Understanding Sun Master Index Processing (Repository)

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 820–2667–15
December 2008

081219@21808

Contents

Understanding Sun Master Index Processing (Repository)

The topics listed here provide conceptual information about standard processing logic for a master index application, the flow of data through a master index application, the database structure, and the dynamic Java API.

Note that Java CAPS includes two versions of Sun Master Index. Sun Master Index (Repository) is installed in the Java CAPS repository and provides all the functionality of previous versions in the new Java CAPS environment. Sun Master Index is a service-enabled version of the master index that is installed directly into NetBeans. It includes all of the features of Sun Master Index (Repository) plus several new features, like data als

nalysis, data cleansing, data loading, and an improved Data Manager GUI. Both products are components of the Sun Master Data Management (MDM) Suite. This document relates to Sun Master Index (Repository) only.

- Understanding Master Index Operational Processes (Repository)
- The Master Index Database Structure (Repository)
- Working with the Master Index Java API (Repository)
- Master Index Match Types and Field Names (Repository)

Related Topics

Several topics provide information and instructions for implementing and using a Repository-based master index application. For a complete list of topics related to working with Sun Master Index (Repository), see "Related Topics" in *Developing Sun Master Indexes (Repository)*.

About Sun Master Index (Repository)

Sun Master Index provides a flexible framework that allows you to create matching and indexing applications called *enterprise-wide master index applications*. It is an application building tool to help you design, configure, and create a master index application that will uniquely identify and cross-reference the business objects stored in your system databases. Business objects can be any type of entity for which you store information, such as customers, patients, vendors, businesses, inventory, and so on.

When you create a master index application, custom database scripts and a custom Java API are automatically generated based on the information you specify in the wizard and the configuration files. Both the database scripts and API are derived from the object structure you define. For example, if you create a master index application with an Address object, the database scripts will define a table named SBYN_ADDRESS and one named SBYN_ADDRESSSBR. The Java API will include a class named `AddressObject` that includes "get" methods for each field you defined for the Address object.

Understanding Master Index Operational Processes (Repository)

Master index applications created by Sun Master Index use a custom Java API library to transform and route data into and out of the master index database. In order to customize the way the Java methods transform the data, it is helpful to understand the logic of the primary processing functions and how messages are typically processed through the master index system.

The following topics describe and illustrate the processing flow of messages to and from the master index application, providing background information to help design and create custom processing rules for your implementation.

- "Learning About Master Index Message Processing (Repository)" on page 6
- "Master Index Inbound Message Processing Logic (Repository)" on page 13
- "Master Index Custom Decision Point Logic (Repository)" on page 19
- "Master Index Primary Function Processing Logic (Repository)" on page 22

Learning About Master Index Message Processing (Repository)

This section provides a summary of how inbound and outbound messages can be processed in a master index application. A master index application cross-references records stored in various computer systems of an organization and identifies records that might represent or do represent the same object. The master index application uses Java CAPS components to

connect to and share data with these external systems. The following topics provide information about inbound and outbound message processing.

- "Master Index Inbound Message Processing (Repository)" on page 7
- "Master Index Outbound Message Processing (Repository)" on page 9

Figure 1 illustrates the flow of information through a master index application that includes a JMS Topic to which updates to the index are published.

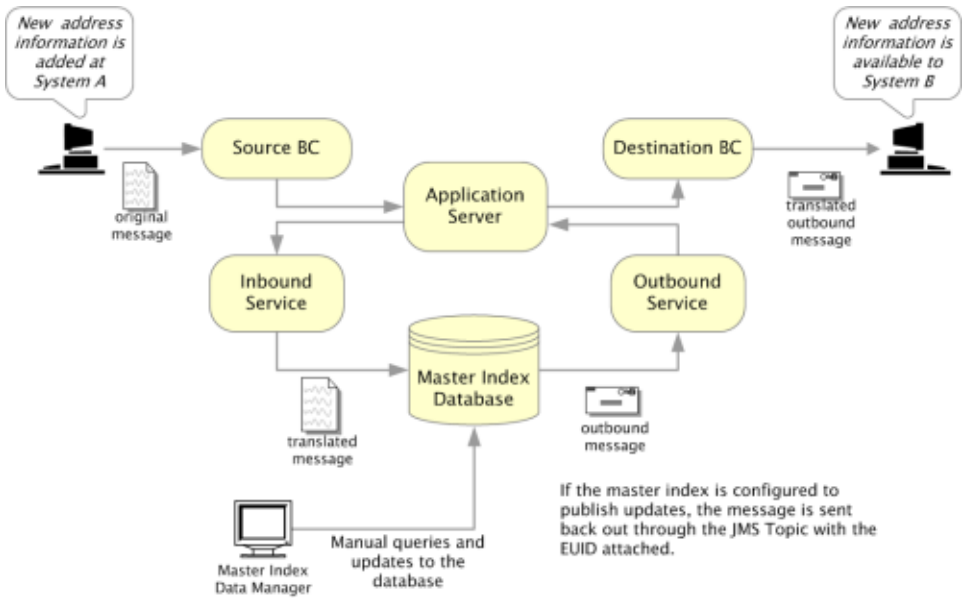


FIGURE 1 Master Index Processing Flow

Master Index Inbound Message Processing (Repository)

An inbound message refers to the transmission of data from external systems to the master index database. These messages can be sent into the database through a number of Services. Inbound messages can be stored in journal files and tracked in the log files. The steps below describe how inbound messages are processed.

1. Messages are created in an external system, and the enveloped message is transmitted to the Enterprise Service Bus through that system's eWay.
2. The Enterprise Service Bus identifies the message and the appropriate Service to which the message should be sent. The message is then routed to the appropriate Service for processing.

3. The message is modified into the appropriate format for the master index database, and validations are performed against the data elements of the message to ensure accurate delivery. The message is validated using the Java code in the Service's Collaboration and other information stored in the master index configuration files.
4. If the message was successfully transmitted to the database, the appropriate changes to the database are processed.
5. After the master index application processes the message, an enterprise-wide universal identifier (EUID) is returned (for either a new or updated record). That EUID can be sent back out through a different Service to the external system. Alternatively, the entire updated message can be published using the outbound OTD (see "Master Index Outbound Message Processing (Repository)" on page 9).

Figure 2 below illustrates the flow of a message inbound to an Sun Master Index application.

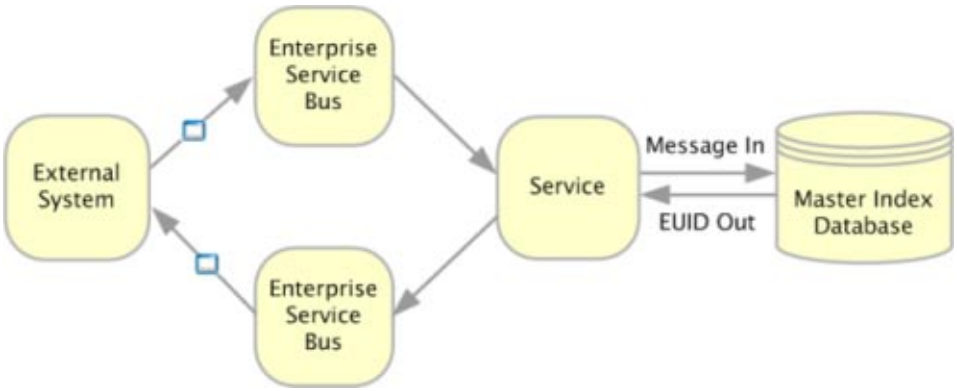


FIGURE 2 Inbound Message Processing Data Flow

About Inbound Messages

The format of inbound messages is defined by the inbound OTD, located in the client project for each external system. The inbound messages can either conform to the required format for the master index application or they can be mapped to the correct format in the Collaboration. The required format depends on how the object structure of the master index application is defined (in the Object Definition file in the master index project).

In addition to the objects and fields defined in the Object Definition file, you can include standard master index application fields. For example, you must include the system and local ID fields and you can also include transaction information, such as the date and time of the transaction, the transaction type, user ID, and so on. If you want to use transaction information from the source systems, be sure to include the fields in the OTD.

Transaction fields include the following:

- MessageId
- EventTypeCode
- UserId
- AssigningSystem
- Source
- Department
- TerminalId
- DateOfEvent
- TimeOfEvent

  If you do not send these fields into the master index application, default values are used (for example, the date and time fields default to the date and time the transaction is processed by the master index application). The inbound OTD also includes the standard Java methods `marshal`, `unmarshal`, `marshalToString`, `unmarshalFromString`, `marshalToBytes`, `unmarshalFromBytes`, and `reset`. For information about the default OTD for Sun Master Patient Index, see *Understanding Sun Master Patient Index Configuration*.

Master Index Outbound Message Processing (Repository)

An outbound message refers to the transmission of data from the master index database to any external system. Messages can be transmitted from the master index application in two ways. The first way is by transmitting the output of `executeMatch` (an EUID). This is described in"Master Index Inbound Message Processing (Repository)" on page 7 and is only used for messages received from external systems.

The second way is by publishing updates from the master index application to a JMS Topic, which allows you to publish complete, updated single best records (SBRs) to any system subscribing to that topic. When updates are made to the database from either external systems or the Enterprise Data Manager, the master index application generates outbound messages in the format of the outbound OTD.

Note – A Sun master index application only publishes the outbound message to JMS Topics and not to JMS Queues.

This section describes how the second type of outbound message is processed. A JMS Topic must be defined in the Connectivity Maps for the master index server project and the appropriate client projects for this type of processing to occur.

1. When a message is received from an external system or data is entered through the Enterprise Data Manager (EDM), the master index application processes the information and generates an XML message, which is then sent to the JMS Topic that is configured to publish messages from the master index application.
2. Messages published by the JMS Topic are processed through a Service whose Collaboration uses the master index outbound OTD. This Service modifies the message into the appropriate format.
3. The Enterprise Service Bus identifies the message and the external systems to which it should be sent and then routes the message for processing by an external system eWay.

Note – Outbound messages are stored and tracked in the Enterprise Service Bus journal and log files.

Figure 3 below illustrates the flow of data for a message outbound from a master index application.

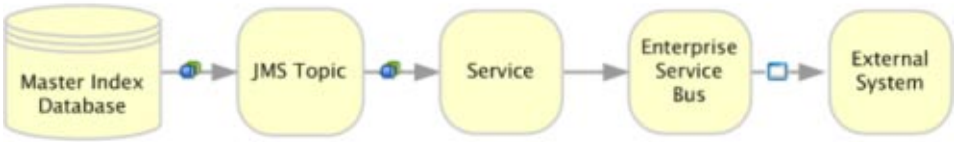


FIGURE 3 Outbound Message Processing Data Flow

About Outbound Messages

When you generate the master index application, an outbound OTD is created, the structure of which is based on the object definition. This OTD is used to publish changes in the master index database to external systems using a JMS Topic. The output of the `executeMatch` process is an EUID of the new or updated record. You can use this EUID to obtain additional information and configure a Collaboration and Service to output the data, or you can process all updates in the master index application through a JMS Topic using the outbound OTD.

Outbound OTD Structure

The outbound OTD is named after the master index application (for example, OUTCompany or OUTPerson). This OTD contains eight primary nodes: Event, ID, SBR, and the standard Java methods `marshal`, `unmarshal`, `marshalToString`, `unmarshalFromString`, `marshalToBytes`, `unmarshalFromBytes`, and `reset`. The Event field is populated with the type of transaction that created the outbound message, and the ID field is populated with the unique identification code of that transaction. The SBR node is the portion of the OTD created from the Object Definition file. In the sample, the outbound OTD publishes messages in XML format. Table 1 describes the components of the SBR portion of the outbound OTD.

TABLE 1 Outbound OTD SBR Nodes

Node	Description
EUID	The EUID of the record that was inserted or modified.
Status	The status of the record.
CreateFunction	The date the record was first created.
CreateUser	The logon ID of the user who created the record.
UpdateSystem	The processing code of the external system from which the updates to an existing record originated.
ChildType	The name of the parent object.
CreateSystem	The processing code of the external system from which the record originated.
UpdateDateTime	The date and time the record was last updated.
CreateDateTime	The date and time the record was created.
UpdateFunction	The type of function that caused the record to be modified.
RevisionNumber	The revision number of the record.
UpdateUser	The logon ID of the user who last updated the record.
SystemObject	The object's local identifier in a specified system. This field has three sub-fields: **LID**: The local ID assigned to the person in the system of origin. **System**: The processing code of the system of origin. **Status**: The status of the local ID in the enterprise record.
Object_Name	The fields in this node are defined by the object structure (as defined in the Object Definition file). It is named by the parent object and contains all fields and child objects defined in the structure. This section varies depending on the customizations made to the object structure.

Outbound Message Trigger Events

When outbound messaging is enabled, the following transactions automatically generate an outbound message that is sent to the JMS Topic (if a JMS Topic has been incorporated into the master index project).

- Activating a system record
- Activating an enterprise record
- Adding a system record
- Creating an enterprise record
- Deactivating a system record
- Deactivating an enterprise record
- Merging an enterprise record
- Merging a system record
- Transferring a system record
- Unmerging an enterprise record
- Unmerging a system record
- Updating an enterprise record
- Updating a system record

Sample Outbound Message

The following text is a sample outbound message for a master index application based on a master person index. Your outbound messages might appear differently depending on how you configure the client project connectivity components.

```
<?xml version="1.0" encoding="UTF-8"?>
<OutMsg Event="UPD" ID="00000000000000044005">
<SBR EUID="1000008001"  Status="active" CreateFunction="Add" ChildType="Person"
CreateSystem="System" UpdateFunction="Update" RevisionNumber="5" CreateUser="eview"
UpdateSystem="System" UpdateDateTime="12/16/2003 17:40:44"
CreateDateTime="12/16/2003 17:36:58" UpdateUser="eview">
<SystemObject SystemCode="CBMC" LID="434900094" Status="active">
</SystemObject>
<Person PersonId="00000000000000017000" PersonCatCode="PT" LastName="WRAND"
FirstName="ELIZABETH" MiddleName="SU" Suffix="" Title="PHD" DOB="12/12/1972 00:00:00"
Death="" Gender="F" MStatus="M" SSN="555665555" Race="B" Ethnic="23" Religion="AG"
Language="ENGL" SpouseName="MARCUS" MotherName="TONIA" MotherMN="FLEMING"
FatherName="JOSHUA" Maiden="TERI" PobCity="KINGSTON" PobState="" PobCountry="JAMAICA"
VIPFlag="N" VetStatus="N" FnamePhoneticCode="E421" LnamePhoneticCode="RAN"
MnamePhoneticCode="S250" MotherMNPhoneticCode="FLANANG" MaidenPhoneticCode="TAR"
SpousePhoneticCode="M622" MotherPhoneticCode="T500" FatherPhoneticCode="J200"
DriversLicense="CT111333111" DriversLicenseSt="CT" Dod="" DeathCertificate=""
Nationality="USA" Citizenship="USA" PensionNo="" PensionExpDate="" RepatriationNo=""
DistrictOfResidence="" LgaCode="" MilitaryBranch="NONE" MilitaryRank="NONE"
MilitaryStatus="NONE" StdLastName="WRAND" StdMiddleName="SUSAN">
<Phone PhoneId="00000000000000011001" PhoneType="CC" Phone="9895558768" PhoneExt="">
```

```
</Phone>
<Phone PhoneId="0000000000000011000" PhoneType="CH" Phone="9895554687" PhoneExt="">
</Phone>
<Address AddressId="0000000000000011001" AddressType="H" AddressLine1="1220 BLOSSOM
STREET" AddressLine2="UNIT 12" AddressLine3="" AddressLine4="" City="SHEFFIELD"
StateCode="CT" PostalCode="09877" PostalCodeExt="" County="CAPEBURR"
CountryCode="UNST" HouseNumber="1220" StreetDir="" StreetName="BLOSSOM"
StreetNamePhoneticCode="BLASAN" StreetType="St">
</Address>
</Person>
</SBR>
</OutMsg>
```

Master Index Inbound Message Processing Logic (Repository)

When records are transmitted to the master index application, one of the "execute match" methods is usually called and a series of processes are performed to ensure that accurate and current data is maintained in the database. The execute match methods include `executeMatch`, `executeMatchUpdate`, `executeMatchDupRecalc`, and `executeMatchUpdateDupRecalc`. The EDM uses `executeMatchGui`. For more information about how these methods differ, refer to the Javadocs.

