hash Values

FirstName: "PHILIP" → LeftSubstring(1) → "P"

LastName: "DOE" → LeftSubstring(5) → "DOE"

Match Key = "P"+"DOE" = "PDOE"

3. Index Operation

The index is an inverted index on the match-keys generated by each input reference as it is processed. As each reference is read into the system, the attribute values are input into a match-key generator. The resulting match-key is then used to lookup all other previously processed references that also generated that same match key. The previously processed references that have the same match-key then become the candidates to test against the input records by the match rules. It is important to note that these are the ONLY candidates to be tested for matching. After the input record has been matched to all of the candidate records, the input record is also inserted into the index (lookup table). Figure 22

shows the first step where the match-key is generated and used to lookup matching candidates.

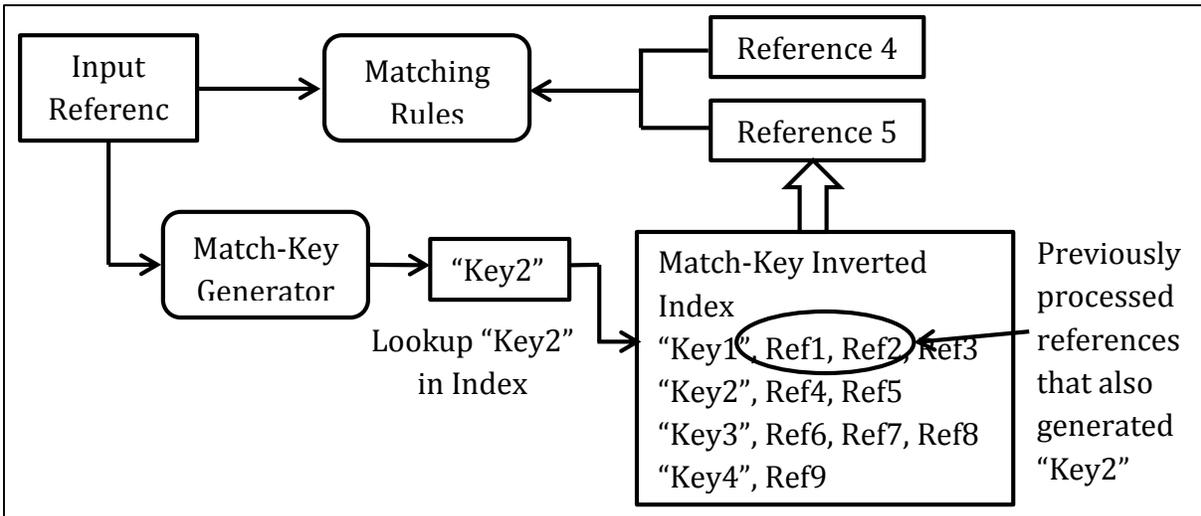


Figure 22: Match-Key Lookup for Match Candidates

The same process show in Figure 22 is also shown in Figure 23 using the match-key generator of Example 1. Note that in Figure 23, the input reference "PHILIP DOE" generates the match-key "PDOE" that returns three matching candidates. Which of these three actually match the input reference will depend on the rules. If the rules require that first and last names must be exact, then only the candidate reference R8 would be a match. On the other hand, if the rules allow the first name to match by their Soundex values, then both candidate references R4 and R8 would be a match.

