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

OYSTER Release 3.6 Reference Guide

Document Draft: 3.0, 2018-08-20

Authors:

Fumiko Kobayashi, Sapna Srimal, John R. Talburt

Revision History

Version	Date	Prepared By	Reason for Update
1.1	11/17/2010	Fumiko Kobayashi	Added Output section Modified the section heading to contain defining numbers Removed redundant information Added Introduction
1.2	12/01/10	Fumiko Kobayashi	Modified to reflect new XML requirements for OYSTER v2.4 Expanded Log section of the Output section
1.3	2/4/2011	Fumiko Kobayashi	Modified document to reflect new changes to Oyster V2.6
1.4	2/27/2011 3/1/2011	Fumiko Kobayashi	Added Comparator section Modified document to reflect new changes to OYSTER V3.0
1.5	5/2/2011	Fumiko Kobayashi	Modified document to reflect new changes to OYSTER V3.1
1.6	6/14/2011	Fumiko Kobayashi	Updated Output section and RunScript section
1.7	6/22/2011	Fumiko Kobayashi	Removed References to RunScriptName
1.8	10/28/2011 02/01/2012	Fumiko Kobayashi	Modified document to reflect new changes to OYSTER V3.2
1.9	02/24/2012 04/18/2012	Fumiko Kobayashi	Added clarification regarding discrepancies in Change Report between engines.
1.10	08/24/2012	Fumiko Kobayashi	Added UDI, CAC, KILL, and new comparators.
1.11	12/02/2012 03/18/2013	Fumiko Kobayashi	Updated UDI and Log File sections Update to reflect OYSTER v3.3 change
1.12	2017-08-21	John Talburt	Start of update to Version 2.4 Update SCAN for 0 length
2.0	2018-04-26	Sapna Srimal	Start of update to Oyster Version 3.5 code, describe DataPrep and Scoring rules.

2.2	2018-05-28	John Talburt	Edit of OYSTER function Appendix and insertion of DataPrep and Scoring Rules
3.0	2018-08-16	John Talburt	Update to coincide with OYSTER Release 3.6
3.0	2018-08-20	John Talburt	Change Version to Draft, Update TOC

Table of Contents

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 ≈ 11.9 megabytes
    - Function:
        - Defines the max size of each generated log file.
    - Defautls 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 <EREngine> *Tag*

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

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

The EREngine 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 Attributes 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" DataPrep="code" Similarity="code" CompareTo="list"/>
```

The `Term` tag has up to four attributes.

- `Item` – (REQUIRED)Assigned value must correspond to the value of one of the `Item` attributes defined in the `Attributes` section.
- DataPrep –(OPTIONAL) Oyster function used to transform the Item values before comparison is made by the Similarity function.
    - See Appendix A for complete list of Oyster functions available for DataPrep. These functions are designated as DataPrep functions in Appendix A.

- **`Similarity`** (REQUIRED)– Oyster function used to determine the similarity between the Item values. In previous versions of Oyster this attribute was named "MatchCode". You can still use MatchCode, but Similarity is now the preferred attribute name.
    - See Appendix A for complete list of Oyster functions available for Similarity. These functions are designated as Similarity functions in Appendix A.

- `CompareTo` – (OPTIONAL) 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 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 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 element. The hash values will be concatenated in the same order as elements occur in the <Index> statement.

Each 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 0 and 30

- If the length is set to 0, SCAN will return all characters that meet the specified criteria. If the length is set to a value greater than zero, then SCAN will return up to the number of characters specified. If there are fewer characters than the length specified, it will pad the results with the asterisk (*) character.

- `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)	"+501-555-1234"	"5551234"
Scan(RL, DIGIT, 7, KeepCase, L2HKeepDup)	"+501-555-1234"	"1234555"
Scan(RL, DIGIT, 7, KeepCase, L2HDropDup)	"+501-555-1234"	"**12345"

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>
```

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.

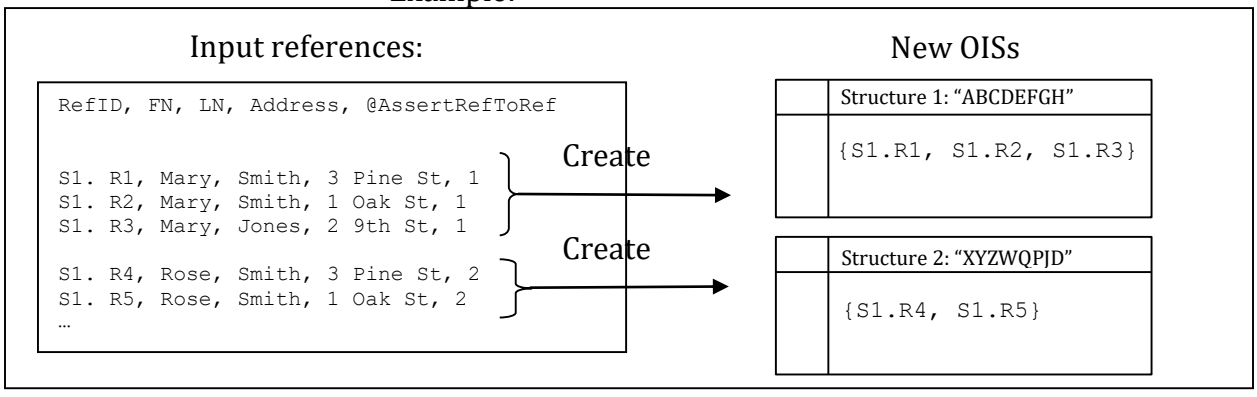
The `Item` tag has three different formats. These formats are directed by the value assigned to the `Type` attribute discussed previously in the `<Source>` Tag section.

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.

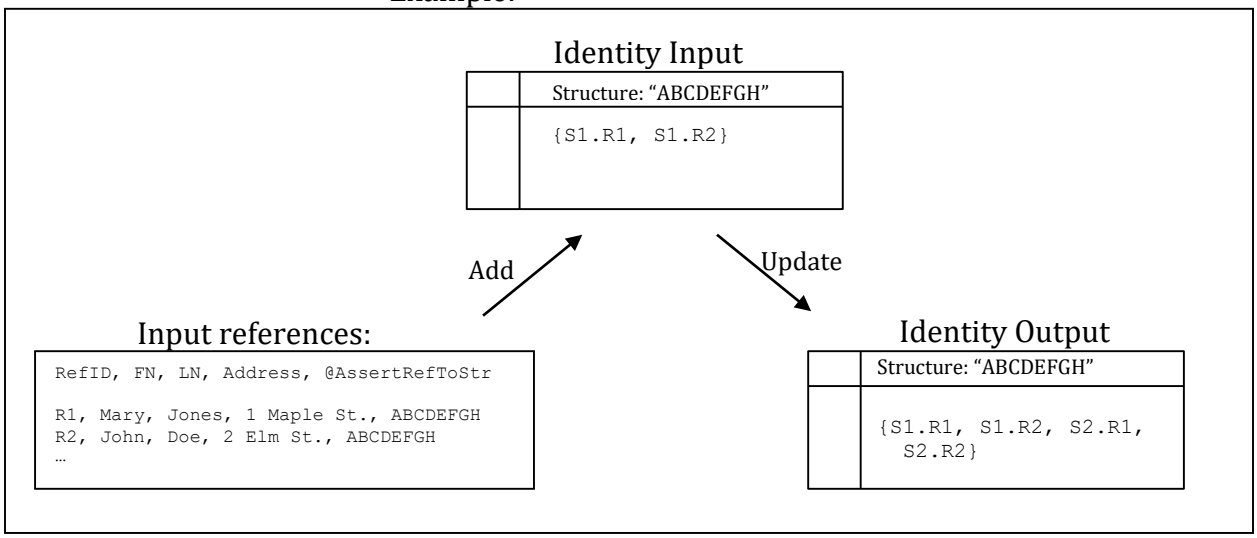
- `Name` – User defined value that names the attribute
- `Attribute` - values must correspond directly to the Item values in the OysterAttributes.xml file
    - OYSTER Keywords
        - `@RefID` - is used to inform OYSTER what field in the data is the unique keyed field
            - **NOTE: All RefIDs 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:



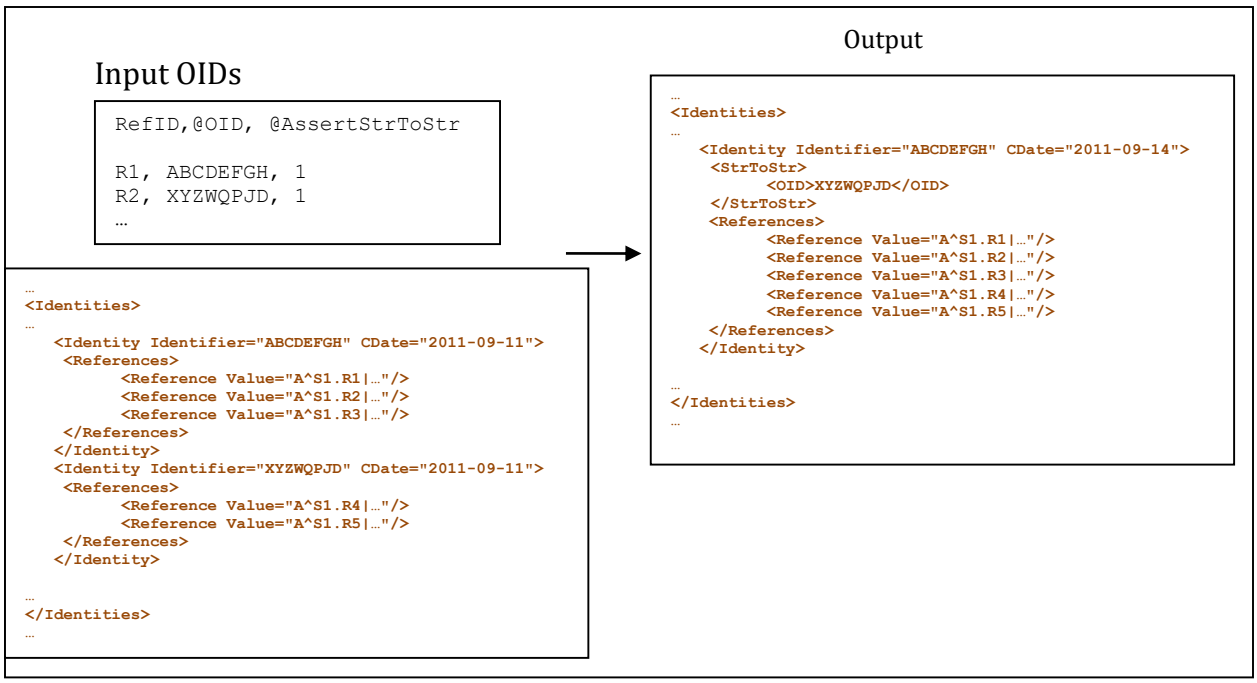
- `@AssertRefToStr` – is used when performing a Reference to Structure assertion run. This keyword informs OYSTER that the reference should be merged into the structure that matches the