You can define these processes in the Collaboration using the functions defined in the customized method OTD. The steps performed by the standard `executeMatch` method are outlined below, and the diagrams on the following pages illustrate the message processing flow. The processing steps performed in your environment might vary from this depending on how you customize the Collaboration and Connectivity Map.

The steps outlined below refer to the following parameters in the Threshold file. They are described in "Threshold Configuration (Repository)" in *Understanding Sun Master Index Configuration Options (Repository)*).

- OneExactMatch parameter
- SameSystemMatch parameter
- MatchThreshold parameter
- DuplicateThreshold parameter
- Update mode

Note – There are several decision points in the match process that can be defined by custom logic using custom plug-ins. These decision points are not listed in the below steps, which describe the default processing logic."Master Index Custom Decision Point Logic (Repository)" on page 19 provides the same steps as below with the decision points included.

1. When a message is received by the master index application, a search is performed for any existing records with the same local ID and system as those contained in the message. This search only includes records with a status of **A**, meaning only active records are included. If a matching record is found, an existing EUID is returned.
2. If an existing record is found with the same system and local ID as the incoming message, it is assumed that the two records represent the same object. Using the EUID of the existing record, the master index application performs an update of the record's information in the database.
   - If the update does not make any changes to the object's information, no further processing is required and the existing EUID is returned.
   - If there are changes to the object's information, the updated record is inserted into the database and the changes are recorded in the sbyn_transaction table.
   - If there are changes to key fields (that is, fields used for matching or for the blocking query) and the update mode is set to pessimistic, potential duplicates are reevaluated for the updated record.
3. If no records are found that match the record's system and local identifier, a second search is performed using the blocking query. A search is performed on each of the defined query blocks to retrieve a candidate pool of potential matches.

   Each record returned from the search is weighted using the fields defined for matching in the inbound message.
4. After the search is performed, the number of resulting records is calculated.
   - If a record or records are returned from the search with a matching probability weight above the match threshold, the master index application performs exact match processing (see Step 5).
   - If no matching records are found, the inbound message is treated as a new record. A new EUID is generated and a new record is inserted into the database.
5. If records were found within the high match probability range, exact match processing is performed as follows:
   - If only one record is returned from this search with a matching probability that is equal to or greater than the match threshold, additional checking is performed to verify whether the records originated from the same system (see Step 6).

- If more than one record is returned with a matching probability that is equal to or greater than the match threshold and exact matching is set to false, then the record with the highest matching probability is checked against the incoming message to see if they originated from the same system (see Step 6).
- If more than one record is returned with a matching probability that is equal to or greater than the match threshold and exact matching is true, a new EUID is generated and a new record is inserted into the database.
- If no record is returned from the database search, or if none of the matching records have a weight in the exact match range, a new EUID is generated and a new record is inserted into the database.

Note – Exact matching is determined by the OneExactMatch parameter, and the match threshold is defined by the MatchThreshold parameter. For more information about these parameters, see "Threshold Configuration (Repository)" in *Understanding Sun Master Index Configuration Options (Repository)*.

---

6. When records are checked for same system entries, the master index application tries to retrieve an existing local ID using the system of the new record and the EUID of the record that has the highest match weight.
   - If a local ID is found and same system matching is set to true, a new record is inserted and the two records are considered to be potential duplicates. These records are marked as same system potential duplicates.
   - If a local ID is found and same system matching is set to false, it is assumed that the two records represent the same object. Using the EUID of the existing record, the master index application performs an update, following the process described in Step 2 above.
   - If no local ID is found, it is assumed that the two records represent the same object and an assumed match occurs. Using the EUID of the existing record, the master index application performs an update, following the process described in Step 2 above.
7. If a new record is inserted, all records that were returned from the blocking query are weighed against the new record using the matching algorithm. If a record is updated and the update mode is pessimistic, the same occurs for the updated record. If the matching probability weight of a record is greater than or equal to the potential duplicate threshold, the record is flagged as a potential duplicate (for more information about thresholds and the update mode, see "Threshold Configuration (Repository)" in *Understanding Sun Master Index Configuration Options (Repository)*).

The following flow charts provide a visual representation of the processes performed in the default configuration. Figure 4 and Figure 5 represent the primary flow of information. Figure 6 expands on update procedures illustrated in Figure 4 and Figure 5.

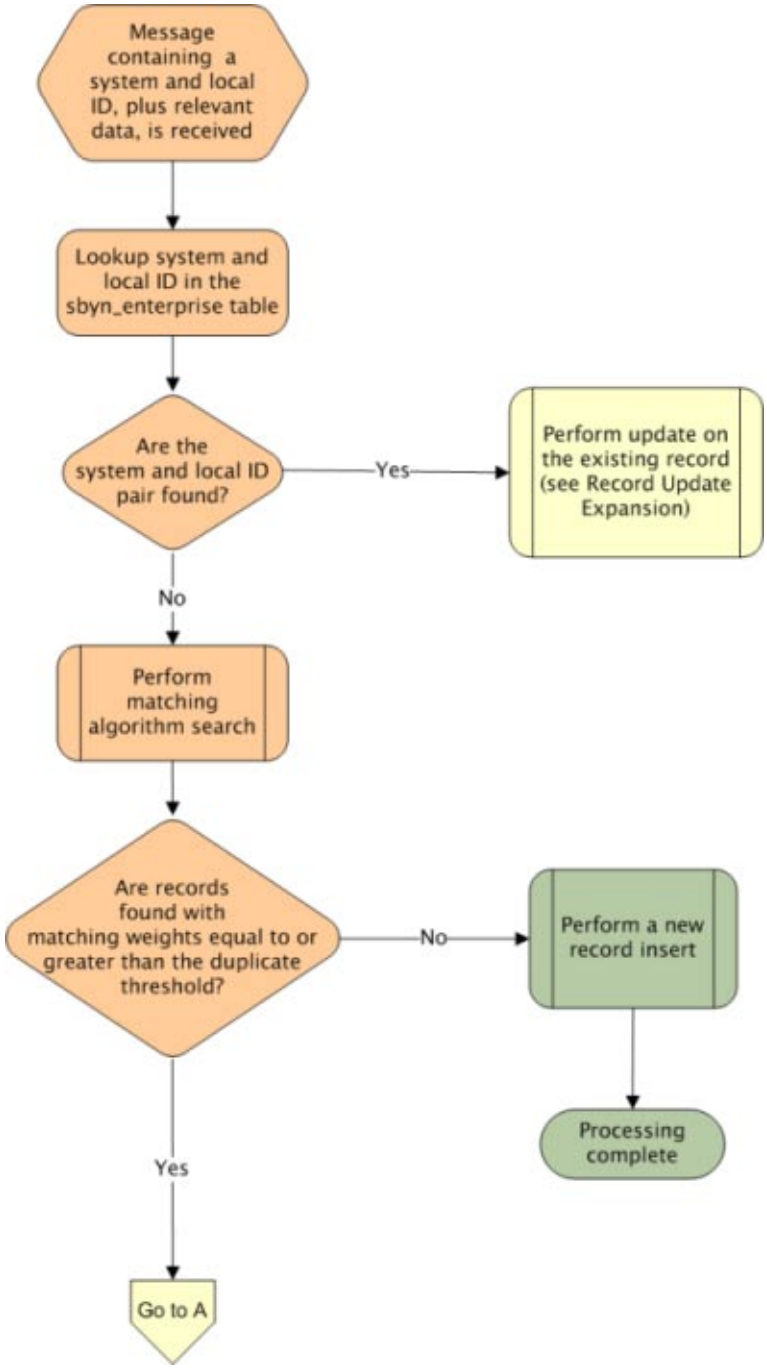


FIGURE 4 Inbound Message Processing

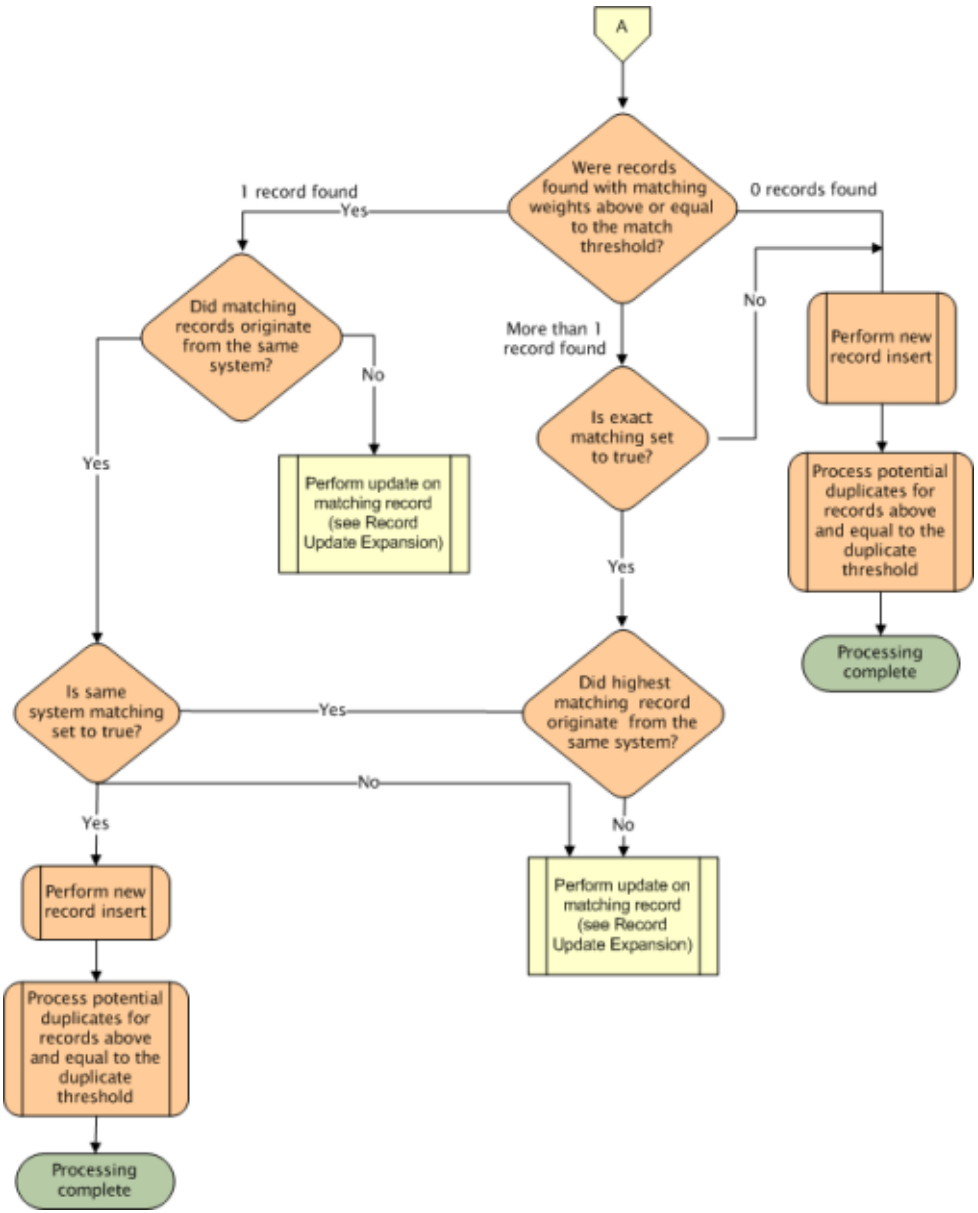


FIGURE 5 Inbound Message Processing (continued)

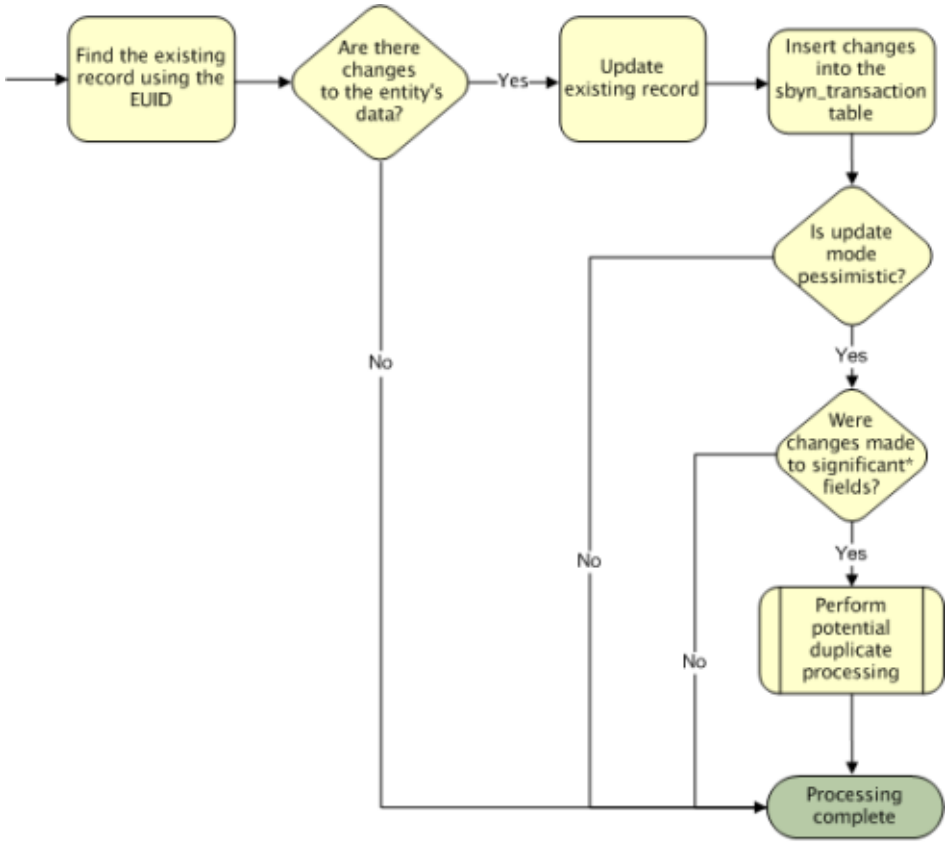


FIGURE 6 Record Update Expansion

Master Index Custom Decision Point Logic (Repository)

You can customize the way the execute match methods process inbound messages by defining custom plug-ins that include decision-point methods. There are several decision points in the match process that can be defined by custom logic using custom plug-ins. This topic describes the standard inbound processing logic as described in "Master Index Inbound Message Processing Logic (Repository)" on page 13, but also includes how the decision-point methods alter the process. If no custom logic is defined, the decisions default to false, and processing is identical to that described in "Master Index Inbound Message Processing Logic (Repository)" on page 13.