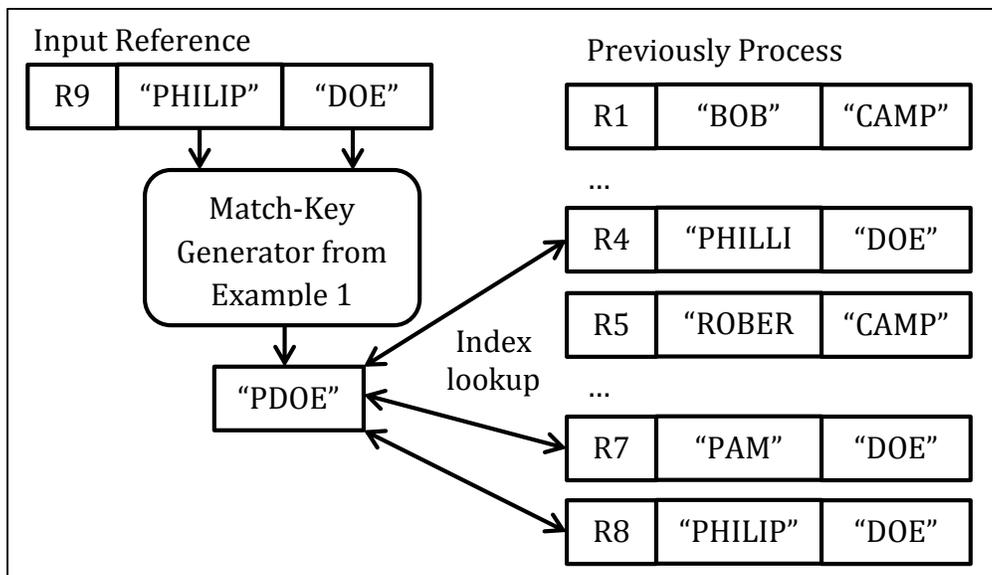


Figure 23: Candidates Returned by Match-Key "PDOE"

4. Alignment of Index with Match Rules

It is important to understand that the index generators work independently of Match Rules. In order to get accurate ER results, the match rules and the index generators must work together. The match rules can only compare an input record to the set of candidate references returned by the index. Even though an input is capable of matching a previously processed reference, that match will only be known by the system if the matching reference is found by the Index, i.e. the input and previously processed reference generate same match key.

<p>Proper Alignment</p> <p>Match Rule: First Name: Soundex Last Name: Exact</p> <p>Index: LeftSubstring(FirstName, 1) LeftSubstring(LastName, 5)</p>	<p>Misalignment</p> <p>Match Rule: First Name: NickName Last Name: Exact</p> <p>Index: LeftSubstring(FirstName, 1) LeftSubstring(LastName, 5)</p>
--	---

Figure 24: Examples of Good and Poor Rule-to-Index Alignment

Figure 4: Examples of Good and Poor Rule-to-Index Alignment

The Rules and Index in Figure 24 labeled “Proper Alignment” are in alignment because if two references match on first name by Soundex, then the two names must begin with the same first letter. This means that the hash function that extracts the first left character of the name will also be the same since the Soundex algorithm does not change the first character of the input string. Therefore if two names match by Soundex then it follows that they must begin with the same letter. Similarly if two last names are the same (Exact match), then the hash function that extracts the first five letters of the name will also give the same value.

On the other hand, the Rules and Index shown in Figure 24 and labeled “Misalignment” fail to align properly. The problem is that the first name comparator is by nickname or alias. Two nicknames may not begin with the same first letter. For example, a reference with first name “ROBERT” and last name “SMITH” will generate the match key “RSMITH” by this index. Another reference with first name “BOB” and last name “SMITH” will generate the match key “BSMITH”. However, these two references will match since “BOB” is a common nickname for “ROBERT”, but these references generate different match keys. Therefore there is a rule-to-index misalignment.

5. Index Recall and Precision

Two measures often applied to an index are its recall measure and its precision measure. Recall is the percentage of record pairs that match by one of the rules and that also

generate the same match key by one of the indices. Proper alignment is achieved between rules and indices when the indices have 100% recall. On the other hand, precision is the percentage of record pairs generating the same match key that also match by one of the rules.

When an index has less than 100% precision, it means that it will return some records to the rules for match comparison that will not match. The lower the precision the more unnecessary effort that will be expended by the process in comparing an input record to previously processed records that will not match by any of the rules. It is the recall measure that has the most impact the accuracy of the ER process. When the recall is less than 100%, it means that the indices will fail to find some matches between records that are present in the data. In this respect, recall is the more important consideration for accuracy of the ER process, whereas precision the more important consideration for the efficiency (performance) of the ER process. The general rule for index design is to first achieve 100% recall (alignment), then work to increase precision while maintaining 100% recall.