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

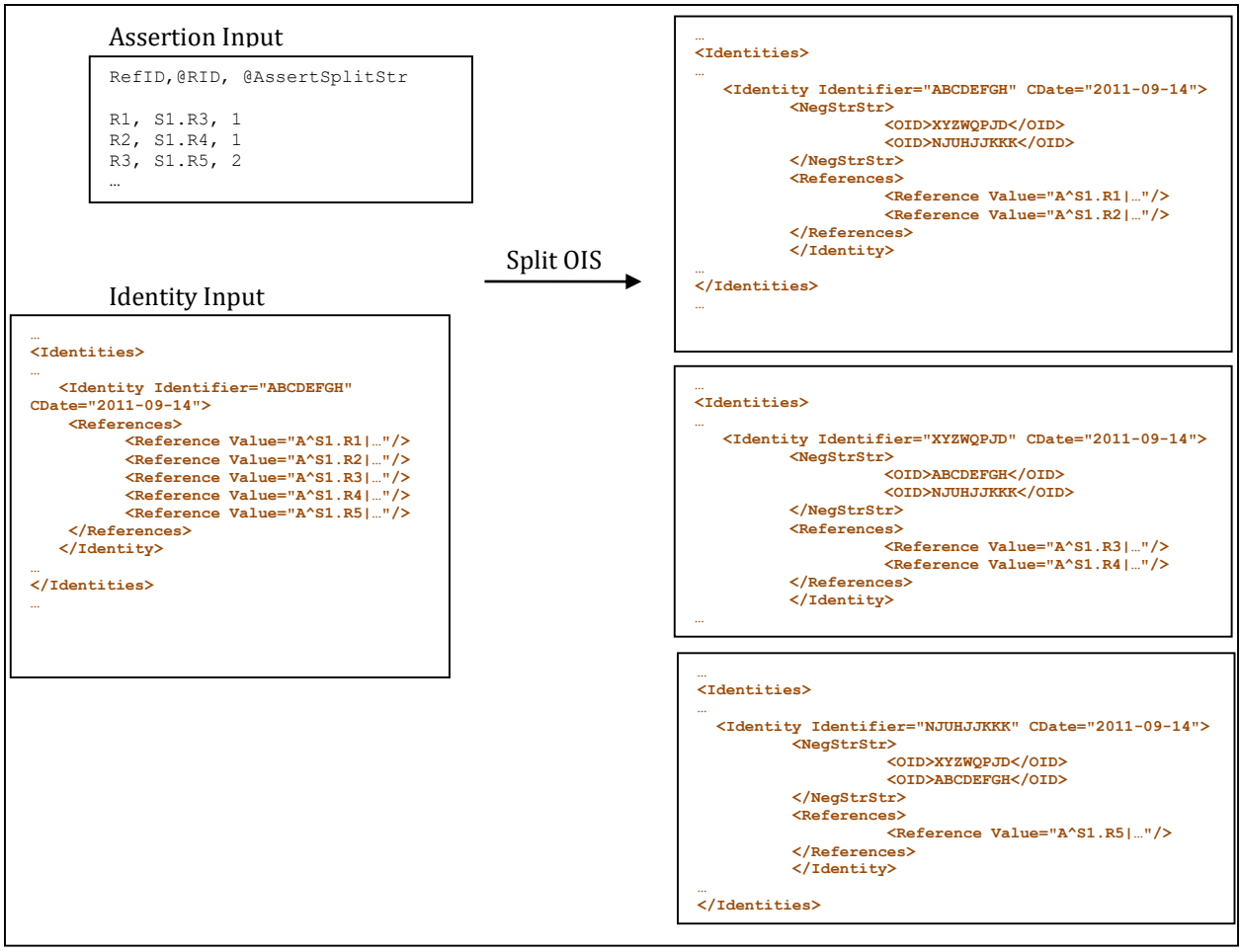
- Example:



- `@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|Samual|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="SUGPQC4UBKPBAU9O" 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.

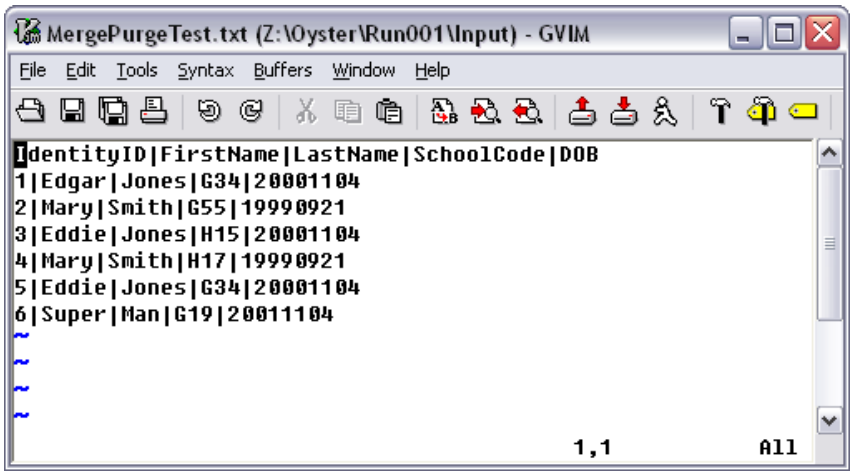


Figure 1: Delimited Text Input

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.

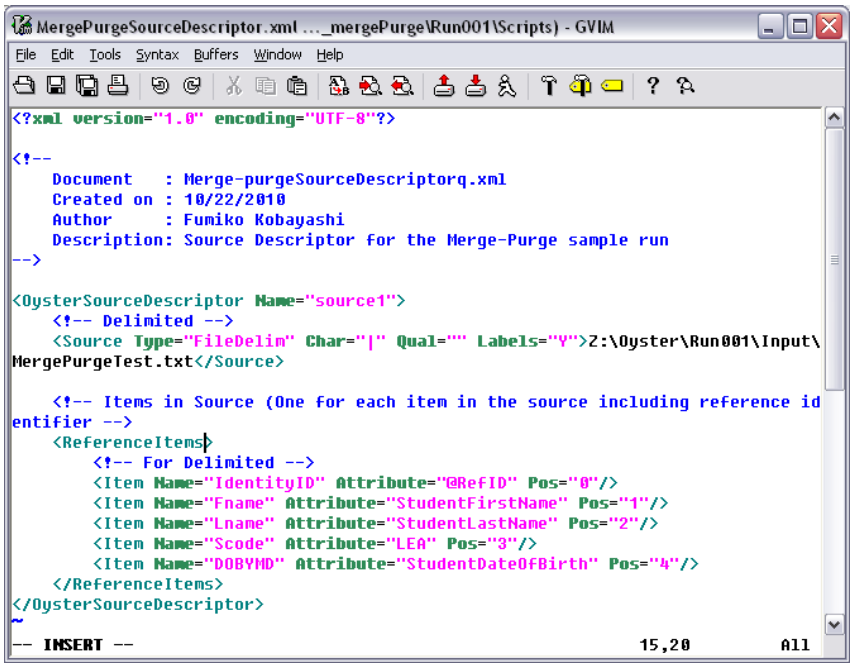


Figure 2: Source Descriptor for a Delimited Text File

1.2 From Fixed Width Files

Figure 3 illustrates a fixed width text file that can be used as input for an OYSTER run.

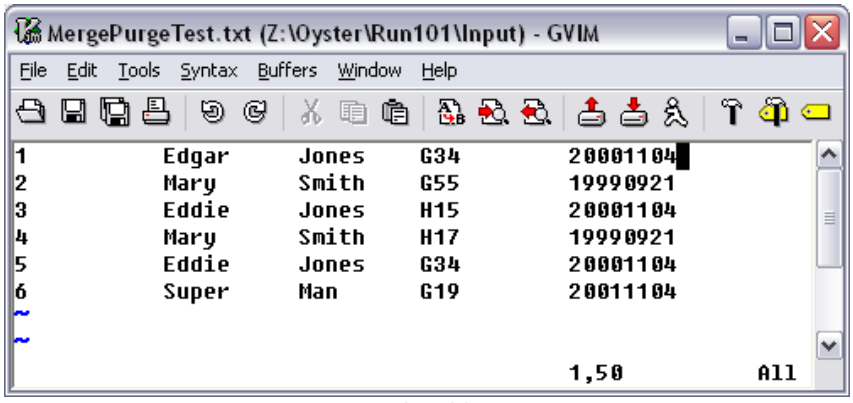


Figure 3: Fixed Width Text Input

The SourceDescriptor illustrated in Figure 4 can be used in an OYSTER run if the input source is stored in a fixed width text file like the one shown in Figure 3.

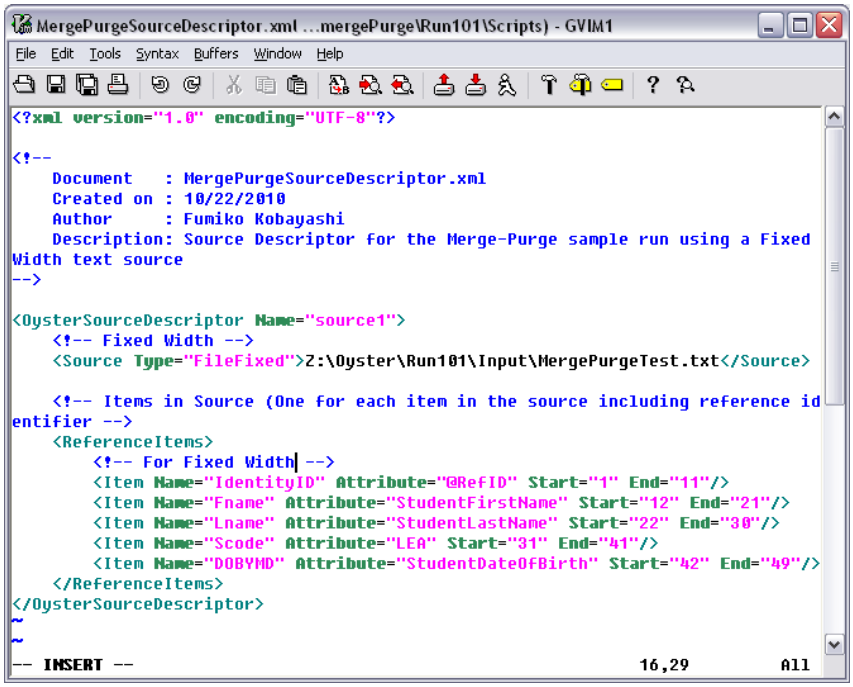


Figure 4: Source Descriptor for a Fixed Width Text File

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.

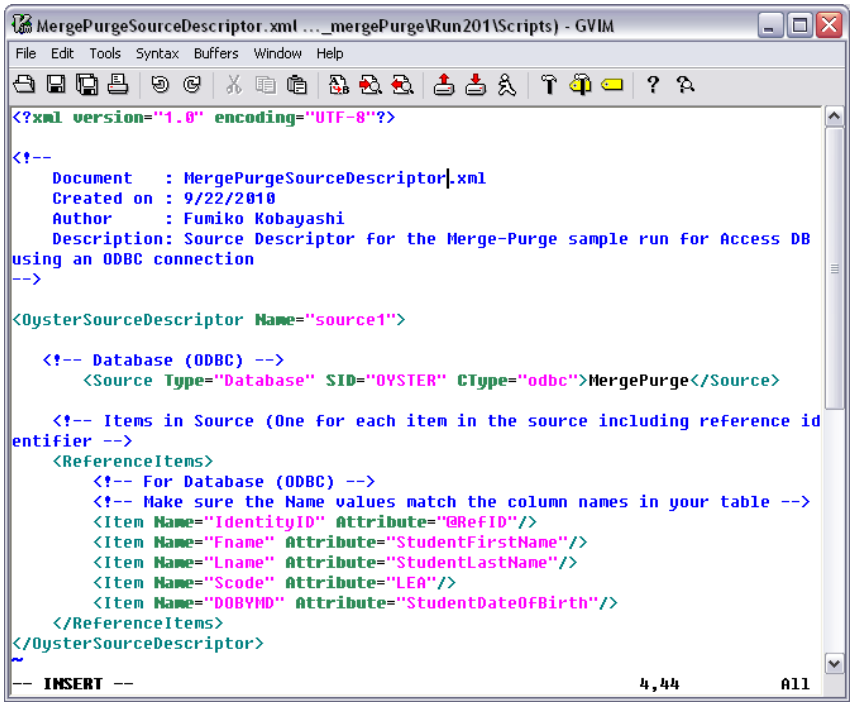


Figure 5: Source Descriptor for accessing an Access DB Table

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.

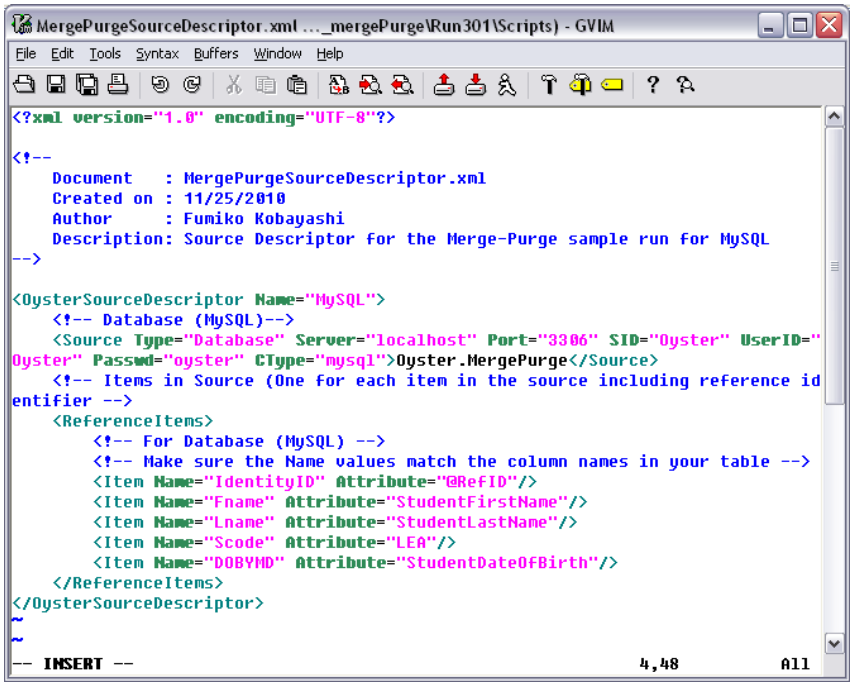


Figure 6: Source Descriptor for accessing a MySQL DB Table

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.