For more information about the methods and plug-ins, see "Master Index Match Processing Logic Plug-ins (Repository)" in *Developing Sun Master Indexes (Repository)*. For detailed information about the methods, see the Javadocs provided with Sun Master Index. The methods are contained in the `ExecuteMatchLogics` class in the package `com.stc.eindex.master`.

1. When a message is received by the master index application, a search is performed for any existing records with the same local ID and system as those contained in the message. This search only includes records with a status of **A**, meaning only active records are included. If a matching record is found, an existing EUID is returned.
2. If an existing record is found with the same system and local ID as the incoming message, it is assumed that the two records represent the same entity. Using the EUID of the existing record, the master index application performs an update of the record's information in the database.

   Custom plug-in decision point: If `disallowUpdate` is set to true, the update is not allowed and a `MatchResult` object is returned with a result code of 12. If `disallowUpdate` is set to false and `rejectUpdate` is set to true, the update is not allowed and a `MatchResult` object is returned with a result code of 13.

   - If the update does not make any changes to the object's information, no further processing is required and the existing EUID is returned.
   - If there are changes to the object's information, the updated record is inserted into database, and the changes are recorded in the sbyn_transaction table.
   - If there are changes to key fields (that is, fields used for matching or for the blocking query) and the update mode is set to pessimistic, potential duplicates are reevaluated for the updated record.
3. If no records are found that match the record's system and local identifier, a second search is performed using the blocking query. A search is performed on each of the defined query blocks to retrieve a candidate pool of potential matches.

   Custom plug-in decision point: If `bypassMatching` is set to true, the search steps are bypassed and, if `disallowAdd` is set to false, a new record is added. If `disallowAdd` is set to true, the record is not added and a `MatchResult` object is returned with a result code of 11.

   Each record returned from the search is weighted using the fields defined for matching in the inbound message.
4. After the search is performed, the number of resulting records is calculated.
   - If a record or records are returned from the search with a matching probability weight above the match threshold, the master index application performs exact match processing (see Step 5).
   - If no matching records are found, the inbound message is treated as a new record. A new EUID is generated and a new record is inserted into the database.
5. If records were found within the high match probability range, exact match processing is performed as follows:

- If only one record is returned from this search with a matching probability that is equal to or greater than the match threshold, additional checking is performed to verify whether the records originated from the same system (see Step 6).
- If more than one record is returned with a matching probability that is equal to or greater than the match threshold and exact matching is set to false, then the record with the highest matching probability is checked against the incoming message to see if they originated from the same system (see Step 6).
- If more than one record is returned with a matching probability that is equal to or greater than the match threshold and exact matching is true, a new EUID is generated and a new record is inserted into the database.

  Custom plug-in decision point: If `disallowAdd` is set to true, the new record is not inserted and a `MatchResult` object is returned with a result code of 11.
- If no record is returned from the database search, or if none of the matching records have a weight in the exact match range, a new EUID is generated and a new record is inserted into the database.

  Custom plug-in decision point: If `disallowAdd` is set to true, the new record is not inserted and a `MatchResult` object is returned with a result code of 11.

6. When records are checked for same system entries, the master index application tries to retrieve an existing local ID using the system of the new record and the EUID of the record that has the highest match weight.
   - If a local ID is found and same system matching is set to true, a new record is inserted and the two records are considered to be potential duplicates. These records are marked as same system potential duplicates.

     Custom plug-in decision point: If `disallowAdd` is set to true, the new record is not inserted and a `MatchResult` object is returned with a result code of 11.
   - If a local ID is found and same system matching is set to false, it is assumed that the two records represent the same entity. Using the EUID of the existing record, the master index application performs an update, following the process described in Step 2 above.

     Custom plug-in decision point: If `rejectAssumedMatch` is set to true and `disallowAdd` is set to false, a new record is added; if `disallowAdd` is set to true, the new record is not inserted and a `MatchResult` object is returned with a result code of 11. If `rejectAssumedMatch` and `disallowUpdate` are set to false, the existing record is updated; if `disallowUpdate` is set to true, the update is not performed and a `MatchResult` object is returned with a result code of 13.
   - If no local ID is found, it is assumed that the two records represent the same entity and an assumed match occurs. Using the EUID of the existing record, the master index application performs an update, following the process described in Step 2 above.

     Custom plug-in decision point: If `rejectAssumedMatch` is set to true and `disallowAdd` is set to false, a new record is added; if `disallowAdd` is set to true, the new record is not inserted and a `MatchResult` object is returned with a result code of 11. If

`rejectAssumedMatch` and `disallowUpdate` are set to false, the existing record is updated; if `disallowUpdate` is set to true, the update is not performed and a `MatchResult` object is returned with a result code of 13.

7. If a new record is inserted, all records that were returned from the blocking query are weighed against the new record using the matching algorithm. If a record is updated and the update mode is pessimistic, the same occurs for the updated record. If the matching probability weight of a record is greater than or equal to the potential duplicate threshold, the record is flagged as a potential duplicate (for more information about thresholds, see "Threshold Configuration (Repository)" in *Understanding Sun Master Index Configuration Options (Repository)*).

Master Index Primary Function Processing Logic (Repository)

The primary functions of a master index application can be performed from the Enterprise Data Manager or can be called from the Collaborations in the master index project. Whether potential duplicates are evaluated after a call to any of these functions is dependent on the update mode settings. Potential duplicates are only processed against the single best record (SBR) and not the system records. These functions are all located in the `MasterController` class, and are fully described in the Sun Master Index Javadocs. In the following diagrams, significant fields for potential duplicate processing include fields defined for matching and fields included in the blocking query used for matching. In all of the methods described below, an entry is made in the transaction history table (sbyn_transaction).

The following topics describe the logic for each primary master index function:

activateEnterpriseObject

This method reactivates an enterprise record. The EDM calls this method when you select an EUID and then click Activate EUID=*EUID_number*, (where *EUID_number* is the EUID of the enterprise record to reactivate). Since all potential duplicates were deleted when the EUID was originally deactivated, potential duplicates are always recalculated, regardless of the update mode. Figure 7 illustrates the processing steps.

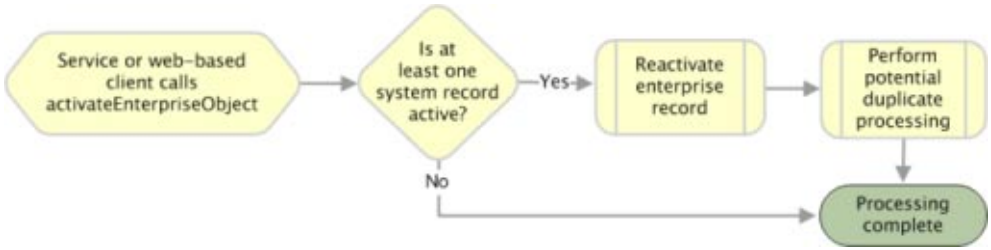


FIGURE 7 activateEnterpriseObject Processing

activateSystemObject

This method reactivates a system record. The EDM calls this method when you select a system from the enterprise record tree and then click Activate *system-ID* (where *system* is the system code and *ID* is the local ID number for the system record to reactivate). If the update mode is set to "pessimistic", the application checks whether any key fields were updated in the SBR. If key fields were updated, potential duplicates are recalculated for the enterprise record. Figure 8 illustrates the processing steps.

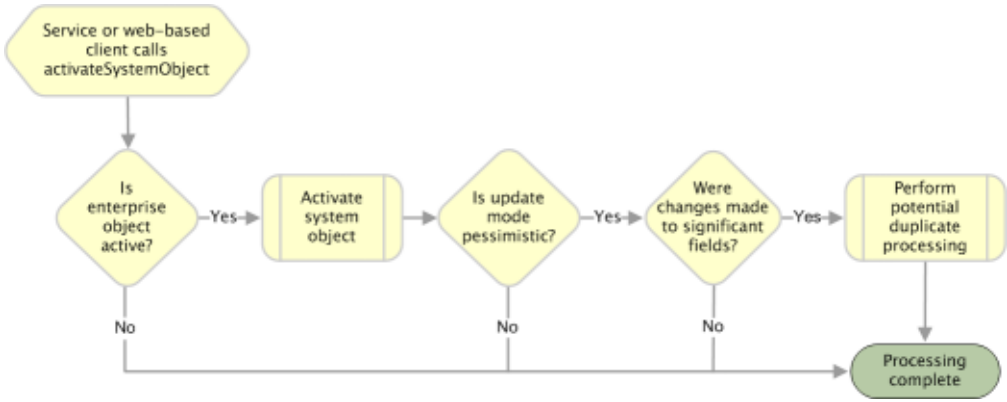


FIGURE 8 activateSystemObject Processing

addSystemObject

This method adds a system record to an enterprise record. The EDM calls this method when you add a system record to an existing enterprise record. If the update mode is set to "pessimistic", the application checks whether any key fields were updated in the SBR. If key fields were updated and the update mode is set to pessimistic, potential duplicates are recalculated for the enterprise record. Figure 9 illustrates the processing steps.

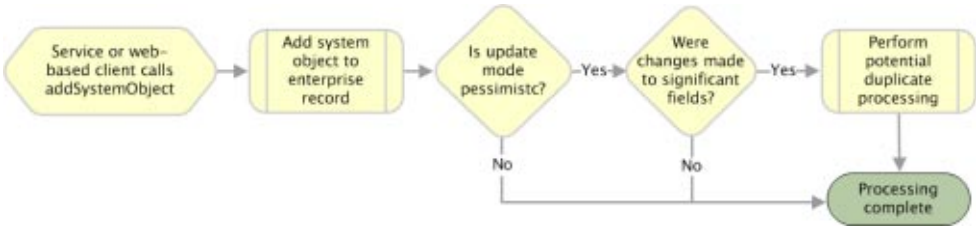


FIGURE 9 addSystemObject Processing

createEnterpriseObject

There are two `createEnterpriseObject` methods, both of which add a new enterprise record to the database and bypass any potential duplicate processing. One method takes only one system record as a parameter and the other takes an array of system records. These methods cannot be called from the EDM and are designed for use in Collaborations.

deactivateEnterpriseObject

This method deactivates an enterprise record specified by its EUID. The EDM calls this method when you select an enterprise record and then click Deactivate EUID=*EUID_number* (where *EUID_number* is the EUID of the enterprise record to deactivate). When an enterprise record is deactivated, all potential duplicate listings for that record are deleted.

deactivateSystemObject

This method deactivates a system record in an enterprise record. The EDM calls this method when you select a system from the enterprise record tree and then click Deactivate *system-ID* (where *system* is the system code and *ID* is the local ID number for the system record to deactivate). If the enterprise record containing this system record has no active system records remaining, the enterprise record is deactivated and all potential duplicate listings are deleted. (Note that if the system record is reactivated, then the enterprise record is recreated.) If the enterprise record has active system records after the transaction and the update mode is set to "pessimistic", the application checks whether any key fields were updated in the SBR. If key fields were updated, potential duplicates are recalculated for the enterprise record. Figure 10 illustrates the processing steps.

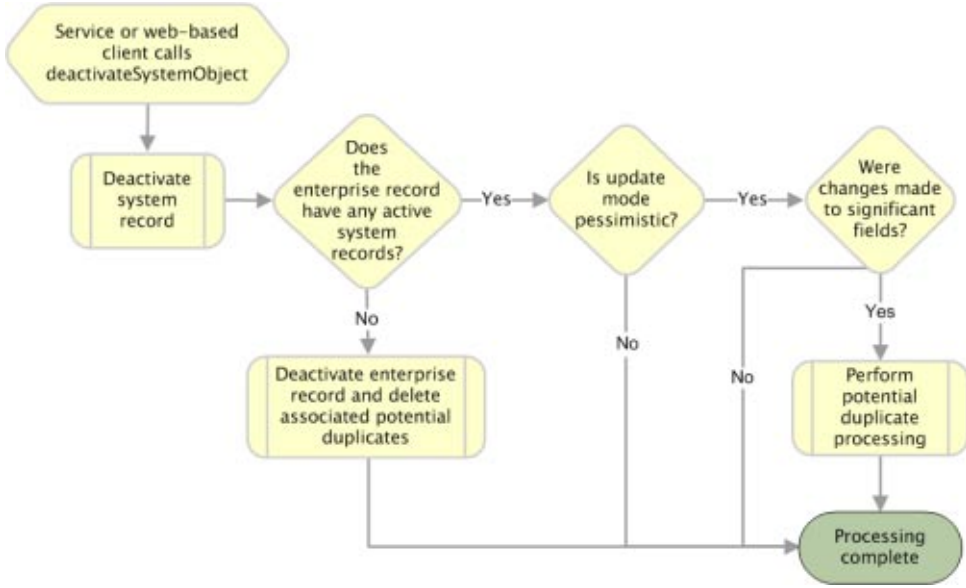


FIGURE 10 deactivateSystemObject Processing

deleteSystemObject

Unlike `deactivateSystemObject`, this method permanently removes a system record from an enterprise record. This method cannot be called from the EDM. If the enterprise record containing the deleted system record has no active system records remaining, the enterprise record is deactivated (even if the enterprise record does have deactivated system records). If the enterprise record has no remaining system records after the system object is deleted, the enterprise record is also deleted. In both cases, any potential duplicate listings for that enterprise record are removed. If the enterprise record has active system records after the transaction and the update mode is set to "pessimistic", the application checks whether any key fields were updated in the SBR. If key fields were updated, potential duplicates are recalculated for the enterprise record. Figure 11 illustrates the processing steps.

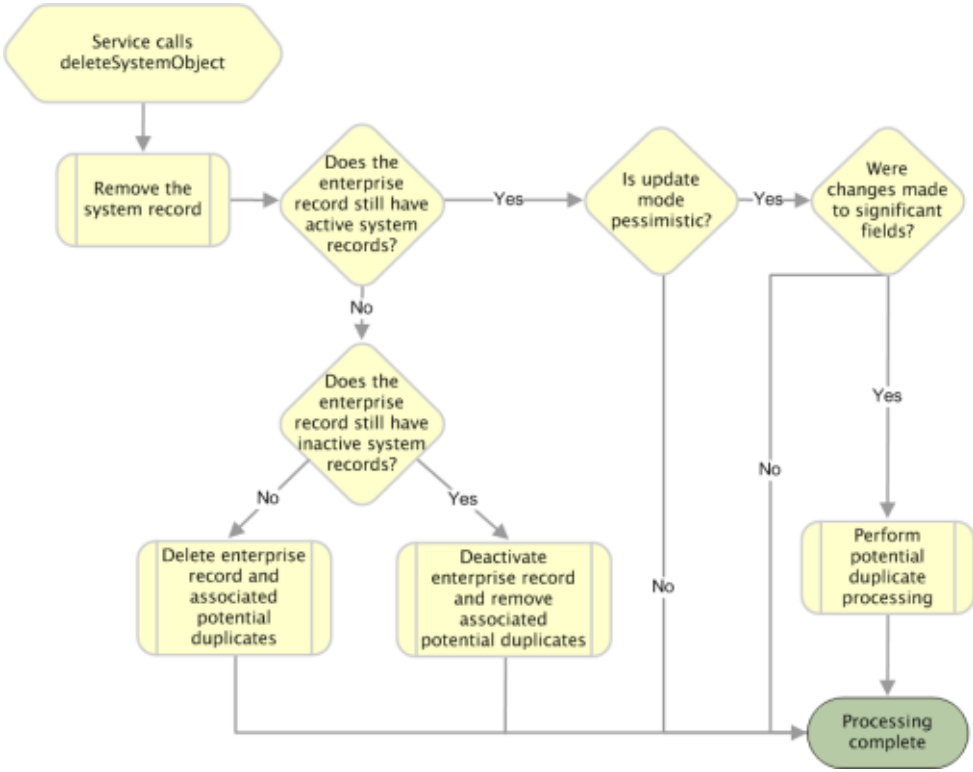


FIGURE 11 deleteSystemObject Processing

mergeEnterpriseObject

There are four `mergeEnterpriseObject` methods that merge two enterprise records (see the Javadocs for more information about each). The EDM calls a merge method twice during a merge transaction. When you first click the EUID Merge arrow, the method is called with the *calculateOnly* parameter set to true in order to display the merge result record for you to view. When you confirm the merge, the EDM calls this method with the *calculateOnly* parameter set to false in order to commit the changes to the database and recalculate potential duplicates if needed. The method called by the EDM checks the SBRs of the records involved in the merge against their corresponding SBRs in the database. If the SBRs differ, the merge is not performed since that means the records were changed by someone else during the merge process.