5.1 Alignment Scenarios

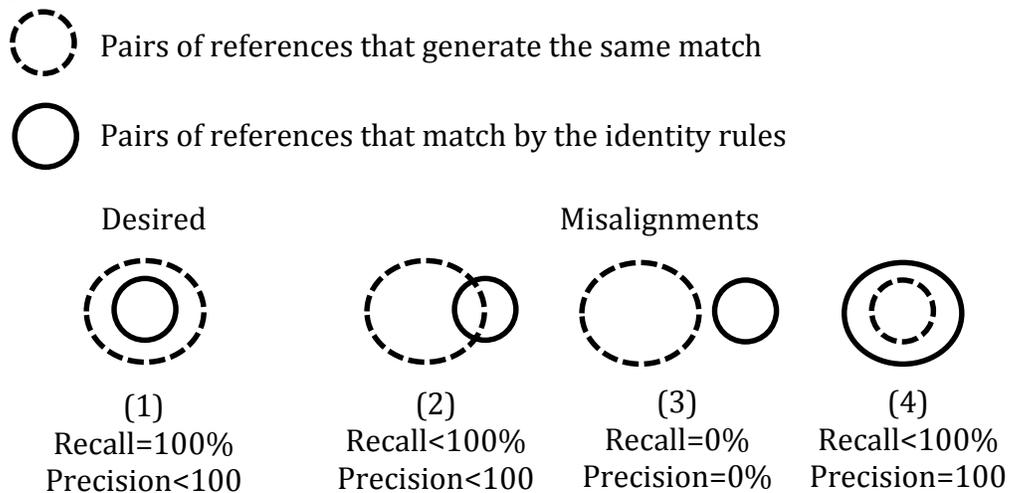


Figure 25: Diagrams of Rule-to-Alignment Scenarios

The circles labeled (1) in Figure 25 illustrate 100% alignment. Each pair of references that match by the rules also generates the same match key. The circles labeled (2) represent a typical misalignment where some, but not all pairs of references that match by the rules generate the same match key. The circles labeled with (3) represent total misalignment where none of the matching pairs of references generate the same match key and none of the pairs generating the same key will match. The scenario illustrated by the circles labeled (4) is one where every pair of references that produce the same match key will

match by one of the rules (100% precision), but some pairs that match by the rules do not generate the same match key.

5.2 Index Fewer Attributes Strategy

It is not necessary to index every attribute used in a rule. For some situations it may be better for the match-key generator to use fewer inputs in order to attain alignment. The left-most box of Figure 26 shows the same misalignment of match rules and index as in Figure 24. The right-most box of Figure 26 shows how this misalignment can be corrected by dropping the hash on the FirstName attribute and only generating the match-key from the LastName attribute.

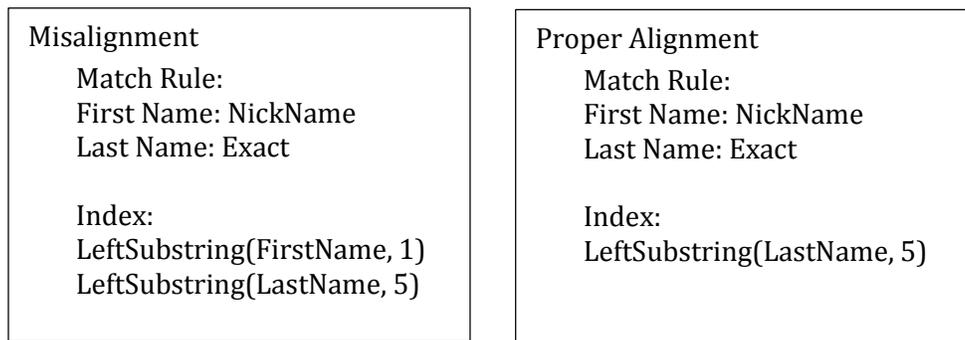


Figure 26: Indexing on Fewer Attributes to Attain Alignment

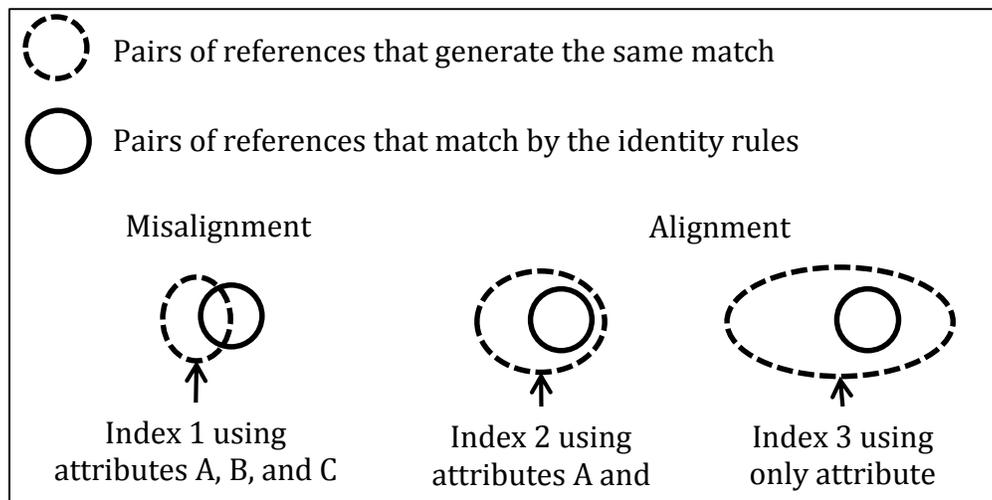


Figure 27: The Effect of Removing Attributes from the Match-Key Generator

As illustrated in Figure 27, for a given set of attributes and their corresponding hash algorithms, it will always be true deleting one or more of the terms (i.e. an attribute and its hash algorithm) will never decrease the number of pairs of references generating the same match-key, and in most cases will increase that number.