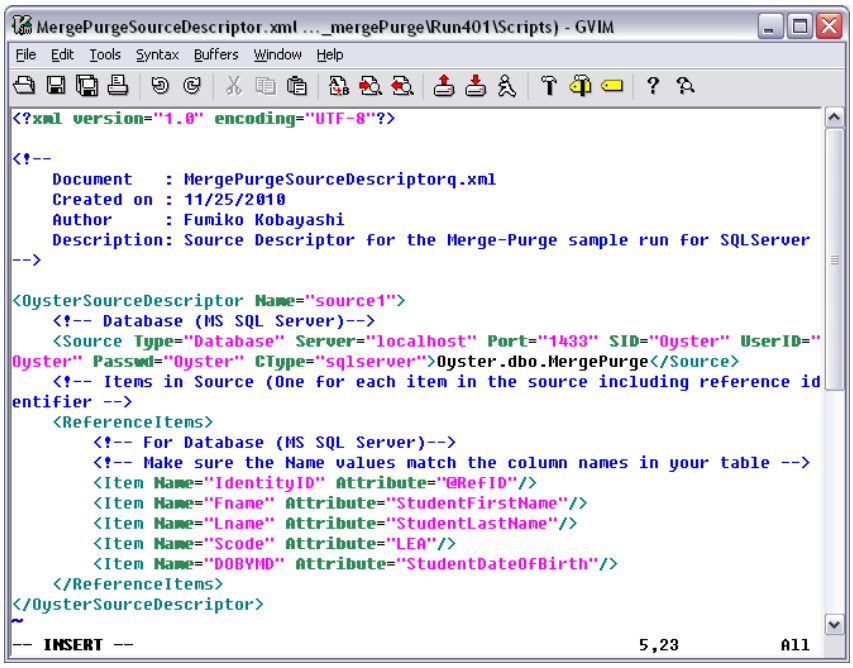


Figure 7: Source Descriptor for accessing SQL Server Table

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 Modification 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.

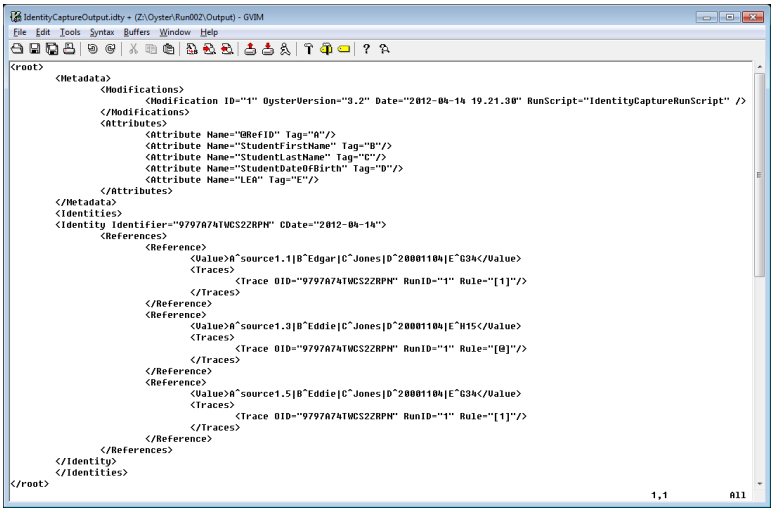


Figure 8: Example Identity Document

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.

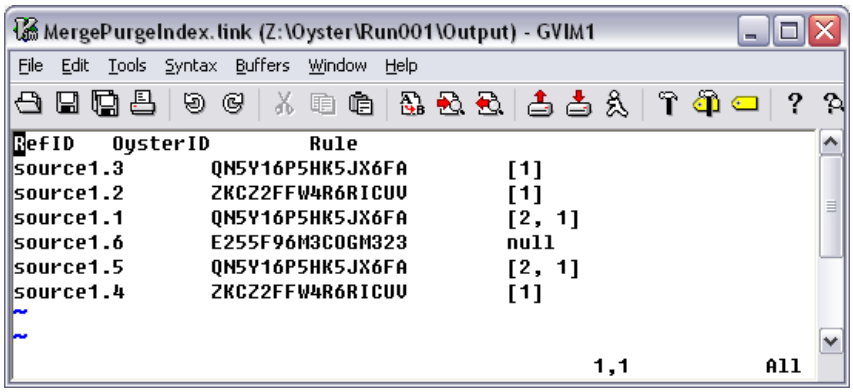


Figure 9: Example Link index

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.

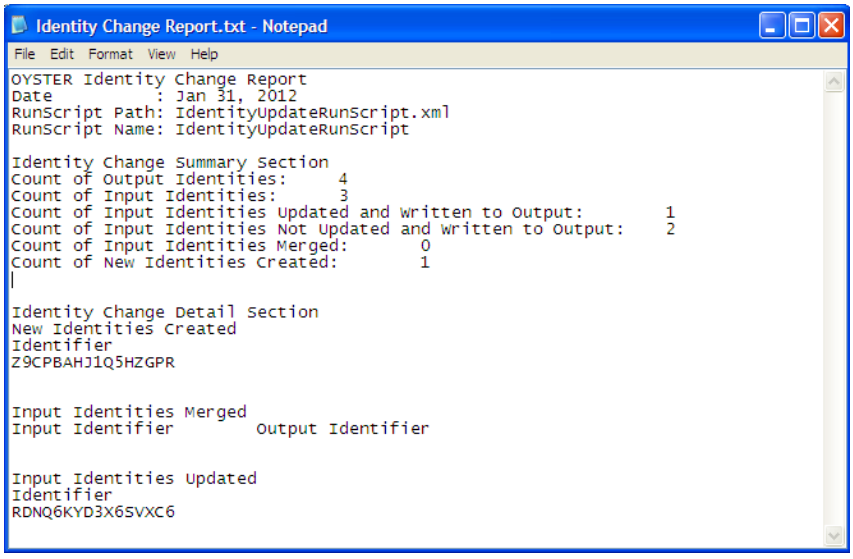


Figure 10: Identity Change Report with ChangeReportDetail="Off"

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.

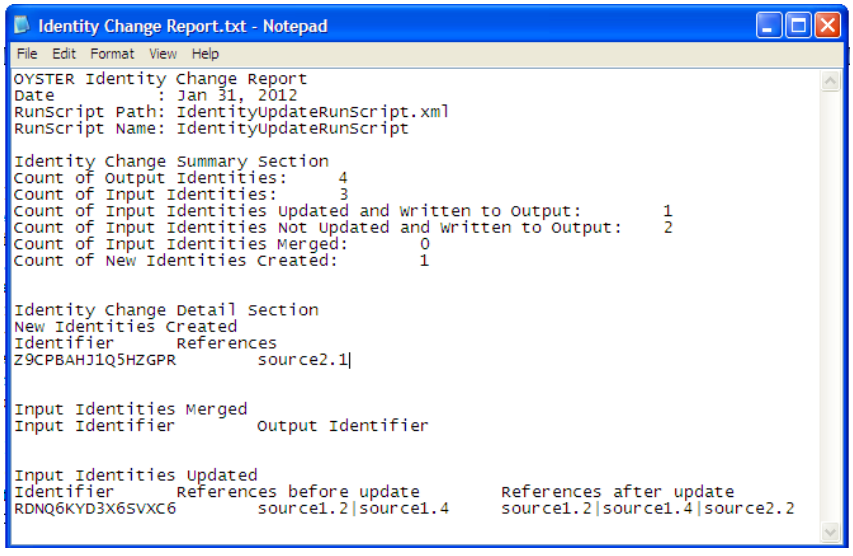


Figure 11: Identity Change Report with ChangeReportDetail="On"

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.

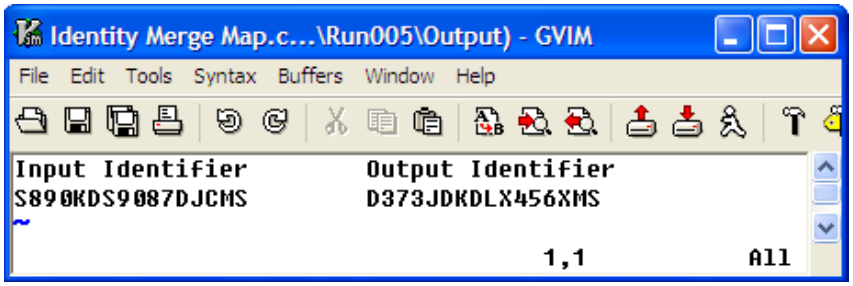


Figure 12: Identity Merge Map

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 Sizes 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.
  $\sum$ (Cluster Size – 1) * # 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 - (Total clusters / 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 it 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 indicies.
- `Unique tokens` - The unique number of RefID's held in the indicies.
  - 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 explaine 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 ?3 (excess ?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.

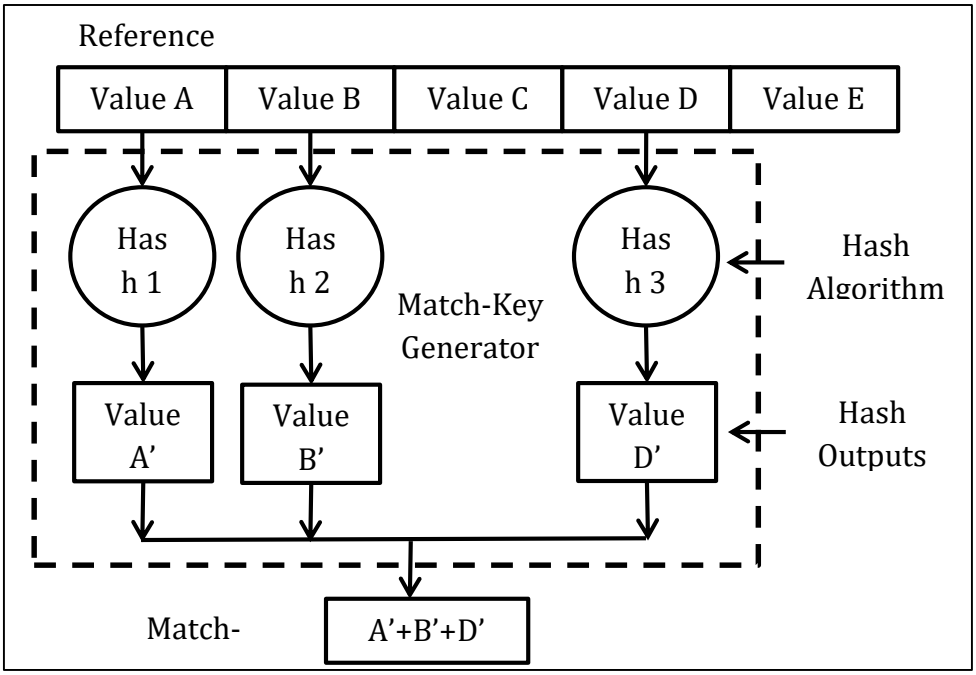


Figure 21: Schematic of Match-Key Generator

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.

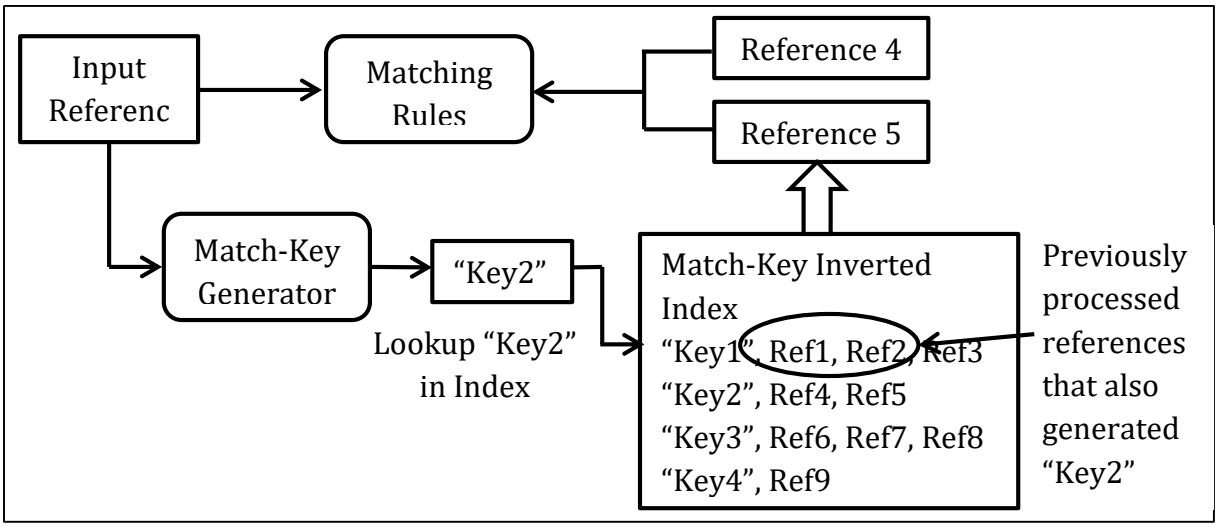


Figure 22: Match-Key Lookup for Match Candidates

The same process show in Figure 22 is also shown in Figure 23using 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.