When this method is called with *calculateOnly* set to false, the application changes the status of the merged enterprise record to merged and deletes all potential duplicate listings for the merged enterprise record. If the update mode is set to pessimistic, the application checks whether any key fields were updated in the SBR of the surviving enterprise record. If key fields

were updated, potential duplicates are recalculated for the enterprise record. Figure 12 illustrates the processing steps, and includes the check for SBR differences, which only occurs in two of the merge methods.

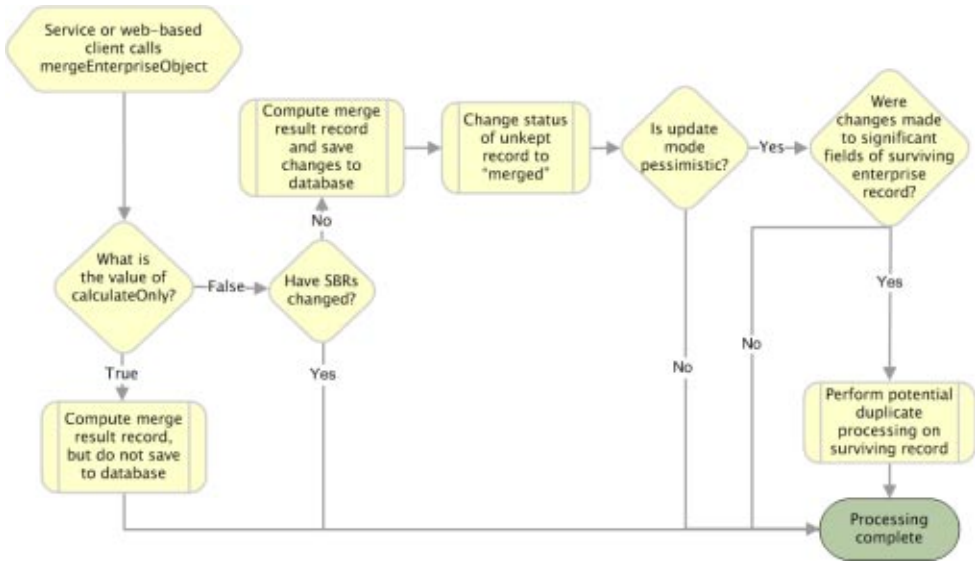


FIGURE 12 mergeEnterpriseObject Processing

mergeSystemObject

There are four methods that merge two system records that are either from the same enterprise record or from two different enterprise records (for more information about each method, see the Javadocs provided with Sun Master Index). The system records must originate from the same external system. The EDM calls this method twice during a system record merge transaction. When you first click the LID Merge arrow, the method is called with the *calculateOnly* parameter set to true in order to display the merge result record for you to view. When you confirm the merge, the EDM calls this method with the *calculateOnly* parameter set to false in order to commit the changes to the database and recalculate potential duplicates if needed. Two of the merge methods compare the SBRs of the records with their corresponding SBRs in the database to ensure that no updates were made to the records before finalizing the merge.

When this method is called with *calculateOnly* set to "false", the application changes the status of the merged system record to "merged". If the system records were merged within the same enterprise record and the update mode is set to "pessimistic", the application checks whether any key fields were updated in the SBR. If key fields were updated, potential duplicates are recalculated for the enterprise record.

If the system records originated from two different enterprise records and the enterprise record that contained the unkept the system record no longer has any active system records but does contain inactive system records, that enterprise record is deactivated and all associated potential duplicate listings are deleted. (Note that if the system records are unmerged, the enterprise record is reactivated.) If the enterprise record that contained the unkept system record no longer has any system records, that enterprise record is deleted along with any potential duplicate listings.

If both enterprise records are still active and the update mode is set to pessimistic, the application checks whether any key fields were updated in the SBR for each enterprise record. If key fields were updated, potential duplicates are recalculated for each enterprise record. Figure 13 illustrates the processing steps, and includes the check for SBR differences, which only occurs in two of the merge methods.

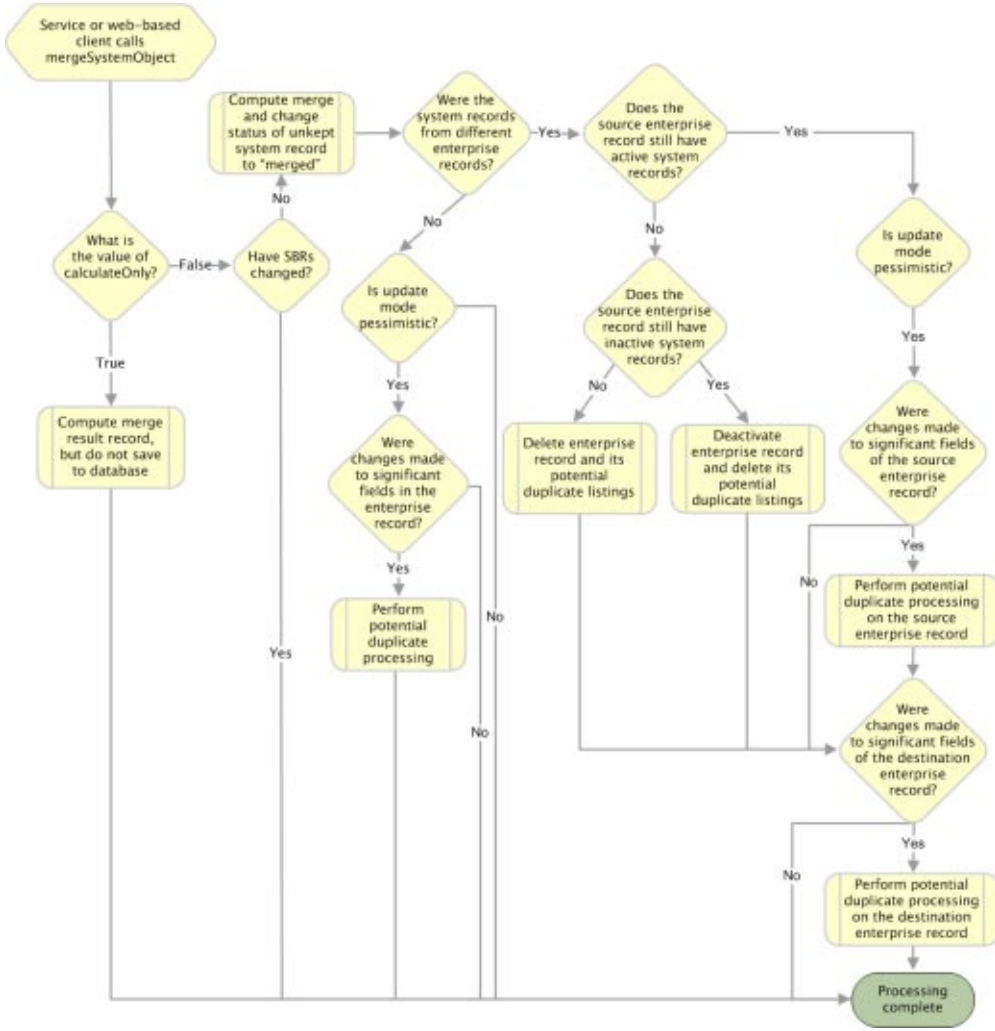


FIGURE 13 mergeSystemObject Processing

transferSystemObject

This method transfers a system record from one enterprise record to another. This method is not called from the EDM. If the enterprise record from which the system record was transferred no longer has any active system records (but still contains deactivated system records), that enterprise record is deactivated and any associated potential duplicate listings are removed. If the enterprise record from which the system record was transferred no longer has any system records, that enterprise record is deleted along with all associated potential duplicate listings. If both enterprise records are still active and the update mode is set to pessimistic, the application

checks whether any key fields were updated in the SBR for each enterprise record. If key fields were updated, potential duplicates are recalculated for each enterprise record. Figure 14 illustrates the processing steps.

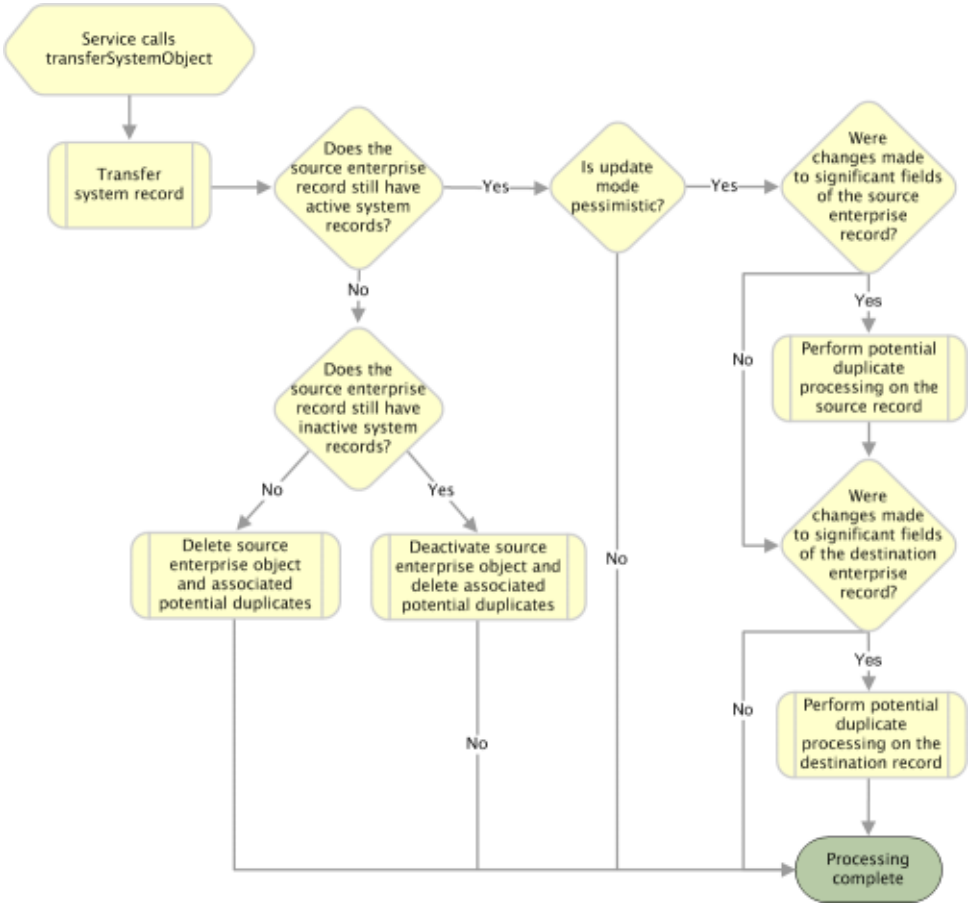


FIGURE 14 transferSystemObject Processing

undoAssumedMatch

This method reverses an assumed match made by the master index application, using the information from the system record that created the assumed match to create a new enterprise record. The EDM calls this method when you confirm the transaction after selecting Undo Assumed Match. Potential duplicates are calculated for the new record regardless of the update mode. If the update mode is set to pessimistic, the application checks whether any key fields

were updated in the SBR of the original enterprise record. If key fields were updated, potential duplicates are recalculated for the enterprise record. Figure 15 illustrates the processing steps.

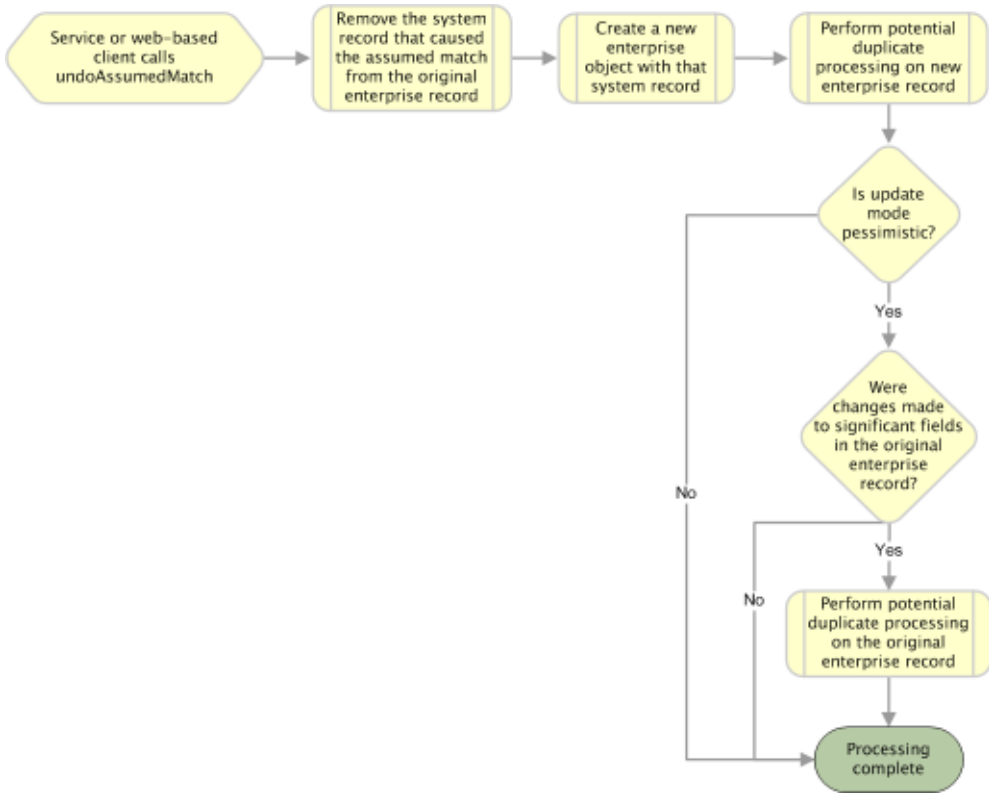


FIGURE 15 undoAssumedMatch Processing

unmergeEnterpriseObject

There are two methods that unmerge two enterprise records that were previously merged. One method unmerges the record without checking to make sure the SBR of the active record was not changed by another process before finalizing the merge and one method performs the SBR check (see the Javadocs provided with Sun Master Index for more information). The EDM calls this method twice during an unmerge transaction. When you first click Unmerge, the method is called with the *calculateOnly* parameter set to true in order to display the unmerge result records for you to view. When you confirm the unmerge, the EDM calls this method with the *calculateOnly* parameter set to false in order to commit the changes to the database and recalculate potential duplicates.