Although indexing on fewer attributes may help attain rule-to-index alignment, the fact that it returns more candidates for matching may result in a performance problem. Another strategy that can help to address the performance issue while still attaining alignment is to implement more than one index.

5.3 Multiple Index Strategy

Most entity resolution systems that use match-key indices will also allow for defining more than one index. When more than one index is defined, the candidate list returned to the match rules is the union of all of the candidates returned by each of the indices. Multiple indices are often needed when different rules compare different sets of attributes.

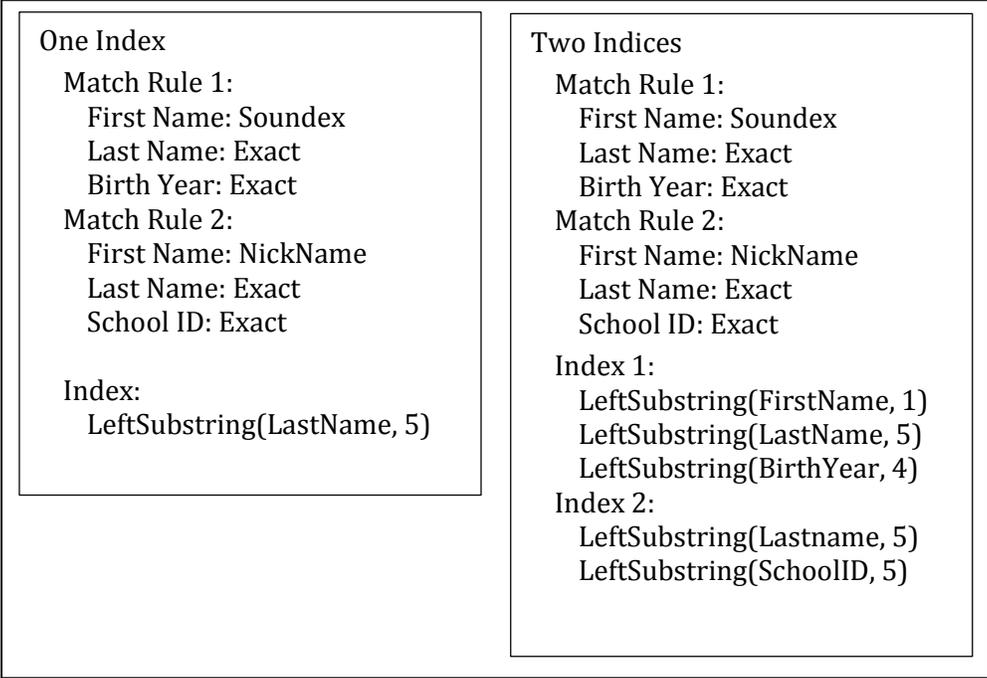


Figure 28: Same Rule Set with One Index and Two Indices

Figure 28 shows a set of two matching rules on four attributes. Even though both rules match on First Name, the comparators are different. Because the second rule compares First Name by NickName, indexing on First Name would cause misalignment. The only alignment solution using a single index would be to simply index on Last Name. This means that for every input reference, the index would return a list of all previously processed references that share the same first five characters of the last name.

The two-index solution shown on the right would provide better performance while still maintaining alignment. Index 1 is designed to be in alignment with Match Rule 1 while taking advantage of the fact that names matching by Soundex must also agree on the first character. It also includes Last Name and Birth Year since these are Exact match attributes in Rule 1. Similarly Index 2 is designed to align with Rule 2. Even though it does not use

First Name because of the NickName comparator, it does use the School ID that is in Rule 2, but not in Rule 1. Since each rule is in alignment with at least one index, the entire set of rules is alignment. Furthermore the combined set of candidates returned for each input record by the two indices will no larger than the set of candidates returned by the single index, and in most cases will be smaller.