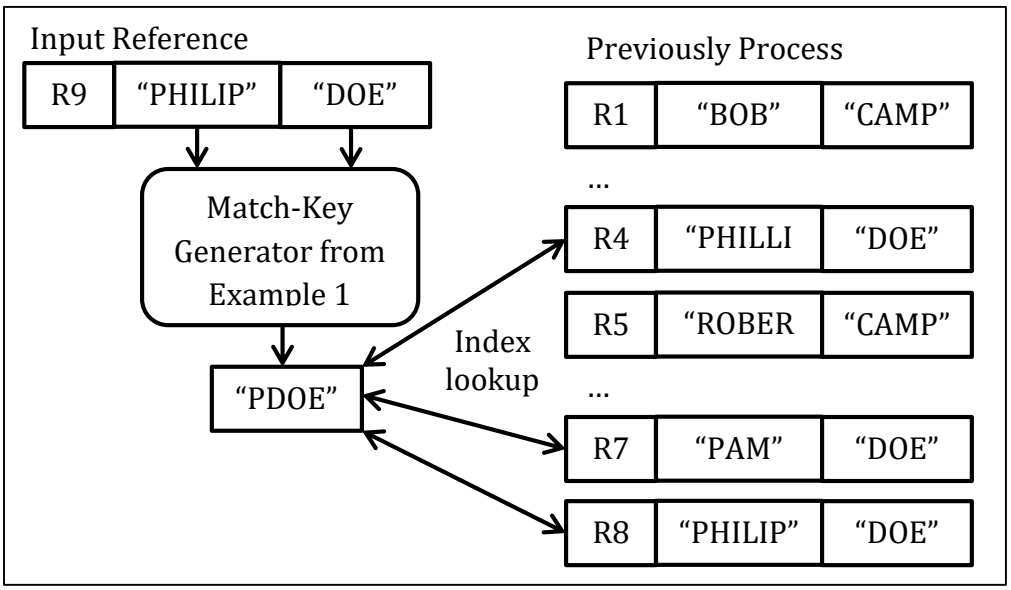


Figure 23: Candidates Returned by Match-Key "PDOE"

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.

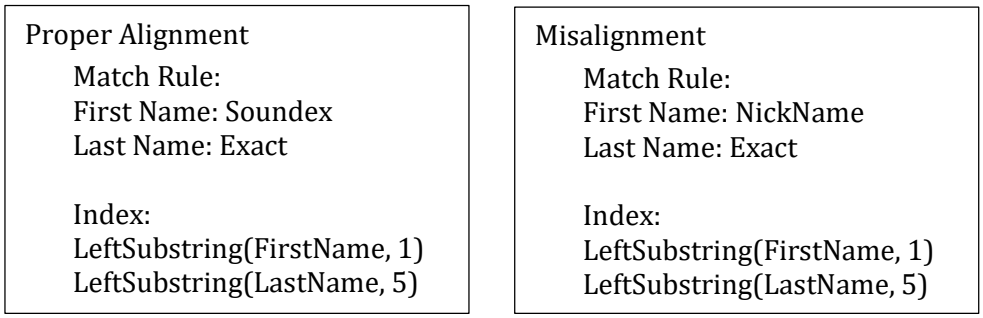


Figure 24: Examples of Good and Poor Rule-to-Index Alignment

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 generated 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

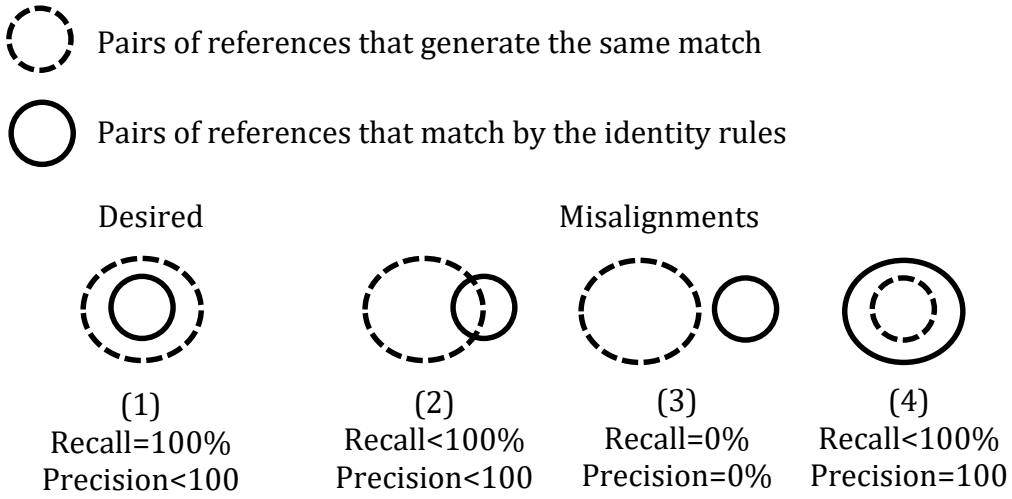


Figure 25: Diagrams of Rule-to-Alignment Scenarios

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.

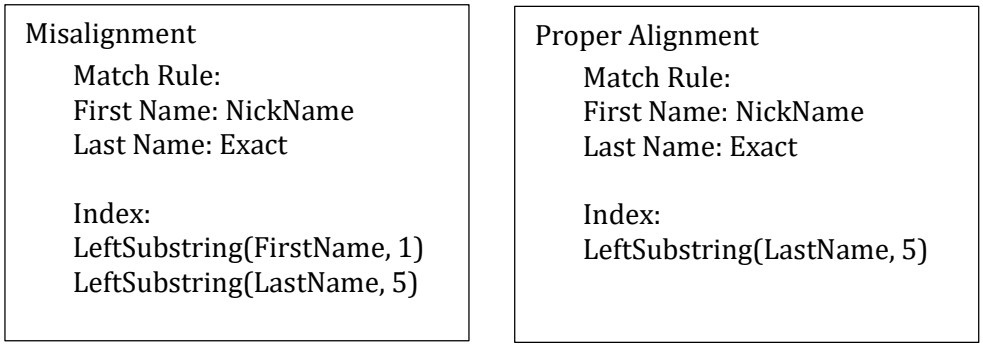


Figure 26: Indexing on Fewer Attributes to Attain Alignment

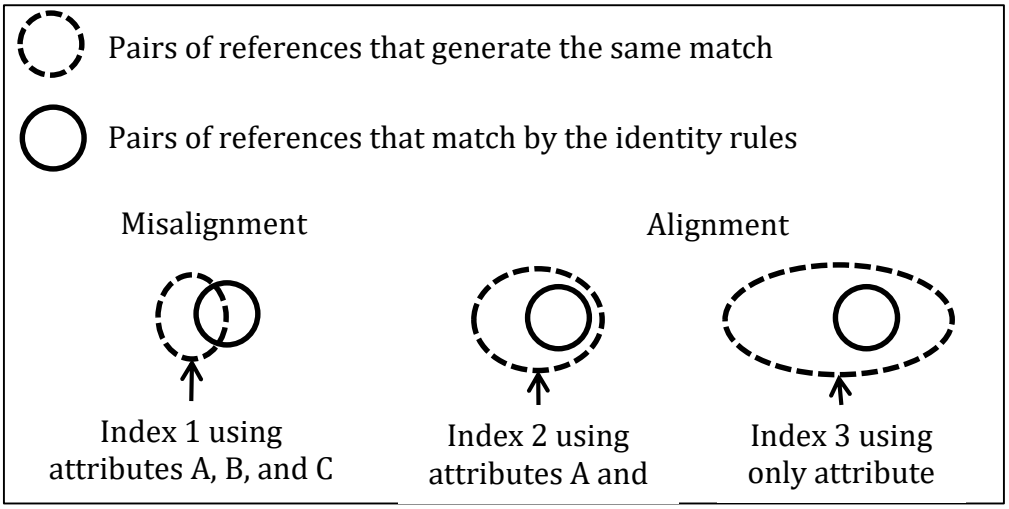


Figure 27: The Effect of Removing Attributes from the Match-Key Generator

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.

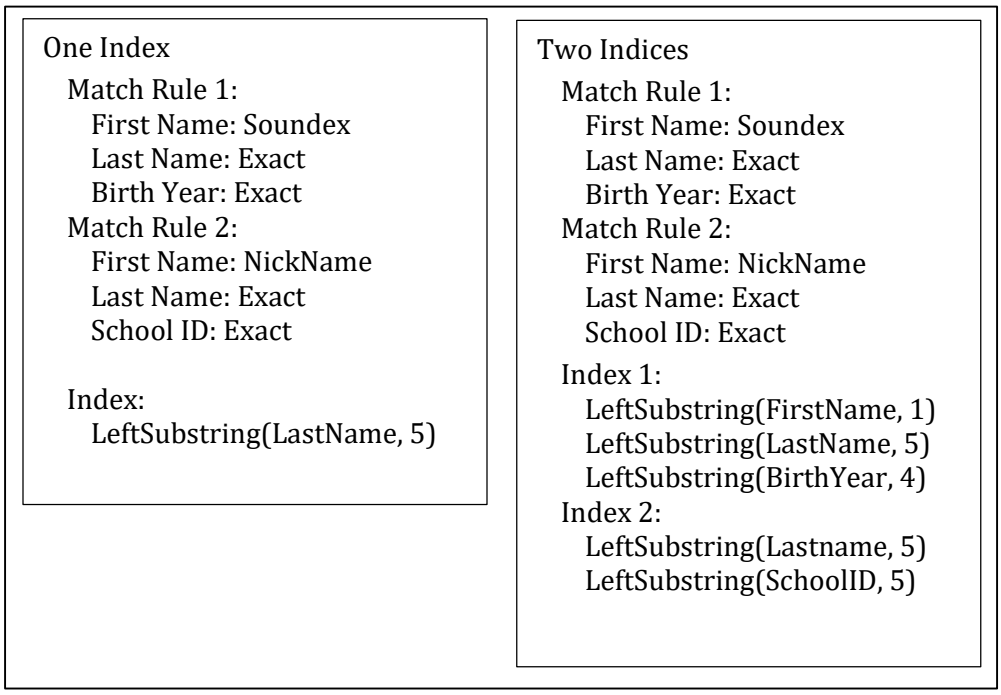


Figure 28: Same Rule Set with One Index and Two Indices

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.