When this method is called with *calculateOnly* set to false, the application changes the status of the merged enterprise record back to active and recalculates potential duplicate listings for the record. If the update mode is set to "pessimistic", the application checks whether any key fields were updated in the SBR of the enterprise record that was still active after the merge. If key fields were updated, potential duplicates are recalculated for that enterprise record. Figure 16 illustrates the processing steps and includes the check for SBR updates.

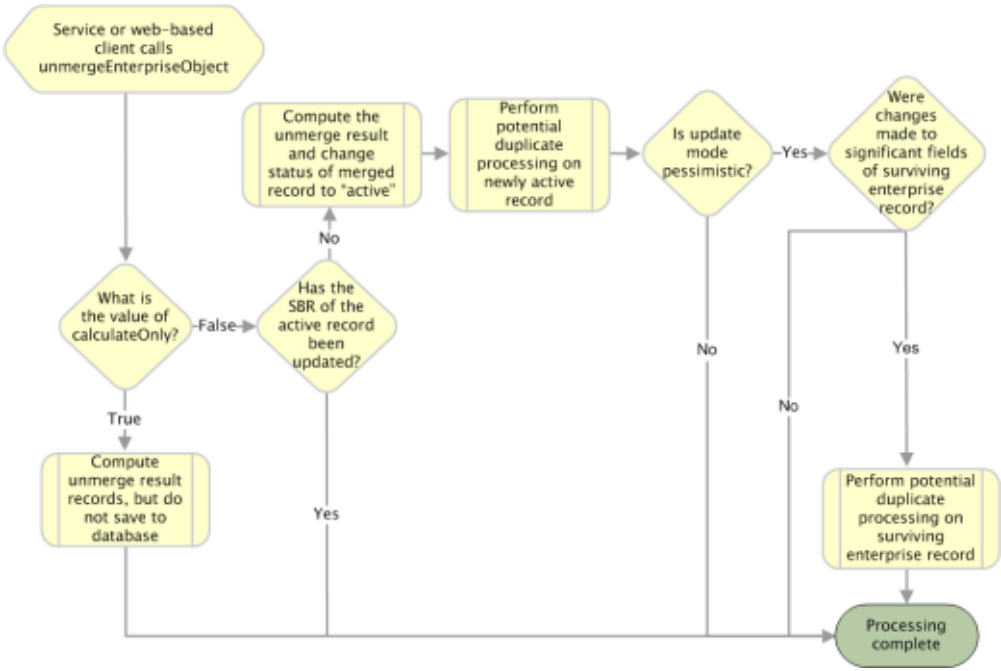


FIGURE 16 unmergeEnterpriseObject Processing

unmergeSystemObject

There are two methods that unmerge two system records that had previously been merged. One method unmerges the record without checking to make sure the SBR of the active record was not changed by another process before finalizing the merge and one method performs the SBR check (see the Javadocs provided with Sun Master Index for more information). The EDM calls this method twice during a system record unmerge transaction. When you first click Unmerge, the method is called with the *calculateOnly* parameter set to true in order to display the unmerge result record for you to view. When you confirm the unmerge, the EDM calls this method with the *calculateOnly* parameter set to false in order to commit the changes to the database and recalculate potential duplicates if needed.

When this method is called with *calculateOnly* set to false, the application changes the status of the merged system record back to active. If the source enterprise record (the record that contained the merge result system record after the merge) has more than one active system record after the unmerge and the update mode is set to pessimistic, the application checks whether any key fields were updated in that record. If key fields were updated, potential duplicates are recalculated for the source enterprise record.

If the source enterprise record has only one active system, potential duplicate processing is performed regardless of the update mode and of whether there were any changes to key fields. If the update mode is set to pessimistic, the application checks whether any key fields were updated in the SBR for destination enterprise record. If key fields were updated, potential duplicates are recalculated for each enterprise record. Figure 17 illustrates the processing steps, assuming the system record unmerge involves two enterprise records and including the check for SBR updates.

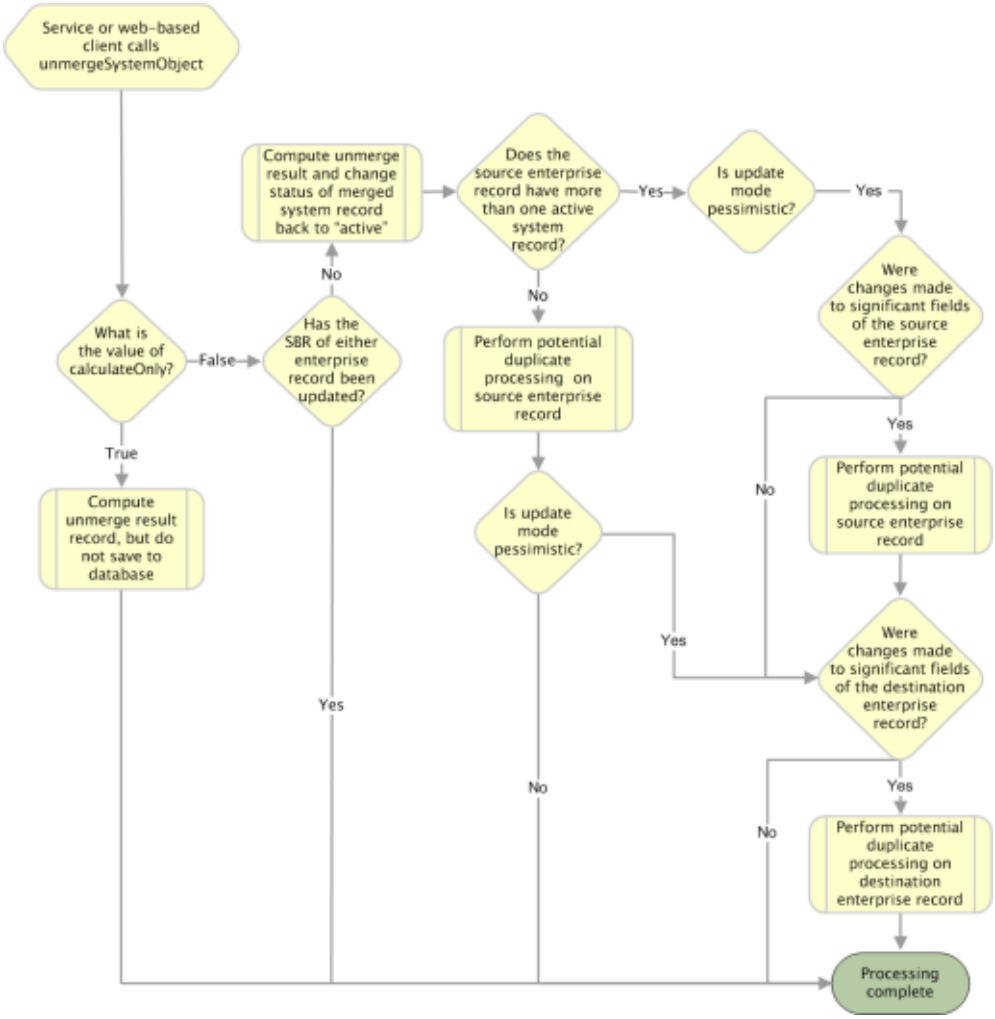


FIGURE 17 unmergeSystemObject Processing

updateEnterpriseDupRecalc

This method updates the database to reflect new values for an enterprise record. It processes records in the same manner as `updateEnterpriseObject`, but provides an override flag for the update mode that allows you to defer potential duplicate processing. The EDM does not call this method. If the enterprise record is deactivated during the update, potential duplicates are deleted for that record. If the enterprise record was changed during the transaction but is still active and the *performPessimistic* parameter is set to true, the application checks whether any key fields were updated in the SBR of the enterprise record. If key fields were updated, potential duplicates are recalculated. Figure 18 illustrates the processing steps.

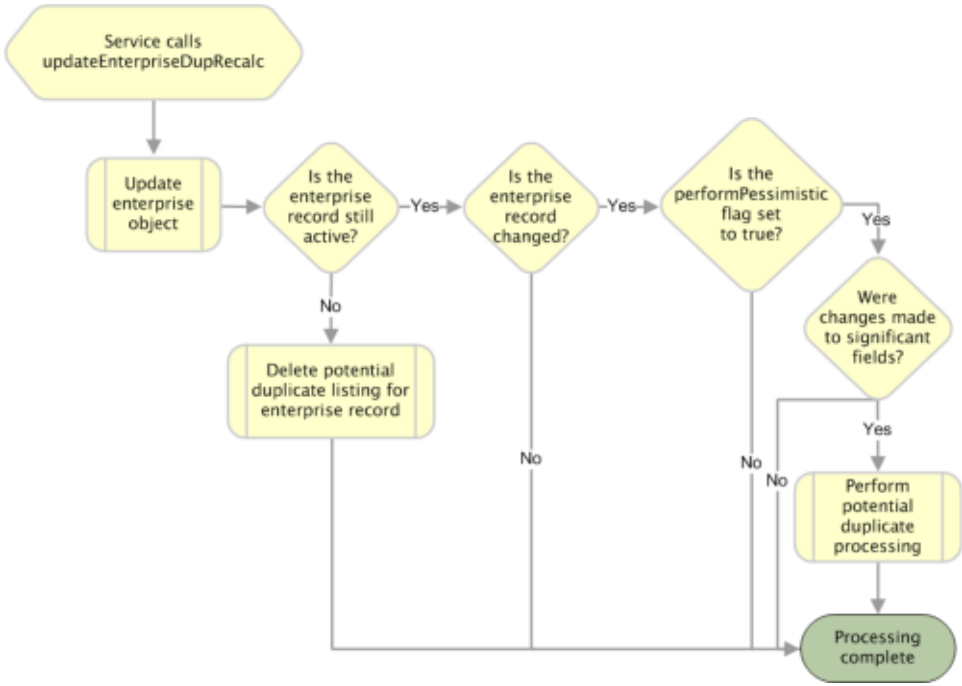


FIGURE 18 updateEnterpriseDupRecalc Processing

updateEnterpriseObject

This method updates the database to reflect new values for an enterprise record, and is called from the EDM when you commit changes to an existing record. If the enterprise record is deactivated during the update, potential duplicates are deleted for that record. If the enterprise record is still active, was changed during the transaction, and the update mode is set to pessimistic, the application checks whether any key fields were updated in the SBR of the enterprise record. If key fields were updated, potential duplicates are recalculated. Figure 19 illustrates the processing steps.

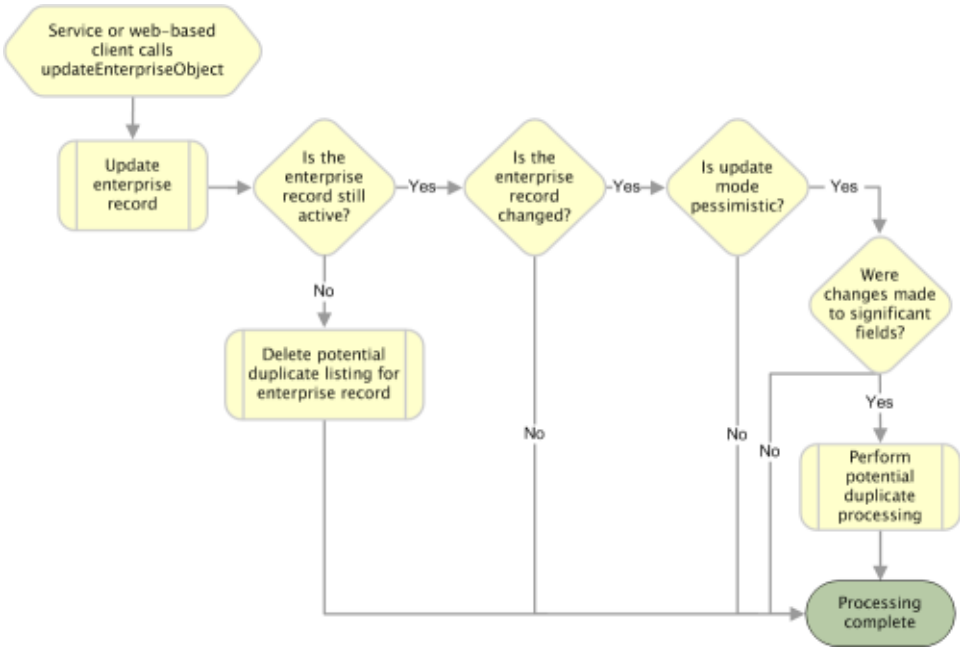


FIGURE 19 updateEnterpriseObject Processing

updateSystemObject

There are two methods that update the database to reflect new values for a system record. One method updates the record without checking that there were no concurrent changes to the record, and the other method compares the SBR of the associated enterprise object in the transaction with that in the database to be sure there were no concurrent changes (see the Javadocs for more information). The EDM calls the method that checks for SBR changes when you commit changes to an existing system record.

If the enterprise record is deactivated during the update, potential duplicates are deleted for that record. If the enterprise record was changed during the transaction, the enterprise record is still active, and the update mode is set to pessimistic, the application checks whether any key fields were updated in the SBR of the enterprise record. If key fields were updated, potential duplicates are recalculated. Figure 20 illustrates the processing steps and includes the check for SBR changes though it only occurs with one of the methods.

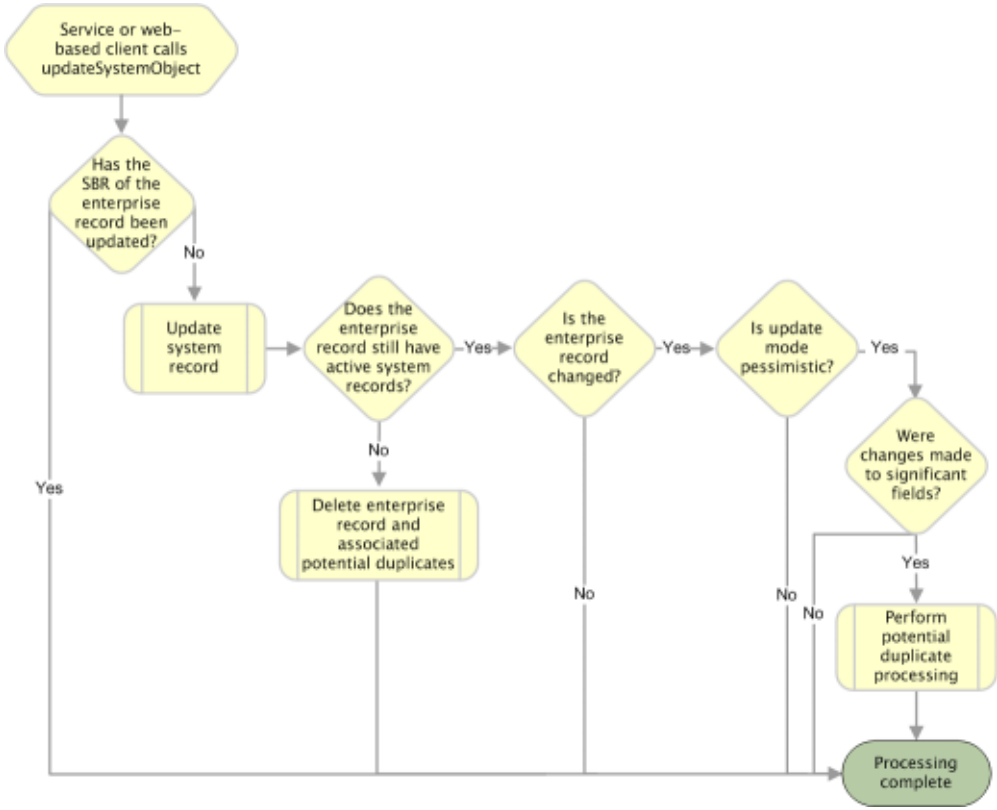


FIGURE 20 updateSystemObject Processing

The Master Index Database Structure (Repository)

The following topics provide information about the master index database, including descriptions of each table and a sample entity relationship diagram. All information in these topics pertains to the default version of the database. Your implementation might vary depending on the customization made to the Object Definition and to the scripts used to create the master index database.