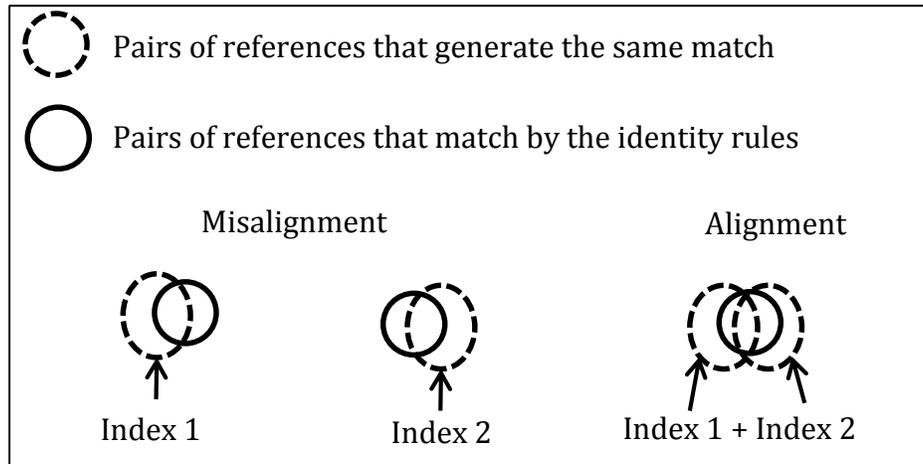


Figure 29: Combined Effect of Two Indices

The effect of combining multiple indices is illustrated in Figure 29. Whereas neither index by itself is in alignment with the rules, together they can create alignment. Any overlap between the indices (i.e. same candidate references returned by more than one rule) can easily be removed prior to matching, so overlaps between multiple indices does not add to the number of comparisons that must be made.

6. The Alignment Process

There are three steps in the alignment process

1. Rule Analysis
2. Index Design
3. Alignment Validation

6.1 Rule Analysis

Rule Analysis is best done using a table or spreadsheet. Each rule makes one row in the table and each column is for an identity attribute. The cells in the table show the type of comparison that is required by the rule for that attribute.

Table 3 shows a table for 17 identity rules operating on 11 identity attributes labeled A1 to A11. The rules and attributes shown in Table, and the average number of candidates returned by various index schemes for these rules are based on tests performed on actual school enrollment information.

Table 3: Showing 17 Rules and 11 Identity Attributes

Rule	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11
1	EXACT		EXACT	EXACT							
2	EXACT		EXACT		EXACT						
3	EXACT	EXACT	EXACT		EXACT						
4	LED(0.80)	EXACT	EXACT	EXACT		EXACT				EXACT	
5	EXACT	EXACT	LED(0.80)	EXACT		EXACT				EXACT	
6	LED(0.65)	EXACT	EXACT	EXACT	EXACT						
7	EXACT		LED(0.55)		EXACT	EXACT	EXACT				
8	EXACT		LED(0.80)	EXACT	EXACT						
9	EXACT		EXACT		EXACT	EXACT					
10	EXACT	EXACT	EXACT			EXACT	EXACT	EXACT			
11	EXACT			EXACT	EXACT	EXACT	EXACT	EXACT			
12	EXACT	EXACT		EXACT			EXACT	EXACT	EXACT		
13	EXACT	EXACT		EXACT	EXACT						
14	LED(0.90)	LED(0.87)	EXACT	EXACT	EXACT						
15	LED(0.85)	EXACT	EXACT	EXACT							
16	EXACT	EXACT		EXACT			EXACT				
17	EXACT		EXACT	EXACT							EXACT

Although most attributes in this table require an “EXACT” match, some allow an approximate match by the Levenshtein Edit Distance function denoted by LED(0.xx) where the number 0.xx in parentheses indicates the matching threshold. For example, Rule 5 requires the Levenshtein similarity to be 55% or higher between the values of Attribute A3.

6.2 Index Design

The goal of index design is two-fold

1. (Alignment) Define one or more indices so that whenever two records match by one the rules, both records will generate the same match key by at least one index
2. (Reduction) Define each index in such a way that a minimal number of records will generate the same match key.