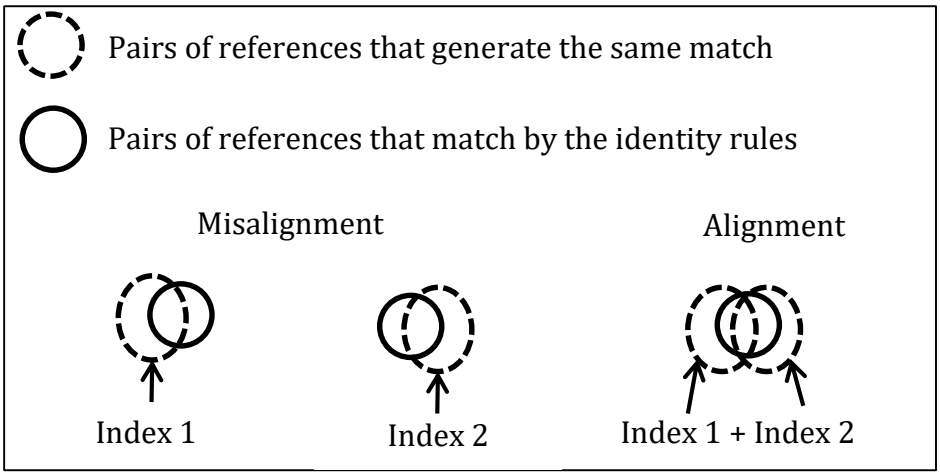


Figure 29: Combined Effect of Two Indices

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 with 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 and 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 compared to each previously processed record (assuming the One-Pass algorithm) by each of the 17 rules, or 17xNx(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 17xNxM 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 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 leave 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 rules 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

Appendix A: List of Oyster Functions by Usage

Function Name	Similarity	Index	DataPrep	Parms
EXACT	Syntatic			
EXACT_IGNORE_CASE	Syntatic			
EXACTORNICKNAME	Combo			
EXACTORBOTHMISSING	Combo			
EXACTOREITHERMISSING	Combo			
INITIAL	Syntatic			
JACCARD	Syntatic			Yes
LED	Syntatic			Yes
ListOverlap	Syntatic			Yes
ListOverlapPV	Syntatic			Yes
MatrixComparator	Syntatic			Yes
MISSING	Syntatic			
PSUBSTR	Syntatic			Yes
QTR	Syntatic			Yes
SCAN	Syntatic	Single Value	Yes	Yes
SMITHWATERMAN	Syntatic			Yes
SORENSEN	Syntatic			Yes
SUBSTRLEFT	Syntatic			Yes
SUBSTRMID	Syntatic			Yes
SUBSTRRIGHT	Syntatic			Yes
TANIMOTO	Syntatic			Yes
TRANSPOSE	Syntatic			
TVERSKY	Syntatic			Yes
NICKNAME	Semantic			
CAVERPHONE	Phonetic			
CAVERPHONE2	Phonetic			
DMSOUNDEX	Phonetic	Single Value	Yes	
IBMALPHACODE	Phonetic	Single Value	Yes	
MATCHRATING	Phonetic			
METAPHONE	Phonetic			
METAPHONE2	Phonetic			
NYSIIS	Phonetic	Single Value	Yes	
SOUNDEX	Phonetic	Single Value	Yes	
ListTokenizer		Multi-Value		Yes
ListTokenizerPV		Multi-Value		Yes
MatrixTokenizer		Multi-Value		Yes
PlaceholderValueFilter		Multi-Value	Yes	Yes

Appendix B: Description of Oyster Functions

EXACT –As a similarity function, EXACT returns True if and only if both strings are non-empty are have exactly same characters in corresponding order. This function is case sensitive. If either or both strings are empty or blank, the comparator returns False. Example: "SAM" compared to "SAM" returns True, "SAM compared to "Sam" returns false.

EXACT_IGNORE_CASE – As a similarity function, EXACT_IGNORE_CASE returns True if and only if both strings are non-empty and share the same characters in corresponding order except that corresponding letters may different by case, i.e. This function is not case sensitive. Example: "Sam" compared to "SAM" returns True. If either or both strings are empty or blank, the comparator returns False.

INITIAL – As a similarity function, INITIAL returns True only under the following conditions: (1) Both string are non-empty, (2) one string has exactly one non-blank character, (3) the other string has two or more non-blank characters, and (4) the first character of both strings are the same. This function is case sensitive. If either or both strings are empty or blank, the comparator returns False. Example: "S" compared to "SAM" returns True; "SAM" compared to "SALLY" returns False.

JACCARD(Threshold) – The JACCARD function takes a single parameter "Threshold", a decimal value between 0.0 and 1.0.
As a similarity function, JACCARD returns True under the following conditions (1) both strings are non-empty, and (2) considering each string as a set of characters, the ratio of the intersection of the character sets to the union of the character sets is greater than or equal to the "Threshold" parameter. This function is case sensitive. If either or both strings are empty or blank, the comparator returns False. Example: "Johnson" compared to "Holston" returns True for JACCARD(0.50) because the intersection is 4 characters {O, H, N, S} and the union is 7 characters {J, O, H, N, S, L, T} and 4/7=0.5714 is greater than 0.50.

LED – The Normalized 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.

TRANSPOSE – As a similarity function TRANSPOSE returns True if and only if the two string are non-empty and differ only by exactly one reversal of 2 adjacent characters. TRANSPOSE is case sensitive. If either or both strings are empty or blank, the comparator returns False. Example: "ABCD" compared to "ACBD" returns True; "ABCD" compared to "ABCD" returns False.

NICKNAME – As a similarity function, NICKNAME returns True if and only if both strings are non-empty and both string occur together on the same row of the ALIAS table in the DATA Directory. This function is not case sensitive. OYSTER comes with a standard table of common North American nicknames. Each row of the alias table

comprises two non-empty string separated by the TAB character. Users can add or delete nickname pairs from the ALIAS table to address particular applications. If either or both strings are empty or blank, the comparator returns False. Example (using the standard table): "Robert" compared to "BOB" returns true.

- **SOUNDEX** – As a DataPrep function, SOUNDEX returns the SOUNDEX code of the attribute value as generated by the standard SOUNDEX algorithm. This function is not case sensitive. Example: the string "PATTERSON" generates the code "P362"
  As an Index function, SOUNDEX inserts the SOUNDEX code of the attribute value as an index key for the input record into the OYSTER Inverted Index. This function is not case sensitive. If either or both strings are empty or blank, the comparator returns False. Example: the string "PATTERSON" indexes the input record by the code "P362"
  As a Similarity Function, SOUNDEX returns true if the strings are non-empty and generate the same SOUNDEX code. This function is not case sensitive. Example: "PATTERSON" compared to "Peterson" returns True because both strings generate the SOUNDEX code P362.
- **DMSOUNDEX** – Operates exactly the same as the SOUNDEX function except the codes are generated by the Daitch-Mokotoff Algorithm.
- **IBMALPHACODE** – Operates exactly the same as the SOUNDEX function except the codes are generated by the IBM Alpha Code Algorithm.
- **NYSIIS** – Operates exactly the same as the SOUNDEX function except the codes are generated by the New York State Identification and Intelligence System Algorithm.
- **MATCHRATING** – Operates exactly the same as the SOUNDEX function except the codes are generated by 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** – Operates exactly the same as the SOUNDEX function except the codes are generated by an algorithm optimized for accents used in the southern part of New Zealand.
- **CAVERPHONE2** – Operates exactly the same as the SOUNDEX function except the codes are generated by an algorithm optimized for accents used in the southern part of New Zealand.
- **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.
- **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 of 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. Note: If the length is given as 0 (zero), all characters of the specified character type will be extracted. 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 0 and 30
                - If Length=0, all characters of the specified type will be extracted
        - 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 charecters 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(LR, ALPHA, 0, ToUpper, SameOrder)	"123 N. Oak St, Apt #5"	"123NOAKSTAPT5"
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"

- **List Overlap Comparator (LOC)**

`Overview`

The purpose of the List Overlap Comparator ("ListOverlap") is to allow users to determine if two strings representing item lists rise to a given level of similarity.

ListOverlap is a similarity function and requires two inputs. Both inputs are assumed to be character strings comprising a list of items where the items are separated by a single character. ListOverlap also takes two control parameters. The first control parameter is the minimum percentage of overlap between the two lists required to signal a "True" match condition. The second control parameter is the list delimiter character. After both input lists have been separated into items, the ListOverlap comparator calculates the similarity between the strings as the percentage of non-empty list items in common between the lists versus the total number of non-empty items in the longer list. If the percentage of overlap is equal or larger than a predefined threshold given as the first input parameter, then the ListOverlap comparator signals a "True" match condition, otherwise the ListOverlap compartor signals a "False" match condition.

Semantics

The comparison of two references is determined as follows:

1. Each input string is separated into a list of items based on the list delimiter character provided as the second control parameter to ListOverlap. An item is deemed "empty" is it contains no characters or all blank characters. Empty items are dropped from the list. If after dropping empty list items one or both lists are empty, then the ListOverlap comparator signals a "False"match.

2. After empty list items are dropped from the list, all duplicate items within the same list are dropped.

3. Given that both lists have at least one item, the ListOverlap comparator determines the degree of overlap between the two lists. The Degree of Overalp between the two lists is the total number of items shared between the two lists is divided by the number of items in the longer list.

4. If the list Degree of Overlap is greater than or equal to the predefined threshold given as the first control parameter, then the ListOverlap comparator signals a "True" match condition, otherwise the ListOverlap comparator signals a "False" match condition

Syntax

The syntax for ListOverlap is "ListOverlap(C1, C2)" where the two parameters C1 and C2 are as follows:

C1 is the similarity threshold given as a decimal value between 0.00 and 1.00.

C2 is a single character enclosed in apostrophes, e.g. ‘,’

For example, if the threshold is 70% and the list delimiter is a comma, then the List Overlap Comparator would be encoded in a match rule as

Similarity = “ListOverlap(0.70, ‘,’)”

If only one parameter given, it is assumed to be the threshold value, and the delimiter defaults to a comma. For example,

Similarity = “ListOverlap(0.90)”

Would indicate a similarity threshold of 90% and a list comparator defaults to a comma character. Another option is to not include parameters. For example,

Similarity = “ListOverlap()”

Defaults to a threshold of 80% and the list delimiter to a comma character.

LOC Requirements

LOC.1 Syntax for the List Overlap Comparator

LOC.1.1 The name of the comparator shall be “ListOverlap”

LOC.1.2 The name shall not be case sensitive

LOC.2 Control parameters for ListOverlap

LOC.2.1 The comparator shall have 2 control parameters C1 and C2 represented as “ListOverlap(C1, C2)”

LOC.2.2 The first control parameter C1 is the match threshold value, and it shall be represented as a numeric decimal value between 0.00 and 1.00

LOC.2.3 If the first control parameter is not a numeric decimal value between 0.00 and 1.00, then the system shall default to a value of 0.80.

LOC.2.4 The second control parameter C2 is the list delimiter character, and it shall be a single character enclosed in apostrophes.

LOC.2.5 If the second control parameter is not a character, then the system will default to a comma character for a list delimiter.

LOC.2.6 The list delimiter character shall be any character except it shall not be an ampersand (&), less-than (<), greater-than (>), quotes (“), apostrophe (‘), or pipe (|) character.

LOC.2.7 If a valid character is not given the system shall default to comma character as the list delimiter.

LOC.3 Inputs for ListOverap

LOC.3.1 The two inputs for ListOverlap shall both be character strings

LOC.3.2 Each string shall be interpreted as a list of items where the items are separated by a list delimiter character given as the second control parameter.

LOC.3.3 Each list item shall be trimmed of leading and trailing blanks

LOC.3.4 If the result of trimming blanks is an empty string, then the item shall be dropped from the list

LOC.3.5 If the result of dropping empty items results in an empty list, then the ListOverlap comparator shall signal a "False" match condition. No further processing is required

LOC.3.6 After dropping items from each list, each list shall be checked for duplicate items within the same list. For purposes of comparison, all letter characters shall be converted to upper case. If an item in the list is a duplicate of another item in the same list, then the duplicate item shall be dropped.

LOC.4 Determination of True/False Match

LOC.4.1 In the case that either or both of the final lists are empty, then the comparator shall signal a "False" match and no further processing necessary.

LOC.4.2 Otherwise, all of the items in the first list shall be compared to all of the items in the second list to determine the number of duplicate items between the two lists. For purpose of comparison, all letters characters shall be converted to upper case.

PVC.4.3 After all comparisons are made, the degree of overlap shall be calculated as Degree of Overlap = (Number of Duplicates)/(Length of the Longer List)

PVC 4.4 If the Degree of Overlap is greater than or equal to the match threshold given as the first control parameter, then ListOverlap shall signal a "True" match, else ListOverlap shall signal a "False" match.

LOC Example:
Consider a matching term given as
<Term Item="List" Similarity="ListOverlap(0.60, ',')"
The first value of "List" is
"Mary, Robert, Bobby, Suzan, Mary,,James"
The second value of "PROPERY_VALUE_STRING" is
"Mary, ,Bobby, Sam, George, James, Robert"

The empty item (6th item) is removed from the first list. The 5th item "Mary" is a duplicate item also removed from the list. The final list of 5 items is:

1. MARY
2. ROBERT
3. BOBBY
4. SUSAN
5. JAMES

The 2nd empty item is removed from the second list. The final list of 6 items is:

1. Mary
2. Bobby
3. Sam
4. George
5. James
6. Robert

In the comparison between the two lists

	MARY	ROBERT	BOBBY	SUSAN	JAMES	ROBERT
Mary	Overlap					
Bobby			Overlap			
Sam						
George						
James					Overlap	
Robert		Overlap				

Total items in the overlap = 4
Number of items in longer list = 6
Similarity ratio = 4/6 =0.66
Match Result = "True" because 0.66 > 0.60

- **Property-Value List Comparator (PVC)**

Overview

The purpose of the Property-Value List Comparator ("ListOverlapPV") is to allow users to determine if two strings representing property-value pair lists rise to a given level of similarity. ListOverlapPV is a similarity function and requires two inputs. Both inputs are assumed to be character strings comprising a list of items where each item is pair of values. The first value of the pair is the Property Name and the second value is the Property Value. ListOverlapPV assumes that the property values pairs in the list are consistently separate by single character (the list delimiter), and within the pair, the property name is consistently separated from its property value by a different character (the pair delimiter).