About the Master Index Database (Repository)

The master index database stores information about the entities being indexed, such as people or businesses. The database stores records from local systems in their original form and also stores a record for each object that is considered to be the single best record (SBR).

The structure of the database tables that store object information is dependent on the information specified in the Object Definition file created by the wizard. Sun Master Index generates a script to create the tables and fields in the database based on the information in the Object Definition file. If you update the Object Definition file, regenerating the application updates the database scripts accordingly. This allows you to define the database as you define the object structure.

While most of the structures created in the database are based on information in the Object Definition file, some of the tables, such as sbyn_seq_table and sbyn_common_detail, are standard for all implementations. The database includes tables that store information about the objects defined for the master index application as well as tables that store common maintenance information, transactional information, and external system information. The database includes the tables listed in Table 2.

TABLE 2 Master Index Database Tables

Table Name	Description
SBYN_*OBJECT_NAME*	Stores information for the parent objects associated with local system records. This database table is named by the parent object name. For example, a table storing company objects is named sbyn_company; a table storing person objects is named sbyn_person. Only one table stores parent object information for system records.
SBYN_*OBJECT_NAME*SBR	Stores information for the parent objects associated with single best records. This database table is named by the parent object name followed by "SBR". For example, a table storing company objects is named sbyn_companysbr; a table storing person objects is named sbyn_personsbr. Only one table stores parent object information for SBRs.
SBYN_*CHILD_OBJECT*	Stores information for child objects associated with local system records. These database tables are named by their object name. For example, a table storing address objects is named sbyn_address; a table storing comment objects is named sbyn_comment. One database table is created for each child object defined in the object structure.

TABLE 2 Master Index Database Tables *(Continued)*

Table Name	Description
SBYN_*CHILD_OBJECT*SBR	Stores information for child objects associated with a single best record. These database tables are named by their object name followed by "SBR". For example, a table storing address objects is named sbyn_addresssbr; a table storing comment objects is named sbyn_commentsbr. One SBR database table is created for each child object defined in the object structure.
SBYN_APPL	Lists the applications with which each item in stc_common_header is associated.
SBYN_ASSUMEDMATCH	Stores information about records that were automatically matched by the master index application.
SBYN_AUDIT	Stores audit information about each time object information is accessed from the EDM. **Note –** If audit logging is enabled, this table can grow very large and might require periodic archiving.
SBYN_COMMON_DETAIL	Contains all of the processing codes associated with the items listed in sbyn_common_header.
SBYN_COMMON_HEADER	Contains a list of the different types of processing codes used by the master index application. These types are also associated with the drop-down lists you can specify for the EDM.
SBYN_ENTERPRISE	Stores the local ID and system pairs, along with their associated EUID.
SBYN_MERGE	Stores information about all merge and unmerge transactions processed from either external systems or the EDM.
SBYN_OVERWRITE	Stores information about fields that are locked for updates in an SBR.
SBYN_POTENTIALDUPLICATES	Stores a list of potential duplicate records and flags potential duplicate pairs that have been resolved.
SBYN_SEQ_TABLE	Stores the sequential codes that are used in other tables in the database, such as EUIDs, transaction numbers, and so on.
SBYN_SYSTEMOBJECT	Stores information about the system objects in the database, including the local ID and system, create date and user, status, and so on.
SBYN_SYSTEMS	Stores a list of systems in your organization, along with defining information.
SBYN_SYSTEMSBR	Stores transaction information about an SBR, such as the create or update date, status, and so on.
SBYN_TRANSACTION	Stores a history of changes to each record stored in the database.

TABLE 2 Master Index Database Tables *(Continued)*

Table Name	Description
SBYN_USER_CODE	Like the sbyn_common_detail table, this table stores processing codes and drop-down list values. This table contains additional validation information that allows you to validate information in a dependent field (for example, to validate cities against the entered postal code).

Master Index Database Table Details (Repository)

The tables in the following topics describe each column in the default database tables.

The columns are identical for Oracle and SQL Server databases, but the data types differ in some cases. Table 3 lists the data type differences, and the differences are noted in the Data Type column for each table in this section.

TABLE 3 Oracle and SQL Server Data Type Differences

Oracle Data Type	SQL Server Data Type
BLOB	Varbinary(MAX)
DATE	DateTime
INTEGER	Int

TABLE 3 Oracle and SQL Server Data Type Differences *(Continued)*

Oracle Data Type	SQL Server Data Type
LONG	Varchar(MAX)
NUMBER	Numeric
TIMESTAMP	DateTime
VARCHAR2	Varchar

SBYN_*OBJECT_NAME*

This table stores the parent object in each system record received by the master index application. It is linked to the tables that store each child object in the system record by the *object_name*id column (where *object_name* is the name of the parent object). This table contains the columns listed below regardless of the design of the object structure, and also contains a column for each field you defined for the parent object in the Object Definition file. Columns to store standardized or phonetic versions of certain fields are automatically added when you specify certain match types in the wizard.

The differences in data types between Oracle and SQL Server are noted in Table 4. The Oracle type is on the first line, and the SQL Server type is on the second.

TABLE 4 SBYN_*OBJECT_NAME* Table Description

Column Name	Data Type	Column Description
SYSTEMCODE	VARCHAR2(20) Varchar(20)	The system code for the system record.
LID	VARCHAR2(25) Varchar(25)	A local identification code assigned by the specified system.
*OBJECT_NAME*ID	VARCHAR2(20) Varchar(20)	A unique ID for the parent object in a system record. This is named according to the parent object. For example, if the parent object is “Company”, the name of this column is “companyid”; if the parent object is “Person”, the name of this column is “personid”.
FIELD_NAME	Varies	The name of each field in the parent object. A database column is created for each field, and the data type depends on the type specified in the Object Definition file.

SBYN_*OBJECT_NAME*SBR

This table stores the parent object of the SBR for each enterprise object in the master index database. It is linked to the tables that store each child object in the SBR by the *object_name*id column (where *object_name* is the name of the parent object). This table contains the columns

listed below regardless of the design of the object structure, and also contains a column for each field defined for the parent object in the Object Definition file. In addition, columns to store standardized or phonetic versions of certain fields are automatically added when you specify certain match types in the wizard.

The differences in data types between Oracle and SQL Server are noted in Table 5. The Oracle type is on the first line, and the SQL Server type is on the second.

TABLE 5 SBYN_*OBJECT_NAME*SBR Table Description

Column Name	Data Type	Column Description
EUID	VARCHAR2(20) Varchar(20)	The enterprise unique identifier assigned by the master index application.
*OBJECT_NAME*ID	VARCHAR2(20) Varchar(20)	A unique ID for the parent object in a system record. This is named according to the parent object. For example, if the parent object is "Company", the name of this column is "companyid"; if the parent object is "Person", the name of this column is "personid".
FIELD_NAME	Varies	The name of each field in the parent object. A database column is created for each field, and the data type depends on the type specified in the Object Definition file.

SBYN_*CHILD_OBJECT*

The sbyn_*child_object* tables (where *child_object* is the name of a child object in the object structure) store information about the child objects associated with a system record in the master index application. All tables storing child object information for system records contain the columns listed below. The remaining columns are defined by the fields you specify for each child object in the object structure, including any standardized or phonetic fields.

The differences in data types between Oracle and SQL Server are noted in Table 6. The Oracle type is on the first line, and the SQL Server type is on the second.

TABLE 6 SBYN_*CHILD_OBJECT* and SBYN_*CHILD_OBJECT*SBR Table Description

Column Name	Data Type	Column Description
*OBJECT_NAME*ID	VARCHAR2(20) Varchar(20)	The unique ID for the parent object associated with the child object in the system record.
*CHILD_OBJECT*ID	VARCHAR2(20) Varchar(20)	The unique ID for each record in the table. This column cannot be null.

TABLE 6 SBYN_*CHILD_OBJECT* and SBYN_*CHILD_OBJECT*SBR Table Description *(Continued)*

Column Name	Data Type	Column Description
FIELD_NAME	Varies	The name of each field in the child object. A database column is created for each field, and the data type depends on the type specified in the Object Definition file.

SBYN_*CHILD_OBJECT*SBR

The sbyn_*child_object*sbr tables (where *child_object* is the name of a child object in the object structure) store information about the child objects associated with an SBR in the master index application. All tables storing child object information for SBRs contain the columns listed below. The remaining columns are defined by the fields you specify for each child object in the Object Definition file, including any standardized or phonetic fields.

The differences in data types between Oracle and SQL Server are noted in Table 7. The Oracle type is on the first line, and the SQL Server type is on the second.

TABLE 7 SBYN_*CHILD_OBJECT* and SBYN_*CHILD_OBJECT*SBR Table Description

Column Name	Data Type	Column Description
*OBJECT_NAME*ID	VARCHAR2(20) Varchar(20)	The unique ID for the parent object associated with the child object in the SBR.
*CHILD_OBJECT*ID	VARCHAR2(20) Varchar(20)	The unique ID for each record in the table. This column cannot be null.
FIELD_NAME	Varies	The name of each field in the child object. A database column is created for each field, and the data type depends on the type specified in the Object Definition file.

SBYN_APPL

This table stores information about the applications used in the master index system. The differences in data types between Oracle and SQL Server are noted in Table 8. The Oracle type is on the first line, and the SQL Server type is on the second.

TABLE 8 SBYN_APPL Table Description

Column Name	Data Type	Description
APPL_ID	NUMBER(10) Numeric(10, 0)	The unique sequence number code for the listed application.

TABLE 8 SBYN_APPL Table Description *(Continued)*

Column Name	Data Type	Description
CODE	VARCHAR2(8) Varchar(8)	A unique code for the application.
DESCR	VARCHAR2(30) Varchar(30)	A brief description of the application.
READ_ONLY	CHAR(1)	An indicator of whether the current entry can be modified. If the value of this column is "Y", the entry cannot be modified.
CREATE_DATE	DATE datetime	The date the application entry was created.
CREATE_USERID	VARCHAR2(20) Varchar(20)	The logon ID of the user who created the application entry.

SBYN_ASSUMEDMATCH

This table maintains a record of each assumed match transaction that occurs in the master index application, allowing you to review these transactions and, if necessary, reverse an assumed match. This table can grow quite large over time and might require periodic archiving. The differences in data types between Oracle and SQL Server are noted in Table 9. The Oracle type is on the first line, and the SQL Server type is on the second.

TABLE 9 SBYN_ASSUMEDMATCH Table Description

Column Name	Data Type	Description
ASSUMEDMATCHID	VARCHAR2(20) Varchar(20)	The unique ID for the assumed match transaction.
EUID	VARCHAR2(20) Varchar(20)	The EUID into which the incoming record was merged.
SYSTEMCODE	VARCHAR2(20) Varchar(20)	The system code for the source system (that is, the system from which the incoming record originated).
LID	VARCHAR2(25) Varchar(25)	The local ID of the record in the source system.
WEIGHT	VARCHAR2(20) Varchar(20)	The matching weight between the incoming record and the EUID record into which it was merged.

TABLE 9 SBYN_ASSUMEDMATCH Table Description *(Continued)*

Column Name	Data Type	Description
TRANSACTION NUMBER	VARCHAR2(20) Varchar(20)	The transaction number associated with the assumed match.

SBYN_AUDIT

This table maintains a log of each instance in which any of the master index database tables are accessed through the EDM. This includes each time a record appears on a search results page, a comparison page, the View/Edit page, and so on. This log is only maintained if the EDM is configured for it. This table can grow very large over time and might require periodic archiving. The differences in data types between Oracle and SQL Server are noted in Table 10. The Oracle type is on the first line, and the SQL Server type is on the second.

TABLE 10 SBYN_AUDIT Table Description

Column Name	Data Type	Description
AUDIT_ID	VARCHAR2(20) Varchar(20)	The unique identification code for the audit record. This column cannot be null.
PRIMARY_OBJECT_TYPE	VARCHAR2(20) Varchar(20)	The name of the parent object as defined in the Object Definition file.
EUID	VARCHAR2(15) Varchar(15)	The EUID whose information was accessed during an EDM transaction.
EUID_AUX	VARCHAR2(15) Varchar(15)	The second EUID whose information was accessed during an EDM transaction. A second EUID appears when viewing information about merge and unmerge transactions, comparisons, and so on.
FUNCTION (Oracle) OPERATION (SQL Server)	VARCHAR2(32) Varchar(32)	The type of transaction that caused the audit record to be written. This column cannot be null.
DETAIL	VARCHAR2(120) Varchar(120)	A brief description of the transaction that caused the audit record to be written.
CREATE_DATE	DATE datetime	The date the transaction that created the audit record was performed. This column cannot be null.
CREATE_BY	VARCHAR2(20) Varchar(20)	The user ID of the person who performed the transaction that caused the audit log. This column cannot be null.

SBYN_COMMON_DETAIL

This table stores the processing codes and descriptions for all of the common maintenance data elements. This is the detail table for sbyn_common_header. Each data element in sbyn_common_detail is associated with a data type in sbyn_common_header by the common_header_id column. None of the columns in this table can be null. The differences in data types between Oracle and SQL Server are noted in Table 11. The Oracle type is on the first line, and the SQL Server type is on the second.

TABLE 11 SBYN_COMMON_DETAIL Table Description

Column Name	Data Type	Description
COMMON_DETAIL_ID	NUMBER(10) numeric(10, 0)	The unique identification code of the common table data element.
COMMON_HEADER_ID	NUMBER(10) numeric(10, 0)	The unique identification code of the common table data type associated with the data element (as stored in the common_header_id column of the sbyn_common_header table).
CODE	VARCHAR2(20) Varchar(20)	The processing code for the common table data element.
DESCR	VARCHAR2(50) Varchar(50)	A description of the common table data element.
READ_ONLY	CHAR(1)	An indicator of whether the common table data element can be modified.
CREATE_DATE	DATE datetime	The date the data element record was created.
CREATE_USERID	VARCHAR2(20) Varchar(20)	The user ID of the person who created the data element record.

SBYN_COMMON_HEADER

This table stores a description of each type of common maintenance data and is the header table for sbyn_common_detail. Together, these tables store the processing codes and drop-down menu descriptions for each common table data type. For a person index, common table data types might include Religion, Language, Marital Status, and so on. For a business index, common table data types might include Address Type, Phone Type, and so on. None of the columns in this table can be null.

The differences in data types between Oracle and SQL Server are noted in Table 12. The Oracle type is on the first line, and the SQL Server type is on the second.