The ideal situation would be to define one index for each rule that only generates the same match key for records that match by that rule. For certain rule sets this is possible by simply translating each rule term-by-term into a corresponding index. For example, suppose that a system only used Rules 1 and 2 shown in Table 3. Since Rule 1 requires an exact match on Attributes A1, A3, and A4, the index for this rule would simply be generate a match key by concatenating these three values, i.e. the match key would be A1+A3+A4. Similarly, the index for Rule 2 would only need to concatenate Attributes A1, A3, and A5, i.e. the match key would be A1+A3+A5. In this case, for each rule, every record that matches

by the rule would create the same match key, and any two records creating the match key would match by the rule. An OYSTER implementation of these two rules and indices is shown in the Figure 30.

```

<IdentityRules>
  <Rule Ident="R1">
    <Term Item="A1" MatchResult="Exact">
    <Term Item="A3" MatchResult="Exact">
    <Term Item="A4" MatchResult="Exact">
  </Rule>
  <Rule Ident="R2">
    <Term Item="A1" MatchResult="Exact">
    <Term Item="A3" MatchResult="Exact">
    <Term Item="A5" MatchResult="Exact">
  </Rule>
</IdentityRules>
<Indices>
  <Index Ident="P1">
    <Segment Item="A1" Hash="Scan(LR,All, 0, KeepCase, SameOrder)">
    <Segment Item="A3" Hash="Scan(LR,All, 0, KeepCase, SameOrder)">
    <Segment Item="A4" Hash="Scan(LR,All, 0, KeepCase, SameOrder)">
  </Index>
  <Index Ident="P2">
    <Segment Item="A1" Hash="Scan(LR,All, 0, KeepCase, SameOrder)">
    <Segment Item="A3" Hash="Scan(LR,All, 0, KeepCase, SameOrder)">
    <Segment Item="A5" Hash="Scan(LR,All, 0, KeepCase, SameOrder)">
  </Index>
</Indices>

```

Figure 30: OYSTER Implementation of Rules 1 and 2 of Table 1 with Index

Note that the hash generator SCAN produces the equivalent of an EXACT match with the settings shown in Figure 1Figure 30. Here the SCAN is set to scan the string from left-to-right (Direction parameter set to "LR"), extract all of the characters in string (Character Type parameter set to "All"), scan the entire length of the string (Length set to "0"), and keep the characters in the same order (Ordering parameter set to "SameOrder"). However, for many rule sets is not possible because of comparators that perform approximate matching. Most comparators are built to allow for some level of variation between the values being compared yet still considering the values to be (approximately) the same. In many cases this is done by reducing both values the common third value. Even for exact comparison between strings, there is often consideration for ignoring differences between corresponding letters due to case, i.e. upper case versus lower case. In OYSTER this corresponds to the "ExactIgnoreCase" comparator or the "SCAN" comparator using all upper case conversion. For example, this would map the strings "John", "john", and "JOHN" all to the common value string of "JOHN" before making the comparison.

6.2.1 Match Key Comparators versus Similarity Comparators

Each hash algorithm in a match key generator is designed as a function which takes as its argument a single attribute value and maps it (transforms it) to a single output value. On the other hand, rule comparators take a pair of attribute values and map it a Boolean value, either True or False. However, as described earlier, many rule comparators such as the “ExactIgnoreCase” comparator work on the same principle as a match key function. When a pair of attribute values is input to the comparator, it first operates separately on each value to produce two outputs. If the two outputs are the same, then the comparator value for the pair is True, else the comparator value is False. For this reason, comparators of this type are called *match key comparators*.

The simplest type of match key comparator is the EXACT each value of the pair is mapped to itself. Another type of match key comparator is the Soundex comparator primarily used for strings represented a person’s name. The Soundex comparator maps each name string to a 5-character string that starts with the first letter of the input string and is followed by four decimal digits. The four digits are derived according to the Soundex Algorithm. If both input name strings produce the same 5-character output string, then the Soundex comparator produces a True value. For example, the names “Phillip” and “Philip” product the same Soundex value of “P410”.

Matching rules that use only match key comparators can be translated directly in an index by a series of hash functions that correspond to the match key comparator as was illustrated in Figure X. Unfortunately not all comparators operate in this way. Many comparators are a type called *similarity comparators*. Similarity comparators generate a numerical value that represents some measure of the similarity between two attribute values. These comparators cannot operate on each attribute value independently, but must consider both values to product a similarity measure. A primary example is the Levenshtein Edit Distance (LED) algorithm that measures the similarity between two character strings. Unlike Soundex, it does not make sense to talk about the LED of a single string. The LED can only be generated between two strings where it is based on the number simple character transformations that are required to convert one string into the other string. For example if one value is “John” then it could be transformed into each of the values “Jon”, “Jhn”, or “ohn” with a single deletion transformation. This makes “John” 75% similar to each of these three other strings (one change out of four characters). However, there is no function that can operate on each of these strings independently and produce a single value that will predict that it will be 75% LED similar to each of the other strings. The similarity can only be known when both strings are present and are processed together by the LED algorithm.