ListOvelapPV also takes three control parameters. The first control parameter is the minimum percentage of overlap between the two lists required to signal a "True" match condition. The second control parameter is a two-character string specifying the list delimiter character followed by the pair delimiter character. The optional third control parameter is string containing a list of placeholder property values.

After both input lists have been separated into pairs, the ListOverlapPV first removes any pairs where the property value is empty or where the property value is in the list of placeholder values. After these pairs are removed, any duplicate property-value pairs within the same input are list are removed. Finally, the comparator calculates the degree of overlap between the two lists as the percentage of pairs in common between the lists versus the total number of pairs in the longer list. If the degree of overlap is equal to, or larger than, a predefined threshold given as the first input parameter, then the ListOverlapPV comparator signals a "True" match condition, otherwise the ListOverlapPV compartor signals a "False" match condition.

Semantics

The similarity between two property-value pair lists is determined as follows:

1. Each input string is separated into a list of property-value pairs based on the list delimiter character given in the second control parameter.
2. A property-value pair is deemed "empty" is it contains no characters or all blank characters. Empty property-value pairs are dropped from the list.
3. Next each property value pair is separated into its property name and corresponding property value based on the pair delimiter character given in the second control parameter. If either the property name or property value is empty, the property-value pair is dropped from the list.
4. Next each property value is compared to the list of placeholder property values given in third control parameter. If the property value of a property-value pair matches one of the placeholder values, the property-value pair is dropped from the list.
5. 5. Next, all duplicate property-value pairs within the same input string are dropped.
6. If the final list of property-value pairs extracted from either input string is empty, then the comparator signals a "False" match condition.

7. Otherwise the pairs in both lists are compared for matching items. The number of matching items between the pair lists is divided by the number of items in the longer of the two lists. If this ratio is greater than or equal to the first control parameter, then the ListOverlapPV comparator signals a "True" match condition, otherwise the ListOverlapPV comparator signals a "False" match condition

Syntax

The syntax for ListOverlapPV is "ListOverlapPV(C1, C2, C3, C4)" where

C1: The similarity threshold given as a decimal value between 0.00 and 1.00.

C2: The list delimiter given as a single character enclosed in apostrophes.

C3: The pair delimiter given as a single character enclosed in apostrophes.

C4: The list of placeholder values given as a string enclosed in apostrophes. The placeholder values in the list must be separated from each other by the pipe (|) character.

For example, if the threshold is 80%, the list delimiter is a comma, the pair delimiter is a colon (:), and there are two placeholder values "UNK" and "?", then the OYSTER call to the Property-Value List Comparator would be encoded in a match rule as

Similarity = "ListOverlapPV(0.80, ',', ':', 'UNK, ?')"

PVC Requirements

PVC.1 Syntax for the Property-Value List Comparator

- PVC.1.1 The name of the comparator shall be "ListOverlapPV"
- PVC.1.2 The name shall not be case sensitive

PVC.2 Control parameters for ListOverlapPV

- PVC.2.1 The comparator shall have 4 control parameters (arity of 4)
- PVC.2.2 The first control parameter shall be the degree of overlap threshold value.
  - PVC.2.2.1 The degree of overlap threshold value shall be given as a numeric decimal value between 0.00 and 1.00.
  - PVC.2.2.2 If the degree of overlap threshold value is not given or in not in the proper format the system shall default the value to 0.80.
- PVC.2.3 The second control parameter shall be the list delimiter character
  - PVC.2.3.1 The list delimiter character shall be given as a single character enclosed by apostrophe characters
  - PVC.2.3.2 The list delimiter character shall be any character except it shall not be anampersand (&), less-than (<), greater-than (>), quotes ("), apostrophe ('), or pipe (|) character.

PVC.2.3.3 If the list delimiter character is not given or not in the proper format, the system shall default its value to the comma (,) character.

PVC.2.4 The third control parameter shall be the pair delimiter character.

PVC.2.4.1 The pair delimiter character shall be given as a single character enclosed by apostrophe characters.

PVC.2.4.2 The pair delimiter character shall be any character except it shall not be an ampersand (&), less-than (<), greater-than (>), quotes (“), apostrophe (‘), or pipe (|) character, and it shall not be the same as the list delimiter character.

PVC.2.4.3 If the pair delimiter character is not given or not in the proper format, the system shall default its value to the colon (:) character.

PVC.2.5 The fourth control parameter shall be the list of property placeholder values.

PVC.2.5.1 The list of placeholder values shall be given as a string of characters enclosed by apostrophe characters.

PVC.2.5.2 If there is more than one placeholder value in list, consecutive values shall be separated from each other by the pipe (|) character.

PVC.2.5.3 Placeholder values shall be comprised of any characters except they shall contain an ampersand (&), less-than (<), greater-than (>), quotes (“), apostrophe (‘), or pipe (|) character.

PVC.3 Creating the lists of property value pairs

PVC.3.1 The two inputs for ListOverlapPV shall both be character strings

PVC.3.2 Each string shall be interpreted as a list of property-value pairs where the pairs are separated from each other by the list delimiter character given in the second control parameter.

PVC.3.3 Each property-value pair shall be trimmed of leading and trailing blanks

PVC.3.4 If the result of trimming blanks is an empty string, then the property-value pair shall be dropped from the list

PVC.3.5 If the result of dropping empty property-value pairs results in an empty list, then the ListOverlapPV comparator shall signal a “False” match condition. No further processing is required

PVC.3.6 Otherwise, if there are items remaining in both lists of property-value pairs, each pair in each list shall be separated into a property name and corresponding property value according to the pair delimiter character given as the third control parameter.

PVC.3.7 Each property name and each property value shall be trimmed of leading and trailing blacks.

PVC.3.8 If either the property name or the property value of a property-value pair is empty, then the property-value pair shall be dropped from the list.

PVC.3.9 Each property value shall be compared to the list of placeholder property values given in the third control parameter. For purposes of comparison, all letter characters shall be converted to upper case.

PVC.3.10 If the property value of a property-value pair matches one of the placeholder values, the property-value pair shall be dropped from the list.

PVC.3.11 After dropping empty pairs and pairs with placeholder values, each list shall be checked for duplicate property value pairs within the same list. For purposes of comparison, all letter characters shall be converted to upper case. If a pair in the list is a duplicate of another pair in the same list, then the duplicate pair shall be dropped.

PVC.4 Determination of True/False Match

PVC.4.1 In the case that either or both of the final lists of property-value pairs are empty, then the comparator shall signal a “False” match and not further processing is necessary.

PVC.4.2 Otherwise, all of the pairs in the first list shall be compared to all of the pairs in the second list to determine the number of duplicate pairs between the two lists. For purpose of comparison, all letters characters shall be converted to upper case.

PVC.4.3 After all comparisons are made, the degree of overlap shall be calculated as

Degree of Overlap = (Number of Pairs in Common)/(Length of the Longer List)

If the Degree of Overlap is greater than or equal to the match threshold given as the first control parameter, then ListOverlapPV shall signal a “True” match, else ListOverlapPV shall signal a “False” match.

PVC Example:

Consider a matching term given as

<Term Item=”PV_String” Similarity=”ListOverlapPV(0.80, ‘,’, ‘:’, ‘UNKNOWN|UNK’)”

And

The first list of pairs for “PV_String” has four list items

“TYPE:COLOR, COLOR:UNK, APP:LASER 42, YIELD:6000, PACKAGE:BOX”

The second list of pairs for “PV_String” has six list items

“TYPE:Color,,APPL:Laser 36,YIELD:8000,PACKAGE:Box,Mount:”

The second pair COLOR:UNK in the first list is removed because “UNK” is one of the placeholder values given by the fourth control parameter.

1. TYPE:COLOR
2. YIELD:6000
3. PACKAGE:BOX

The second item of the second list is removed because it is empty. The sixth item of the second list is removed because the property value is missing.

1. TYPE:COLOR
2. APP:LASER 36
3. YIELD:8000
4. PACKAGE:BOX

In the between the two lists

	TYPE:COLOR	YIELD:6000	PACKAGE:BOX
TYPE:COLOR	Overlap		
APP:LASER 36			
YIELD:6000			
PACKAGE:BOX			Overlap

Overlapping pairs = 2
Items in longer list = 4
Degree of Overlap = 2/4 =0.50
Match Result = “False” because 0.50 < 0.80

- **ListTokenizer Function (LTK)**

Overview

The primary purpose of the ListTokenizer Function (“ListTokenizer”) is to support indexing (blocking) for multivalued ListOverlap comparator. ListTokenizer is a parsing function to be used in an index generator (Hash attribute). The function takes as input a character string representing a list of items, and parses it into individual items. The substrings of the input string selected as list items will depend upon the character provided by the user to be the list delimiter character. ListTokenizer has three control parameters. The first control parameter specifies the delimiting character. The second

control parameter specifies the minimum length of an item in the list to be indexed, and the third is an optional list of items be excluded from indexing (the exclusion list).

Semantics

The operation of ListTokenizer is as follows:

1. The input string from the reference is separated into a list of items based on the delimiter character provided as the first control parameter.
2. After tokenization, any list item that meets one of the following conditions is dropped from the list of items to be indexed
    a. The length of the item is less than the minimum item length specified by the second control parameter
    b. The item matches one of the excluded items in the exclusion list specified by the third control parameter
3. The reference is then indexed for each remaining item in concatenation with all other hash values produced within the same index generator.

Syntax

The syntax for ListTokenizer is "ListTokenizer(C1, C2, C3)" where

C1 is the list delimiter character enclosed by apostrophes
C2 is a positive integer value indicating the minimum length of tokens to index
C3 is an optional string enclosed in apostrophes representing a list of excluded items separated by the pipe (|) character.

For example, if the list delimiters is a comma character (,), the minimum token length is 3, and the excluded list items are "ABC" and "EFG", then the ListTokenizer Function would be encoded in the element of an <Index> element as

<Segment Item="VarOne", Hash="ListTokenizer(',', 3, 'ABC|EFG')"

LTK Requirements

LTK.1 Syntax for the ListTokenizer Function
- LTK.1.1 The name of the function shall be "ListTokenizer"
- LTK.1.2 The name shall not be case sensitive

LTK.2 Control parameters for ListTokenizer
- LTK.2.1 The comparator shall have 3 control parameters (arity of 3) C1, C2, and C3 represented as "ListTokenizer(C1, C2, C3)"
- LTK.2.2 The first control parameter C1 shall specify the tokenizing characters.
    - LTK.2.2.1 The list delimiting character shall be enclosed in apostrophes.
    - LTK.2.2.2 The list delimiting character shall be any characters except it shall not be an ampersand (&), less-than (<), greater-than (>), quotes ("), apostrophe ('), or pipe (|) character.

LTK.2.2.3 If no list delimiter character is given or is not given in the proper format, the system shall default its value to the comma (,) character.

LTK.2.3 The second control parameter C1 shall specify the minimum length of a list item to be indexed.

LTK.2.3.1 The minimum length shall be given as an integer value.

LTK.2.3.2 If the minimum length given is 0, then all tokens will be indexed.

LTK.2.3.3 If the minimum length is not given or is not given in the proper format, the system shall default the minimum length to a value of 2.

LTK.2.4 The third control parameter shall be the list of excluded list items.

LTK.2.4.1 The list of excluded list items shall be given as a string of characters enclosed by apostrophe characters.

LTK.2.4.2 If there is more than one excluded item in list, consecutive list values shall be separated from each other by the pipe (|) character.

LTK.2.4.3 Excluded list items shall be comprised of any characters except they shall not contain an ampersand (&), less-than (<), greater-than (>), quotes (“), apostrophe (‘), or pipe (|) character.

LTK.2.4.4 If the string of excluded items is not given or are not in the proper format, no excluded items shall be defined.

LTK.3 Input Tokenization

LTK.3.1 The input for ListTokenizer shall be character string.

LTK.3.2 The input shall be divided into list items where is each list item is a substring of the input delimited by the list delimiter character.

LTK.3.3 The list delimiter character shall not be included in the list items it defines.

LTK.3.4 Each list item shall be trimmed of leading and trailing blanks.

LTK.3.5 If the result of trimming blanks results in an empty string, then the list item shall not be indexed.

LTK.3.6 If the length of a list item is less than the defined minimum length, then the list item shall not be indexed

LTK.3.7 The letters in all remaining list items in the list shall be changed to Upper Case letters

LTK.3.7 If an item matches a value in the exclusion list, then the list item shall not be indexed. For purposes of comparison, the excluded items shall be changed to uppercase

LTK.4 Reference Indexing

LTK.4.1 In the case that the final list of items is empty, then the reference shall not be indexed.

LTK.4.2 Otherwise, the reference shall be indexed for each item in the final list of index items in concatenation with each index value produced by other segments in the same <Index>definition including other ListTokenizer functions.