TABLE 12 SBYN_COMMON_HEADER Table Description

Column Name	Data Type	Description
COMMON_HEADER_ID	VARCHAR2(10) Varchar(10)	The unique identification code of the common table data type.
APPL_ID	VARCHAR2(10) Varchar(10)	The application ID from sbyn_appl that corresponds to the application for which the common table data type is used.
CODE	VARCHAR2(8) Varchar(8)	A unique processing code for the common table data type.
DESCR	VARCHAR2(50) Varchar(50)	A description of the common table data type.
READ_ONLY	CHAR(1)	An indicator of whether an entry in the table is read-only (if this column is set to "Y", the entry is read-only).
MAX_INPUT_LEN	NUMBER(10) numeric(10, 0)	The maximum number of characters allowed in the code column for the common table data type.
TYP_TABLE_CODE	VARCHAR2(3) Varchar(3)	This column is not currently used.
CREATE_DATE	DATE datetime	The date the common table data type record was created.
CREATE_USERID	VARCHAR2(20) Varchar(20)	The user ID of the person who created the common table data type record.

SBYN_ENTERPRISE

This table stores a list of all the system and local ID pairs assigned to the enterprise records in the database, along with the associated EUID for each pair. This table is linked to sbyn_systemobject by the systemcode and lid columns, and is linked to sbyn_systemsbr by the euid column. This table maintains links between the SBR and its associated system objects. None of the columns in this table can be null.

The differences in data types between Oracle and SQL Server are noted in Table 13. The Oracle type is on the first line, and the SQL Server type is on the second.

TABLE 13 SBYN_ENTERPRISE Table Description

Column Name	Data Type	Description
SYSTEMCODE	VARCHAR2(20) Varchar(20)	The processing code of the system associated with the local ID.
LID	VARCHAR2(25) Varchar(25)	The local ID associated with the system and EUID.
EUID	VARCHAR2(20) Varchar(20)	The EUID associated with the local ID and system.

SBYN_MERGE

This table maintains a record of each merge transaction that occurs in the master index application, both through the EDM and from external systems. It also records any unmerges that occur. The differences in data types between Oracle and SQL Server are noted in Table 14. The Oracle type is on the first line, and the SQL Server type is on the second.

TABLE 14 SBYN_MERGE Table Description

Column Name	Data Type	Description
MERGE_ID	VARCHAR2(20) Varchar(20)	The unique, sequential identification code of merge record. This column cannot be null.
KEPT_EUID	VARCHAR2(20) Varchar(20)	The EUID of the record that was retained after the merge transaction. This column cannot be null.
MERGED_EUID	VARCHAR2(20) Varchar(20)	The EUID of the record that was not retained after the merge transaction.
MERGE_TRANSACTIONNUM	VARCHAR2(20) Varchar(20)	The transaction number associated with the merge transaction. This column cannot be null.
UNMERGE_TRANSACTIONNUM	VARCHAR2(20) Varchar(20)	The transaction number associated with the unmerge transaction.

SBYN_OVERWRITE

This table stores information about the fields that are locked for updates in the SBRs. It stores the EUID of the SBR, the ePath to the field, and the current locked value of the field. The differences in data types between Oracle and SQL Server are noted in Table 15. The Oracle type is on the first line, and the SQL Server type is on the second.

TABLE 15 SBYN_OVERWRITE Table Description

Column Name	Data Type	Description
EUID	VARCHAR2(20) Varchar(20)	The EUID of an SBR containing fields for which the overwrite lock is set.
PATH	VARCHAR2(200) Varchar(20)	The ePath to a field that is locked in an SBR from the EDM.
TYPE	VARCHAR2(20) Varchar(20)	The data type of a field that is locked in an SBR.
INTEGERDATA	NUMBER(38) numeric(38, 0)	The data that is locked for overwrite in an integer field.
BOOLEANDATA	NUMBER(38) numeric(38, 0)	The data that is locked for overwrite in a boolean field.
STRINGDATA	VARCHAR2(200) Varchar(200)	The data that is locked for overwrite in a string field.
BYTEDATA	CHAR(2)	The data that is locked for overwrite in a byte field.
LONGDATA	LONG varchar(MAX)	The data that is locked for overwrite in a long integer field.
DATEDATA	DATE datetime	The data that is locked for overwrite in a date field.
FLOATDATA	NUMBER(38,4) numeric(38, 4)	The data that is locked for overwrite in a floating decimal field.
TIMESTAMPDATA	DATE datetime	The data that is locked for overwrite in a timestamp field.

SBYN_POTENTIALDUPLICATES

This table maintains a list of all records that are potential duplicates of one another. It also maintains a record of whether a potential duplicate pair has been resolved or permanently resolved. The differences in data types between Oracle and SQL Server are noted in Table 16. The Oracle type is on the first line, and the SQL Server type is on the second.

TABLE 16 SBYN_POTENTIALDUPLICATES Table Description

Column Name	Data Type	Description
POTENTIALDUPLICATEID	VARCHAR2(20) Varchar(20)	The unique identification number of the potential duplicate transaction.
WEIGHT	VARCHAR2(20) Varchar(20)	The matching weight of the potential duplicate pair.
TYPE	VARCHAR2(15) Varchar(15)	This column is reserved for future use.
DESCRIPTION	VARCHAR2(120) Varchar(120)	A description of what caused the potential duplicate flag.
STATUS	VARCHAR2(15) Varchar(15)	The status of the potential duplicate pair. The possible values are: ■ **U** – Unresolved ■ **R** – Resolved ■ **A** – Resolved permanently
HIGHMATCHFLAG	VARCHAR2(15) Varchar(15)	This column is reserved for future use.
RESOLVEDUSER	VARCHAR2(30) Varchar(30)	The user ID of the person who resolved the potential duplicate status.
RESOLVEDDATE	DATE datetime	The date the potential duplicate status was resolved.
RESOLVEDCOMMENT	VARCHAR2(120) Varchar(120)	Comments regarding the resolution of the duplicate status. This is not currently used.
EUID2	VARCHAR2(20) Varchar(20)	The EUID of the second record in the potential duplicate pair.
TRANSACTIONNUMBER	VARCHAR2(20) Varchar(20)	The transaction number associated with the transaction that produced the potential duplicate flag.
EUID1	VARCHAR2(20) Varchar(20)	The EUID of the first record in the potential duplicate pair.

SBYN_SEQ_TABLE

This table controls and maintains a record of the sequential identification numbers used in various tables in the database, ensuring that each number is unique and assigned in order. Several of the ID numbers maintained in this table are determined by the object structure. The numbers are assigned sequentially, but are cached in chunks of 1000 numbers for optimization (so the application does not need to query the sbyn_seq_table table for each transaction). The chunk size for the EUID sequence is configurable. If the server is reset before all allocated numbers are used, the unused numbers are discarded and never used, and numbering is restarted at the beginning of the next 1000-number chunk.

The differences in data types between Oracle and SQL Server are noted in Table 17. The Oracle type is on the first line, and the SQL Server type is on the second.

TABLE 17 SBYN_SEQ_TABLE Table Description

Column Name	Data Type	Description
SEQ_NAME	VARCHAR2(20) Varchar(20)	The name of the object for which the sequential ID is stored.
SEQ_COUNT	NUMBER(38) numeric(38, 0)	The current value of the sequence. The next record will be assigned the current value plus one.

The default sequence numbers are listed in Table 18.

TABLE 18 Default Sequence Numbers

Sequence Name	Description
EUID	The sequence number that determines how EUIDs are assigned to new records. The chunk size for the EUID sequence number is configurable in the Threshold file.
POTENTIALDUPLICATE	The sequence number assigned each potential duplicate transaction record in sbyn_potentialduplicates (column name “potentialduplicateid”).
TRANSACTIONNUMBER	The sequence number assigned to each transaction in the master index application. This number is stored in sbyn_transaction (column name “transactionnumber”).
ASSUMEDMATCH	The sequence number assigned to each assumed match transaction record in sbyn_assumedmatch (column name “assumedmatchid”).
AUDIT	The sequence number assigned to each audit log record in sbyn_audit (column name “audit_id”).

TABLE 18 Default Sequence Numbers *(Continued)*

Sequence Name	Description
MERGE	The sequence number assigned to each merge transaction in sbyn_merge (column name "merge_id").
SBYN_APPL	The sequence number assigned to each application listed in sbyn_appl (column name "appl_id")
SBYN_COMMON_HEADER	The sequence number assigned to each common table data type listed in sbyn_common_header (column name "common_header_id").
SBYN_COMMON_DETAIL	The sequence number assigned to each common table data element listed in sbyn_common_detail (column name "common_detail_id").
OBJECT_NAME	Each parent and child object system record table is assigned a sequential ID. The column names are named after the object (for example, sbyn_address has a sequential column named "addressid"). The parent object ID is included in each child object table.
*OBJECT_NAME*SBR	Each parent and child object SBR table is assigned a sequential ID. The column names are named after the object (for example, sbyn_addresssbr has a sequential column named "addressid"). The parent object ID is included in each child object SBR table.

SBYN_SYSTEMOBJECT

This table stores information about the system records in the database, including their local ID and source system pairs. It also stores transactional information, such as the create or update date and function. The differences in data types between Oracle and SQL Server are noted in Table 19. The Oracle type is on the first line, and the SQL Server type is on the second.

TABLE 19 SBYN_SYSTEMOBJECT Table Description

Column Name	Data Type	Description
SYSTEMCODE	VARCHAR2(20) Varchar(20)	The processing code of the system associated with the local ID. This column cannot be null.
LID	VARCHAR2(25) Varchar(25)	The local ID associated with the system and EUID (the associated EUID is found in sbyn_enterprise). This column cannot be null.
CHILDTYPE	VARCHAR2(20) Varchar(20)	The type of object being processed (currently only the name of the parent object). This column is reserved for future use.
CREATEUSER	VARCHAR2(30) Varchar(30)	The user ID of the person who created the system record.

TABLE 19 SBYN_SYSTEMOBJECT Table Description *(Continued)*

Column Name	Data Type	Description
CREATEFUNCTION	VARCHAR2(20) Varchar(20)	The type of transaction that created the system record.
CREATEDATE	DATE datetime	The date the system record was created.
UPDATEUSER	VARCHAR2(30) Varchar(30)	The user ID of the person who last updated the system record.
UPDATEFUNCTION	VARCHAR2(20) Varchar(20)	The type of transaction that last updated the system record.
UPDATEDATE	DATE datetime	The date the system record was last updated.
STATUS	VARCHAR2(15) Varchar(15)	The status of the system record. The status can be one of these values: ■ active ■ inactive ■ merged

SBYN_SYSTEMS

This table stores information about each system integrated into the master index environment, including the system's processing code and name, a brief description, the format of the local IDs, and whether any of the system information should be masked. The differences in data types between Oracle and SQL Server are noted in Table 20. The Oracle type is on the first line, and the SQL Server type is on the second.

TABLE 20 SBYN_SYSTEMS Table Description

Column Name	Data Type	Description
SYSTEMCODE	VARCHAR2(20) Varchar(20)	The unique processing code of the system.
DESCRIPTION	VARCHAR2(120) Varchar(120)	A brief description of the system, or the system name. This is the value that appears in the tree view panes of the EDM for each system and local ID pair.
STATUS	CHAR(1)	The status of the system in the master index application. "A" indicates active and "D" indicates deactivated.

TABLE 20 SBYN_SYSTEMS Table Description *(Continued)*

Column Name	Data Type	Description
ID_LENGTH	NUMBER numeric(38, 0)	The length of the local identifiers assigned by the system. This length does not include any additional characters added by the input mask.
FORMAT	VARCHAR2(60) Varchar(60)	The required data pattern for the local IDs assigned by the system. For more information about possible values and using Java patterns, see "Patterns" in the class list for `java.util.regex` in the Javadocs provided with the Java™ 2 Platform, Standard Edition (J2SE™ platform). Note that the data pattern is also limited by the input mask described below. All regex patterns are supported if there is no input mask.
INPUT_MASK	VARCHAR2(60) Varchar(60)	A mask used by the EDM to add punctuation to the local ID. For example, the input mask **DD-DDD-DDD** inserts a hyphen after the second and fifth characters in an 8-digit ID. These character types can be used. ■ **D** – Numeric character ■ **L** – Alphabetic character ■ **A** – Alphanumeric character
VALUE_MASK	VARCHAR2(60) Varchar(60)	A mask used to strip any extra characters that were added by the input mask for database storage. The value mask is the same as the input mask, but with an "x" in place of each punctuation mark. Using the input mask described above, the value mask is **DDxDDDxDDD**. This strips the hyphens before storing the ID.
CREATE_DATE	DATE datetime	The date the system information was inserted into the database.
CREATE_USERID	VARCHAR2(20) Varchar(20)	The logon ID of the user who inserted the system information into the database.
UPDATE_DATE	DATE datetime	The most recent date the system's information was updated.
UPDATE_USERID	VARCHAR2(20) Varchar(20)	The logon ID of the user who last updated the system's information.

SBYN_SYSTEMSBR

This table stores transactional information about the system records for the SBR, such as the create or update date and function. The sbyn_systemsbr table is indirectly linked to the sbyn_systemobjects table through sbyn_enterprise. The differences in data types between Oracle and SQL Server are noted in Table 21. The Oracle type is on the first line, and the SQL Server type is on the second.

TABLE 21 SBYN_SYSTEMSBR Table Description

Column Name	Data Type	Description
EUID	VARCHAR2(20) Varchar(20)	The EUID associated with system record (the associated system and local ID are found in sbyn_enterprise). This column cannot be null.
CHILDTYPE	VARCHAR2(20) Varchar(20)	The type of object being processed (currently only the name of the parent object). This column is reserved for future use.
CREATESYSTEM	VARCHAR2(20) Varchar(20)	The system in which the system record was created.
CREATEUSER	VARCHAR2(30) Varchar(30)	The user ID of the person who created the system record.
CREATEFUNCTION	VARCHAR2(20) Varchar(20)	The type of transaction that created the system record.
CREATEDATE	DATE datetime	The date the system object was created.
UPDATEUSER	VARCHAR2(30) Varchar(30)	The user ID of the person who last updated the system record.
UPDATEFUNCTION	VARCHAR2(20) Varchar(20)	The type of transaction that last updated the system record.
UPDATEDATE	DATE datetime	The date the system object was last updated.
STATUS	VARCHAR2(15) Varchar(15)	The status of the enterprise record. The status can be one of these values: ■ active ■ inactive ■ merged

TABLE 21 SBYN_SYSTEMSBR Table Description *(Continued)*

Column Name	Data Type	Description
REVISIONNUMBER	NUMBER(38) numeric(38, 0)	The revision number of the SBR. This is used for version control.

SBYN_TRANSACTION

This table stores a history of changes made to each record in the master index application, allowing you to view a transaction history and to undo certain actions, such as merging two object records. The differences in data types between Oracle and SQL Server are noted in Table 22. The Oracle type is on the first line and the SQL Server type is on the second. This table also includes one column that has a different name for Oracle and for SQL Server.