6.2.2 Balancing Alignment with Reduction

As long as each rule has at least one match key comparator, then it will always be possible to create a set of indices that have proper alignment. For example, each of the 17 rules in Table 3 uses at least one Exact comparator. In fact, Table 3 shows that every rule requires either an exact match on A1 or an exact match on A3. Therefore one possible design to index the rules in Table 3 would be two indices where the first index uses only the value of A1 and the second index uses only the value A3. This would satisfy the alignment condition in that every pair of records that matches by one of the rules, but also have to have either the same value for A1 or the same value for A3.

Whether or not this would be a practical index design will depend on the number of records being processed. The second constraint of index design is reduction. For small sets of records, this may not be a problem, but for large datasets there may be so many records that have the same value for A1 or the same value for A3, that the number of comparisons performed by the rules will be too large, and system performance will be affected.

For example, suppose that the system implementing the 17 rules of Table 3 is able to make 200,000 comparisons per second for each rule. Assuming the match rate is low then a worst case estimate is that each input record has to be compared to every candidate record. If this system runs without an index then each input record will have to be compared to each previously processed record (assuming the One-Pass algorithm) by each of the 17 rules, or $17 \times N \times (N-1) / 2$ for N input records. If N is relatively small, say 10,000 records, then the total run-time, even without an index will only be about 142 minutes or 2 hours and 22 minutes.

Now suppose that the two index design (A1 and A3) is implemented in the system. The reduction in comparisons will depend on the average number of records that generate the same match keys for these indices. With the index in place, each input record will only be compared to all previously processed records that generate the same match key as the input record by either of the two indices. If a set of indices on average returns M candidates for comparison, then the total comparison effort for the system with 17 rules to process N input records is approximately $17 \times N \times M$ comparisons.

Even though there can be some overlap between the records that generate the same match key for both by both indices, for worst case assumption that can be ignored. Therefore assuming on that on average the A1 index returns 200 candidate records and the A3 index returns 400 records, or 600 candidates combined, the total run-time for 10,000 input records can be dramatically reduced to approximately 8.5 minutes.

However, if the number of input records is increased to 1,000,000 records, then the run-time using the two-index design will increase to more than 14 hours. In this case the

reduction afforded by the simple two-index design may no longer be acceptable. One analysis technique that can be used to estimate the level of index reduction is to load in the input data into a database, then translate each index into an SQL query to profile the counts of duplicate match keys that each index will generate. In addition, some systems may provide the user with information on index performance. For example, OYSTER provides information in its standard log file about the average number of tokens (records) per match key, the 10 match keys with the largest number of tokens, and other helpful index statistics.

In general, the more attributes that are concatenated to produce a match key, the fewer records that produce the same match key. At the same time, when there are many rules such as in Table 3, designing indices that use more attributes per index may require adding new indices in order to maintain proper alignment.

Table 4: Alignment Pattern of 4 Index Design

Rule	A1+A4	A1+A5	A3+A4	A1+A2+A7
1	X		X	
2		X		
3		X		
4			X	
5	X			
6			X	
7		X		
8	X	X		
9		X		
10				X
11	X	X		
12	X			X
13	X	X		
14			X	
15			X	
16	X			X
17	X		X	

Table 4 shows an alternative 4-Index design that aligns with the 17 rules of Table 1. An “X” in a cell of Table 4 means that the rule indicated by the cell row is in alignment with the index given by the cell column. Three of the four indices each hash two of the attributes, A1+A4, A1+A5, and A3+A4. These three indices are aligned with all of the rules except for Rule 10. A fourth index based on A1+A2+A7 is added to cover this rule and assure rule-to-index alignment. Although there are more indices in this scheme, the combined number of candidates returned by this design is approximately 25 records per match key. Applying the same calculation as before, the run-time for an input file of 1,000,000 records will be

reduced from 14 hours using the previous design to approximately 35 minutes using the design shown in Table 4.

Another factor to consider is that more indices will increase the storage requirements plus some additional overhead in performance for multiple lookups. However, in general these are minor in consideration of the much larger reduction that can be obtained.