LTK Example:

Suppose a Reference with reference ID “C213” has
an attribute “FirstList” with the value “ABC; CD; X; D/EF; 123”, and
an attribute “SecondList” with the value “XYZ, W4, ?, Copy”.
Then given the index rule
<Index Ident=”X1”>
<Segment Item=”FirstList” Hash=”ListTokenizer(‘,’, 2)”
<Segment Item=”Address” Hash=”ListTokenizer(‘;’, 0, ‘UNK|TBD|?’)”
</Index>

The FirstList value of “ABC; CD; X; D/ef; 123” will be tokenized by semi-colon (;) character as
“ABC”, “CD”, “X”, “D/ef”, and “123”.
Because the item “X” is fewer than two characters, it will be dropped from the list of index items.
After upper casing, the final list of index tokens for Name will be
“ABC”, “CD”, “D/EF”, “123”

The Second List value of “XYZ, W4, ?, Copy”will be tokenized the comma (,) character as
“XYZ”, “W4”, “?”, and “Copy”
All tokens meet the minimum length specification
However, after upper casing and comparing to the exclusion list, the token “?” is dropped leaving the
final list of index items as
“XYZ”, “W4”, and “COPY”

Finally, the reference C213 is indexed by the following 12 tokens
“ABCXYZ”
“CDXYZ”
“D/EFXYZ”
“123XYZ”
“ABCW4”
“CDW4”
“D/EFW4”
“123W4”
“ABCCOPY”
“CDCOPY”
“D/EFCOPY”
“123COPY

- **Pair List Index Function (PLI)**

Overview

The primary purpose of the Pair List Index Function (“ListTokenizerPV”) is to support indexing (blocking) for the Property-Value List Comparator (ListOverlapPV). ListTokenizerPV is parsing function for use in an index generator (<Index> element) that takes a character string given a value in an entity reference as its input and creates a reference index entry for specified substrings (tokens) of the input string. The input string is assumed represent a list of property-value pairs where the pairs are separated from each other by a designated character (list delimiter), and within each pair, the property name is separated from the property value by a different designed character (pair delimiter). ListTokenizerPV has three control parameters. The first control parameter specifies the list delimiting character, the second parameter the pair delimiting character, and the third control parameter is an optional list of property placeholder values.

Semantics

The operation of ListTokenizerPV is as follows:

1. Each input string is separated into a list of property-value pairs based on the list delimiter character given in the second control parameter.
2. A property-value pair is deemed “empty” is it contains no characters or all blank characters. Empty property-value pairs are dropped from the list.
3. Next each property value pair is separated into its property name and corresponding property value based on the pair delimiter character given in the second control parameter. If either the property name or property value is empty, the property-value pair is dropped from the list.
4. Next each property value is compared to the list of placeholder property values given in third control parameter. If the property value of a property-value pair matches one of the placeholder values, the property-value pair is dropped from the list.
5. Next, all duplicate property-value pairs within the same input string are dropped.
6. Finally, the reference is then indexed for each remaining token in concatenation with all other hash values produced within the same index generator.

Syntax

The syntax for ListTokenizerPV is “ListTokenizerPV (C1, C2, C3)” where

C1 is a single character enclosed in apostrophes defining the list delimiter

C2 is a single character enclosed in apostrophes defining the pair delimiter

C3 is a list of placeholder values enclosed in apostrophes. The placeholder values in the list are separated by the pipe (|) character.

Parameter C3 is optional, and if omitted the syntax is “ListTokenizerPV (C1, C2)”

For example, if the list delimiter is a comma character (,), the pair delimiter is a colon character (:), and the placeholder values are “UNKNOWN” and “UNK”, then the Pair List Index Function would be encoded in the element of an <Index> element as

<Segment Item=”VarOne”, Hash=“ ListTokenizerPV(‘,’, ‘:’, ‘UNKNOWN|UNK’)”

If no placeholder values are specified, then the Pair List Index Function would be encoded in the element of an <Index> element as

<Segment Item=”VarOne”, Hash=“ ListTokenizerPV(‘,’, ‘:’)”

PLI Requirements

PLI.1 Syntax for the Pair List Index Function
- PLI.1.1 The name of the function shall be “ListTokenizerPV”
- PLI.1.2 The name shall not be case sensitive

PLI.2 Control parameters for ListTokenizerPV
- PLI.2.1 The comparator shall have up to 3 control parameters C1, C2, and C3 represented as “ListTokenizerPV(C1, C2, C3)”
- PLI.2.2 The first control parameter C1 shall specify the list delimiter character.
  - PLI.2.2.1 The list delimiter character shall be enclosed in apostrophes.
  - PLI.2.2.2 The list delimiter character shall be any character except it shall not be an ampersand (&), less-than (<), greater-than (>), quotes (“), apostrophe (‘), or pipe (|) character.
  - PLI.2.2.3 If the list delimiter character is not given or not in the proper format, the system shall default its value to the comma (,) character.
- PLI.2.3 The second control parameter C1 shall specify the pair delimiter character.
  - PLI.2.3.1 The pair delimiter character shall be given as a single character enclosed by apostrophe characters.
  - PLI.2.3.2 The pair delimiter character shall be any character except it shall not be an ampersand (&), less-than (<), greater-than (>), quotes (“), apostrophe (‘), or pipe (|) character, and it shall not be the same as the list delimiter character.
  - PLI.2.3.3 If the pair delimiter character is not given or not in the proper format, the system shall default its value to the colon (:) character.
- PVC.2.4 The third control parameter shall be the list of property placeholder values.

PVC.2.4.1 The list of placeholder values shall be given as a string of characters enclosed by apostrophe characters.

PVC.2.4.2 If there is more than one placeholder value in list, consecutive values shall be separated from each other by the pipe (|) character.

PVC.2.4.3 Placeholder values shall be comprised of any characters except they shall not contain an ampersand (&), less-than (<), greater-than (>), quotes (“), apostrophe (‘), or pipe (|) character.

PVC.2.4.4 If the string of placeholder values is not given or not in the proper format, no placeholder values will be defined.

PLI.3 Generating the tokes from the property-value pair list.

PLI.3.1 The input for ListTokenizerPV shall be character string.

PLI.3.2 The input string shall be interpreted as a list of property-value pairs where the pairs are separated from each other by the list delimiter character either given in the first control parameter or if not given, by the default delimiter.

PLI.3.3 Each property-value pair shall be trimmed of leading and trailing blanks, all letters shall be changed to uppercase.

PLI.3.4 If the result of trimming blanks results is an empty string, then the property-value pair shall not generate a token.

PLI.3.5 If the result of dropping empty property-value pairs results in an empty list, then the ListIndex function shall not index the reference, and no tokens will be produced.

PLI.3.6 Otherwise, if there are non-empty items in the list of property-value pairs, then each pair remaining in the list shall be separated in a property name value and its corresponding property value according to the pair delimiter character given in the first control parameter, or if not given, by the default pair delimiter.

PLI.3.7 Each property name and each property value shall be trimmed of leading and trailing blacks.

PLI.3.8 If either the property name or the property value of a pair is empty, then the property-value pair shall not generate a token.

PLI.3.9 If placeholder values were given, and if a property value of a property-value pair is one of the placeholder values, the property-value pairs shall not generate a token

PLI.3.10 Duplicate tokens shall be removed from the final list of generated tokens.

PLI.3.11 The output of the ListTokenizerPV shall be the final list of surviving tokens which in some cases may be an empty list if no tokens were generated.

PLI.3.12 If more than one token is returned, each token will be concatenated with every token generated by other hash functions (different segments) in the same index generator.

PLI Example

Suppose a Reference has with reference ID “C213” has
an attribute “PartCategory” with the value “Cartridge”, and
an attribute “PV_String” with the value
“TYPE:Color,COLOR:Unk,APP:,YIELD:6000,PACKAGE:Box”

Then given
<Index Ident=”X1”>
<Segment Item=”PartCategory” Hash=”SCAN(LR, ALL, 0, ToUpper, SameOrder)”
<Segment Item=”PV_String” Hash=”ListTokenizerPV(‘,’, ‘:’, “UNKNOWN, UNK”)”
</Index>

The first segment will produce the hash value “CARTRIDGE” from the value for the attribute PartCategory.

The pair list for the attribute PV_String has 5 pairs. However, two of the pairs will be dropped. The pair
“COLOR:UNK” will be dropped because the property value “UNK” is in the placeholder value list. The
pari “APP:” will be dropped because the property value is empty.
The final list after upper casing is

1. TYPE:COLOR
2. YIELD:6000
3. PACKAGE:BOX

After concatenation with hash value produced from PartCategory, the reference C213 is indexed by
“CARTRIDGETYPE:COLOR”
“CARTRIDGEYIELD:6000”
“CARTRIDGEPACKAGE:BOX”

- **Matrix Comparator (MC)**

Overview

The purpose of the matrix comparator is to allow users to use one comparator (e.g. LED, EXACT) on one identity attribute partially in a matrix. The identity attribute of each references is separated by the ListParser hash function. Then each part of the identity attribute of two references are compared in a matrix. After each comparison, a score (between 0-1) is given, which represents the probabilistic matching of two parts. In each column of the matrix, if one of matching scores equal to 1, this column counted as a match. Finally count how many columns have been defined as a match and calculate the proportion. If the proportion is larger than a predefined threshold, then the rule signals a

match condition. If the proportion is smaller than the predefined threshold, the rule signals a no-match condition.

Semantics

The comparison of two references is determined in five steps:

1. The value of each identity attribute is separated into properties by ListParser hash function.
2. For each identity attribute, the two values are compared by a designed OYSTER similarity function after set up each property of the two values into a comparison matrix.
3. For each comparison, a score (between 0-1) is given, which represents the probabilistic matching of two properties. In each column of the matrix, if there is one matching score equal to 1, this column counted as a match.
4. After all the properties are compared, count the number of columns which are defined as a match and calculate the proportion. If the proportion is larger than the threshold, a match condition exists between these two values otherwise a no-match condition exists.
5. Keep repeating the process until all the attribute values are compared.

Syntax

There is no Syntax change in the Attributes Script. The predefined threshold is built in.

MC Requirements

MC Example 1:

Example 1 shows a comparison matrix for two values in an identity attribute.

Attribute Item = “PROPERTY_VALUE_STRING”
Value 1: “TYPE:COLORSPHERE,COLOR:BLACK,APPLICATION:LASERJET 3600,YIELD:6000 PG,PACKAGE TYPE:BOX”
Value 2: “TYPE:COLORSPHERE,COLOR:BLACK,APPLICATION:LASERJET 3600,YIELD:6000 PG,PACKAGE TYPE:BOX”
Similarity = EXACT
Threshold = 60% match

	TYPE:COLORSPHERE	COLOR:BLACK	APPLICATION:LASERJET 3600	YIELD:600 PG
TYPE:COLORSPHERE	1.00	0	0	0

COLOR:BLACK	0	1.00	0	0
APPLICATION:LASERJET 3600	0	0	1.00	0
YIELD:6000 PG	0	0	0	1.00
PACKAGE TYPE:BOX	0	0	0	0
(Count how many EXACT we got in each column)	1	1	1	1

Total = 100% Match > 60%

MC Example 2:

Example 2 shows a comparison matrix for two values in an identity attribute.