Given the consideration of alignment and reduction, index design may have an impact on rule design. For example, if a proposed rule uses only similarity comparator such as LED or Q-gram, then it may not be possible to design and index that assures alignment. In this case it may be prudent to break the single rule into multiple rules where each rule has at least one match key comparator that can be index. Another strategy is to experiment and see if similarity comparators can be replaced by match key comparators. For example for name values, it may be that the Levenshtein comparator can be replaced with the Soundex, NYSIIS, or other match key name comparators without a significant loss in accuracy.

In some situations performance considerations may dictate that 100% alignment is not possible or practical. Nevertheless, every attempt should be made to attain alignment, or at least to have the highest level of alignment possible.

6.3 Alignment Validation

The final step in index design is to validate the rule-to-index alignment. The best way to test alignment is to run a manageable subset of the records without indexing, then run the same subset with the index. If the rules and indices are in alignment the results from both runs should be identical. Of course if all of the input records could be run in a manageable amount of time, there is no need to index in the first place, so validation is always carried out using a test set. This leaves open the possibility that the validation may not extend to the entire input set because there is always the possibility that the records for which a misalignment occurs may not be present in the test set. As a matter of best practice, the test set should be selected in such a way that every match rule fires at least one time.

VIII. Error Messages

Common Errors

1. `###ERROR: Reference Items AttributeItem not an Attribute.`
 - a. Verify the Item value in the OysterAttributes.xml file matches the value assigned to Attribute in the OysterSourceDescriptor file
 - b. Verify the absolute path to the OysterAttributes.xml file is specified correctly in the OysterRunScript.xml file.
2. `###ERROR: Reference Items and Rules do not match.`
 - a. Verify the value assigned to Name for each Item specified in the OysterSourceDescriptor matches exactly the values assigned to Item in the matching rules defined in the OysterSourceDescriptor
3. `###ERROR: Rule Item: AttributeItem not an Attribute`
 - a. Verify that the attributes in the Item attribute in the rules match the attribute names defined by the `<Attribute>` tag
4. `###ERROR: CompareTo: AttributeItem not an Attribute`
 - a. Verify that the attributes in the CompareTo attribute in the rules match the attribute names defined by the `<Attribute>` tag
5. `###ERROR: RunMode invalid RunMode.` or `###ERROR: Invalid RunScript Mode.`
 - a. Verify the `<RunMode>` value defined in the RunScript is one of the 8 valid RunModes accepted by OYSTER.
6. `###ERROR: Invalid RunScript Name, does not match the external name.`
 - a. Verify the RunScriptName value in the `<Settings>` tag of the RunScript match the file name of the RunScript without the .xml extension.

Figures

Figure 1: Delimited Text Input.....	44
Figure 2: Source Descriptor for a Delimited Text File	45
Figure 3: Fixed Width Text Input.....	45
Figure 4: Source Descriptor for a Fixed Width Text File	46
Figure 5: Source Descriptor for accessing an Access DB Table.....	48
Figure 6: Source Descriptor for accessing a MySQL DB Table.....	49
Figure 7: Source Descriptor for accessing SQL Server Table	50
Figure 8: Example Identity Document.....	56
Figure 9: Example Link index.....	57
Figure 10: Identity Change Report with ChangeReportDetail="Off"	60
Figure 11: Identity Change Report with ChangeReportDetail="On"	60
Figure 12: Identity Merge Map	61
Figure 13: Summary Stats Section of Log File.....	62
Figure 14: Cluster Stats Example Part 1.....	64
Figure 15: Cluster Stats Example Part 2.....	64
Figure 16: Rule Stats Example	65
Figure 17: Index Stats Example Part 1.....	67
Figure 18: Index Stats Example Part 2.....	68
Figure 19: Resolution Stats Example.....	68
Figure 20: Timing Stats Example	69
Figure 21: Schematic of Match-Key Generator.....	73
Figure 22: Match-Key Lookup for Match Candidates.....	74
Figure 23: Candidates Returned by Match-Key "PDOE"	74
Figure 24: Examples of Good and Poor Rule-to-Index Alignment	75
Figure 25: Diagrams of Rule-to-Alignment Scenarios	76
Figure 26: Indexing on Fewer Attributes to Attain Alignment.....	77
Figure 27: The Effect of Removing Attributes from the Match-Key Generator	77
Figure 28: Same Rule Set with One Index and Two Indices	78
Figure 29: Combined Effect of Two Indices	79
Figure 30: OYSTER Implementation of Rules 1 and 2 of Table 1 with Index	81