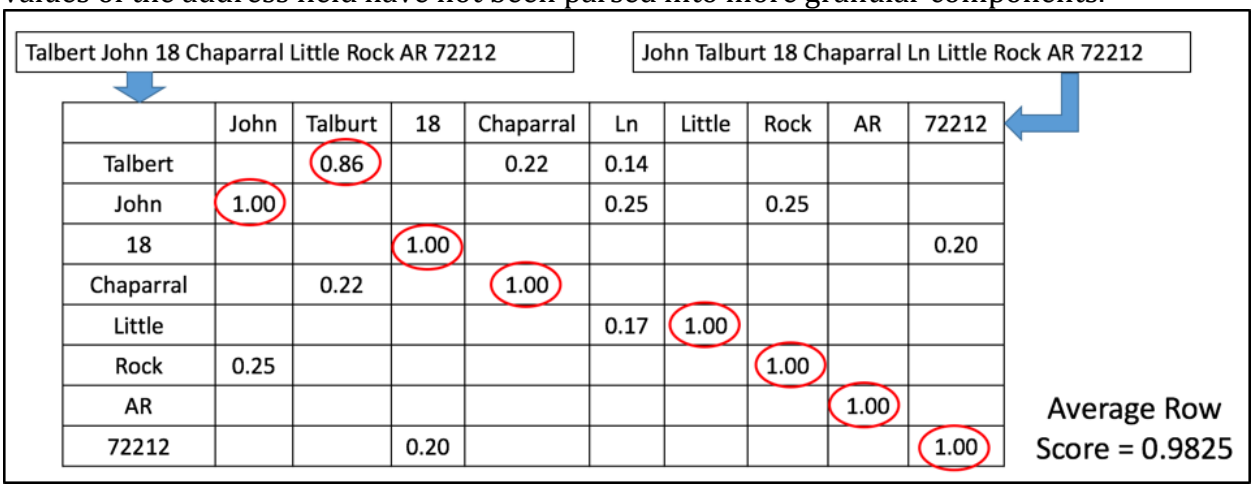
Attribute Item = "PROPERTY_VALUE_STRING"
Value 1: "TYPE:COLORSPHERE,COLOR:BLACK,APPLICATION:LASERJET 3600,YIELD:6000 PG,PACKAGE TYPE:BOX"
Value 2: "TYPE:HIGH RESOLUTION,COLOR:TRI,APPLICATION:INKJET Z13/Z23/Z33,YIELD:275 PG,PACKAGE TYPE:BOX"
Similarity = EXACT
Threshold = 60% match

	TYPE:COLORSPHERE	COLOR:BLACK	APPLICATION:LASERJET 3600	YIELD:6000 PG
TYPE:HIGH RESOLUTION	0.02	0	0	0
COLOR:TRI	0	0.5	0	0
APPLICATION:INKJET Z13/Z23/Z33	0	0	0.05	0
YIELD:275 PG	0	0	0	0.5
PACKAGE TYPE:BOX	0	0	0	0
(Count how many EXACT we got in each column)	0	0	0	0

Total = 20% Match < 60%

`MC Example 3:` Matrix comparator using string tokenization

The following diagram illustrates the concept of a matrix comparator using string tokenization. The attribute being compared is an unstructured "address" field. The two values of the address field have not been parsed into more granular components.



	John	Talburt	18	Chaparral	Ln	Little	Rock	AR	72212
Talbert		0.86		0.22	0.14				
John	1.00				0.25		0.25		
18			1.00						0.20
Chaparral		0.22		1.00					
Little					0.17	1.00			
Rock	0.25						1.00		
AR								1.00	
72212			0.20						1.00

A Matrix Comparator on Tokenized, Semi-Structured Data

In this simple example, the fields are tokenized into the white-space or punctuation delimited substrings (tokens). The tokens are used as labels for rows and columns of a matrix. Each cell of the matrix contains a value representing the similarity between the tokens labeling the row and column of the cell. In the example shown, the similarities are given as the normalized Levenshtein Edit Distance between the two string. For visual clarity, cells with a similarity of 0.00 are left blank.
The basic scheme is to select the highest similarities in each row and column, and use these values to calculate an overall score, in this example simply the unweighted average. However, this example only illustrates the basic technique. Many variations and enhancements are possible. For example, stacking comparators so that each pair of tokens is compared in a different way, e.g. Levenstein, SOUNDEX, Nickname, etc.
Another is to discount (reduce) the scores for matches between short tokens or between high-frequency strings similar to the method used in the standard scoring rule. For example, a match on the token "ST" in an address may be discounted because it is such a common address token.

- **`MatrixTokenizer Function (MXT)`**

`Overview`

The primary purpose of the MatrixTokenizer Index Function ("MatrixTokenizer") is to support indexing (blocking) for the multivalued comparator function Matrix Comparator (MatrixOverlap). MatrixTokenizer is a parsing function to be used as an index generator (<Index> element). The function is to parse a character string into tokens (substrings). The substrings selected as tokens will depend upon the set of delimiter characters provided by the user. MatrixTokenizer has three control parameters. The first control parameter specifies the token delimiting characters. The second control parameter specifies the minimum length of a token to be indexed, and the third is an optional string of tokens to be excluded from the index (the exclusion list).

Semantics

The operation of MatrixTokenizer is as follows:

1. The input string from the reference is separated into a list of tokens based on the delimiter characters provided as the first control parameter.
2. After tokenization, any token that meets one of the following conditions is dropped from the list of tokens to be indexed
    a. The length of the token is less than the minimum token length as specified by the second control parameter (default length = 2)
    b. The token matches one of the excluded tokens in the exclusion list specified by the third control parameter
3. The reference is then indexed for each remaining token in concatenation with all other hash values produced within the same index generator.

Syntax

The syntax for MatrixTokenizer is "MatrixTokenizer(C1, C2, C3)" where

C1 is a string of characters to be used in addition to the blank character as token delimiters.

MatrixTokenizer always uses the blank character as a token delimiter, but the user may specify additional characters through this parameter. The string of additional token delimiters is enclosed by apostrophes

C2 is a positive integer value indicating the minimum length of tokens to index.

C3 is an optional string enclosed in apostrophes representing a list of excluded items separated by the pipe (|) character.

For example, if the additional tokens delimiters are comma character (,) and hyphen (-) character, the minimum token length is 3, and the excluded list of items are "ABC" and

“EFG”, then the MatrixTokenizer function would be encoded in the element of an <Index> element as

<Segment Item=”VarOne”, Hash=“ListTokenizer(‘,-’, 3, ‘ABC|EFG’)”

MXT Requirements

MXT.1 Syntax for the MatrixTokenizer Function
- MXT.1.1 The name of the function shall be “MatrixTokenizer”
- MXT.1.2 The name shall not be case sensitive

MXT.2 Control parameters for MatrixTokenizer
- MXT.2.1 The MatrixTokenizer index function shall have 3 control parameters (arity of 3) C1, C2, and C3 represented as “MatrixTokenizer(C1, C2, C3)”
- MXT.2.2 The first control parameter C1 shall specify the additional tokenizing characters.
  - MXT.2.2.1 The list delimiting characters shall be enclosed in apostrophes.
  - MXT.2.2.2 The list delimiting characters shall be any characters except it shall not be an ampersand (&), less-than (<), greater-than (>), quotes (“), apostrophe (‘), or pipe (|) character.
  - MXT.2.2.3 If no additional list delimiter characters are given or are not given in the proper format, the system shall tokenize only by the blank character.
- MXT.2.3 The second control parameter C1 shall specify the minimum length of a list item to be indexed.
  - MXT.2.3.1 The minimum length shall be given as an integer value.
  - MXT.2.3.2 If the minimum length given is 0, then all tokens shall be indexed.
  - MXT.2.3.3 If the minimum length is not given or is not given in the proper format, the system shall default the minimum length to 2.
- MXT.2.4 The third control parameter shall be the list of excluded token values.
  - MXT.2.4.1 The list of excluded token values shall be given as a string of characters enclosed by apostrophe characters.
  - MXT.2.4.2 If there is more than one excluded token value in list, then consecutive values shall be separated from each other by the pipe (|) character.
  - MXT.2.4.3 Excluded token values shall be comprised of any characters except they shall not contain an ampersand (&), less-than (<), greater-than (>), quotes (“), apostrophe (‘), or pipe (|) character.
  - MXT.2.4.4 If the string of excluded token values is not given or not given in the proper format, no excluded token values shall be defined.

MXT.3 Input Tokenization
- MXT.3.1 The input for MatrixTokenizer shall be a single character string.

MXT.3.2 The input string shall be divided into token values where each token is a substring of the input string delimited by one or more of the token delimiting characters.

MXT.3.3 The token delimiting characters shall not be included in the token values they define.

MXT.3.4 If the length of a token value is less than the user specified minimum length, then the token value shall not be indexed

MXT.3.5 Each letter in the remaining token values shall be changed to an uppercase letter

MXT.3.6 If a token value extracted from the input string matches a token value in the list of excluded token values, then the token value shall not be indexed.

MXT.4 Reference Indexing

MXT.4.1 In the case that the final list of token values is empty, then the reference shall not be indexed.

MXT.4.2 Otherwise, the reference shall be indexed for each token value in the final list of token values in concatenation with each hash value produced by other segments in the same <Index> definition including other MatrixTokenizer functions.

MXT Example:

Suppose a Reference with reference ID "C213" has
an attribute "ContactName" with the value "John J. Smith" and
an attribute "ContactAddress" with the value "123 S. Oak ST, Anyville, CA".
Also suppose both the ContactName and ContactAddress attributes are to be indexed using the MatrixTokenizer function where the both attributes have

- Token delimiting characters as blank, comma, and period
- The minimum token length to be indexed is 2, and

Where the ContactAddress attribute

- Should not index the tokens "ST", and "AVE".

Then given

<Index Ident="X1">
    <Segment Item="ContactName" Hash="MatrixTokenizer(',.', 2)"
    <Segment Item="ContactAddress" Hash="ListTokenizer(',.', 2, 'ST|AVE')"
</Index>

The ContactName string "John J. Smith" will be tokenized by blank, comma, and period as "John", "J", "Smith"
Because the token "J" has fewer than two characters, it will be dropped from the list of index items.
After upper casing, the final list of index tokens for ContactName will be
"JOHN", "SMITH"

The ContactAddress string of “123 S. Oak ST, Anyville, CA” will be tokenized by blank, comma, and period as “123”, “S”, “Oak”, “ST”, “Anyville”, and “CA”.
The token “S” is dropped because it does not meet the minimum length requirement. After upper casing and comparing to the exclusion list, the token “ST” is dropped leaving the final list of ContactAddress index items as “123”, “OAK”, “ANYVILLE”, and “CA”.

Finally, the reference C213 is indexed by the following 8 index values
“JOHN123”
“JOHNOAK”
“JOHNANYVILLE”
“JOHNCA”
“JOHN123”
“JOHNOAK”
“JOHNANYVILLE”
“JOHNCA”
“JOHNCA”

- **Placeholder Value Filter (PHF)**

Overview

The purpose of Placeholder Value Filter (PVF) is to exclude attribute value defined as placeholder values (a.k.a. default) from the matching process. The placeholder value filter is implemented as a DataPrep function.

Semantics

If a rule term implements the Placeholder Value Filter, the two attribute values to be compared are first checked against a user-defined list of placeholder values. If the attribute value is found in the placeholder value filter provided by the user, the filter will replace the attribute value with an empty string, otherwise the filter leaves the attribute value unchanged.

Syntax

The placeholder value filter is implemented as a new DataPrep function named “PlaceHolderFilter”. The PlaceHolderFilter function will take a single argument. The argument value will be a string representing a list of placeholder values separated by the pipe character (|).

For example, the following syntax would be used in rule term to filter three placeholder values “UNKNOWN”, “UNK”, and “?” for the attributed named “PartNbr”

<Item Name=”PartNbr” DataPrep=”PlaceHolderFilter(UNKNOWN|UNK|?)” Similarity=Exact>

PVF Requirements

PVF.1 The syntax of the placeholder value filter

- PVF.1.1 The DataPrep attribute of the <Term> Element shall recognize “PlaceHolderFilter(value_list)” as a valid function signature.
- PVF.1.2 The PlaceHolderFilter() function shall have one argument comprising a single string.
- PVF.1.3 The string argument of the PlaceHolderFilter() function shall be interpreted as a list of placeholder values where the placeholder values are separated by the pipe (|) character.
- PVF.1.4 The argument of the PlaceHolderFilter() function shall not support text qualification, therefore a placeholder value shall not include a pipe character as part of its value.
- PVF.1.5 Each placeholder value parsed from the string argument of the PlaceHolderFilter() function shall be trimmed of leading and training blanks and all letters in each placeholder value shall be changed to upper case.

PVF.2 The operation of the placeholder value filter

- PVF.2.1 When “PlaceHolderFilter(value_list)” is given as the value of the DataPrep attribute of a <Term> element of a matching rule, each value of the OYSTER attribute defined by the XML ITEM attribute of the rule term shall be compared to the list placeholder values.
- PVF.2.2 If the value of the OYSTER attribute matches one of the placeholder values, the value of the OYSTER attribute will be replaced with the empty string (“”), otherwise the actual value of the OYSTER attribute will be passed to the matching rule without change.
- PVF.2.3 For purposes of comparing the OYSTER attribute value to the list of placeholder values, all letters will be changed to upper case.

PVF Example:

In this example, the OYSTER attribute StudentFirstName is to be filtered for two placeholder values. One placeholder value is “NFN” and the other placeholder value is “N/A”. The two values are given to the PlaceHolderFilter() function as the string argument “NFN|N/A”.

```
<IdentityRules>
    <Rule Ident="1">
        <Term Item="StudentFirstName" DATAPREP="PLACEHOLDERFILTER(NFN|N/A)"
```

```
                SIMILARITY="EXACT" />
        <Term Item="StudentLastName" DATAPREP="SCAN(LR, ALPHA, 0, ToUpper, SameOrder)"
                SIMILARITY="SOUNDEX"/>
    </Rule>
<IdentityRules>
```