TABLE 22 SBYN_TRANSACTION Table Description

Column Name	Data Type	Description
TRANSACTIONNUMBER	VARCHAR2(20) Varchar(20)	The unique number of the transaction.
LID1	VARCHAR2(25) Varchar(25)	This column is reserved for future use.
LID2	VARCHAR2(25) Varchar(25)	The local ID of the second system record involved in the transaction.
EUID1	VARCHAR2(20) Varchar(20)	This column is reserved for future use.
EUID2	VARCHAR2(20) Varchar(20)	The EUID of the second object record involved in the transaction.
FUNCTION (Oracle) OPERATION (SQL Server)	VARCHAR2(20) Varchar(20)	The type of transaction that occurred, such as update, add, merge, and so on.
SYSTEMUSER	VARCHAR2(30) Varchar(30)	The logon ID of the user who performed the transaction.
TIMESTAMP	TIMESTAMP datetime	The date and time the transaction occurred.
DELTA	BLOB varbinary(MAX)	A list of the changes that occurred to system records as a result of the transaction.

TABLE 22 SBYN_TRANSACTION Table Description *(Continued)*

Column Name	Data Type	Description
SYSTEMCODE	VARCHAR2(20) Varchar(20)	The processing code of the source system in which the transaction originated.
LID	VARCHAR2(25) Varchar(25)	The local ID of the system record involved in the transaction.
EUID	VARCHAR2(20) Varchar(20)	The EUID of the enterprise record involved in the transaction.

SBYN_USER_CODE

This table is similar to the sbyn_common_header and sbyn_common_detail tables in that it stores processing codes and drop-down list values. This table is used when the value of one field is dependent on the value of another. For example, if you store credit card information, you could list each credit card type and specify a required format for the credit card number field. The data stored in this table includes the processing code, a brief description, and the format of the dependent fields.

The differences in data types between Oracle and SQL Server are noted in Table 23. The Oracle type is on the first line, and the SQL Server type is on the second.

TABLE 23 SBYN_USER_CODE Table Description

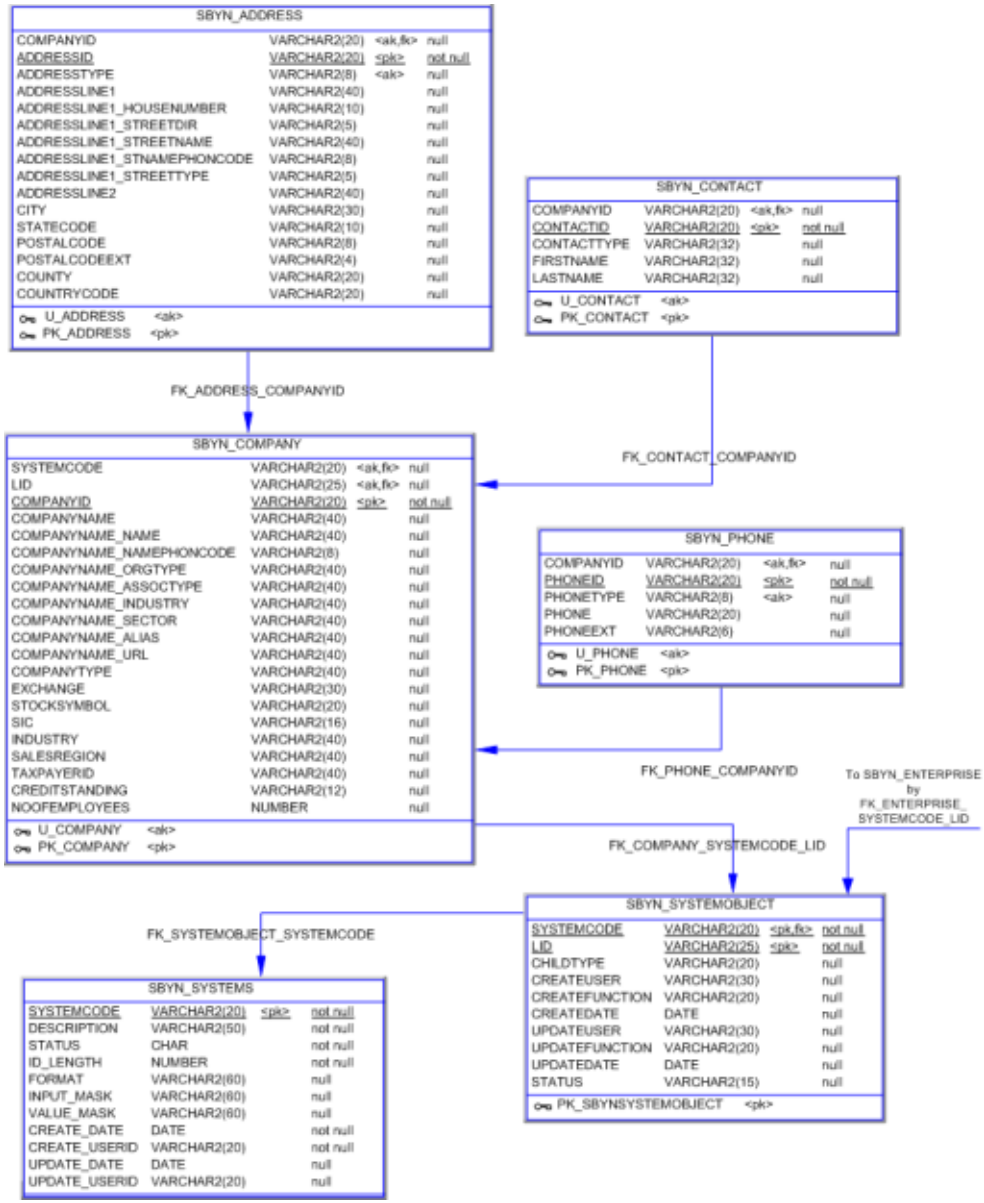
Column Name	Data Type	Description
CODE_LIST	VARCHAR2(20) Varchar(20)	The code list name of the user code type (using the credit card example above, this might be similar to "CREDCARD"). This column links the values for each list.
CODE	VARCHAR2(20) Varchar(20)	The processing code of each user code element.
DESCRIPTION	VARCHAR2(50) Varchar(50)	A brief description or name for the user code. This is the value that appears in the drop-down list.
FORMAT	VARCHAR2(60) Varchar(60)	The required data pattern for the field that is constrained by the user code. For more information about possible values and using Java patterns, see "Patterns" in the class list for `java.util.regex` in the Javadocs provided with the J2SE platform. Note that the data pattern is also limited by the input mask described below. All regex patterns are supported if there is no input mask.

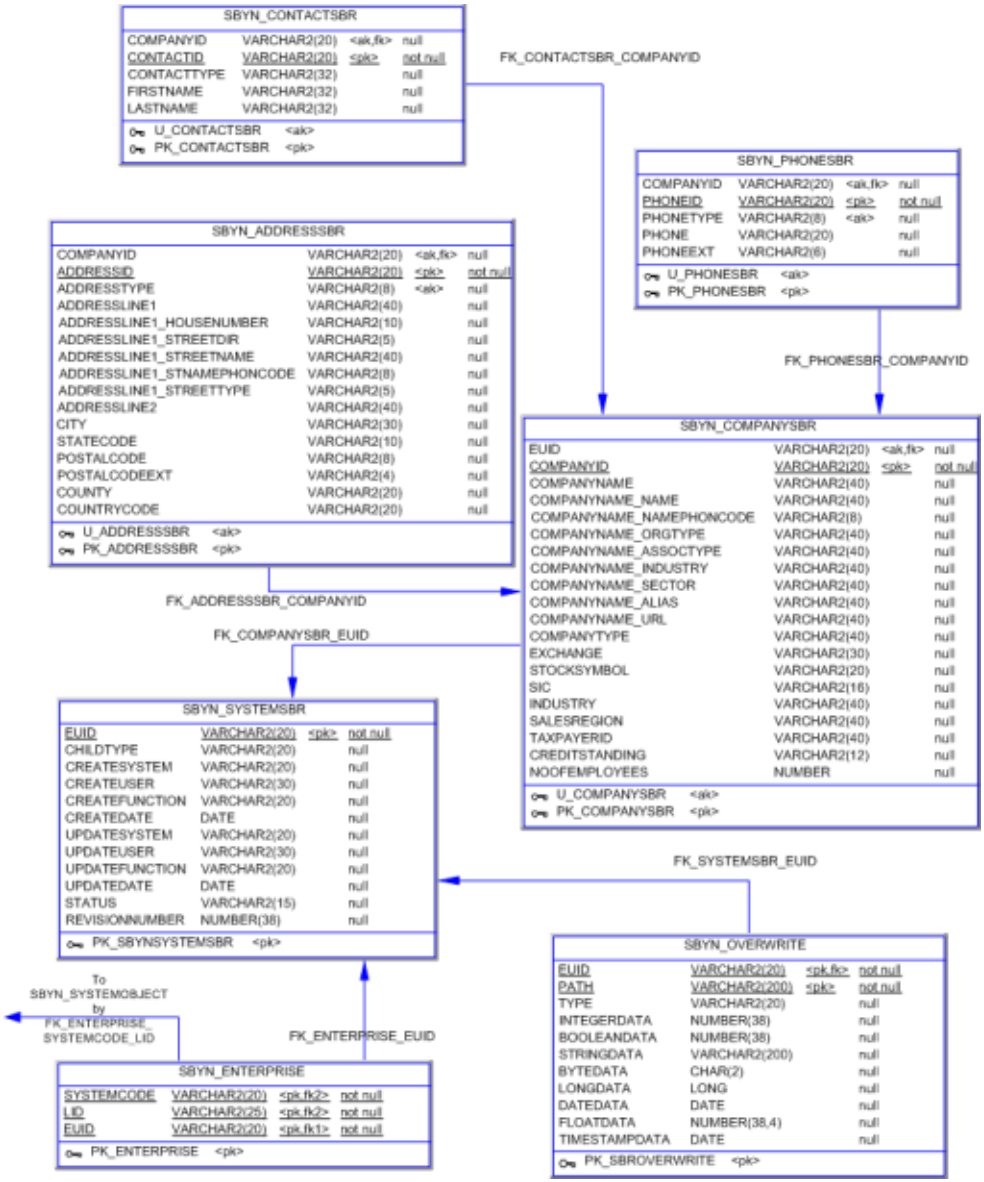
TABLE 23 SBYN_USER_CODE Table Description *(Continued)*

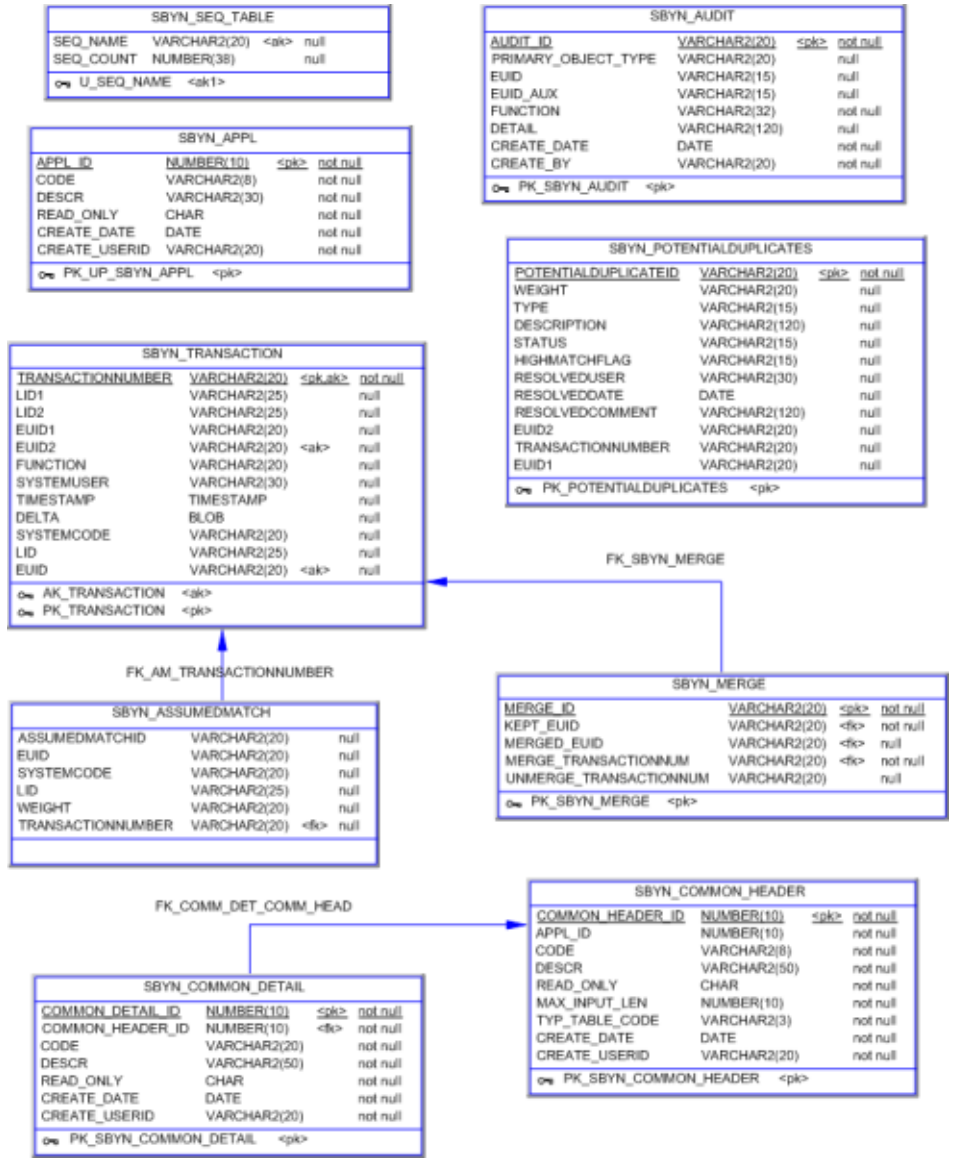
Column Name	Data Type	Description
INPUT_MASK	VARCHAR2(60) Varchar(60)	A mask used by the EDM to add punctuation to the constrained field. For example, the input mask **DD-DDD-DDD** inserts a hyphen after the second and fifth characters in an 8-digit ID. These character types can be used. ■ **D** – Numeric character ■ **L** – Alphabetic character ■ **A** – Alphanumeric character
VALUE_MASK	VARCHAR2(60) Varchar(60)	A mask used to strip any extra characters that were added by the input mask for database storage. The value mask is the same as the input mask, but with an "x" in place of each punctuation mark. Using the input mask described above, the value mask is **DDxDDDxDDD**. This strips the hyphens before storing the ID.

Sample Master Index Database Model (Repository)

The diagrams on the following pages illustrate the table structure and relationships for a sample Oracle database designed for storing information about companies. The diagrams display attributes for each database column, such as the field name, data type, whether the field can be null, and primary keys. They also show directional relationships between tables and the keys by which the tables are related. This diagram is very similar to SQL Server, with the exception of a few column name changes and some different data types as noted in the tables above.


SBYN_ADDRESS
COMPANYID VARCHAR2(20) <ak,fk> null
ADDRESSID VARCHAR2(20) <pk> not null
ADDRESSTYPE VARCHAR2(8) <ak> null
ADDRESSLINE1 VARCHAR2(40) null
ADDRESSLINE1_HOUSENUMBER VARCHAR2(10) null
ADDRESSLINE1_STREETDIR VARCHAR2(5) null
ADDRESSLINE1_STREETNAME VARCHAR2(40) null
ADDRESSLINE1_STNAMEPHONCODE VARCHAR2(8) null
ADDRESSLINE1_STREETTYPE VARCHAR2(5) null
ADDRESSLINE2 VARCHAR2(40) null
CITY VARCHAR2(30) null
STATECODE VARCHAR2(10) null
POSTALCODE VARCHAR2(8) null
POSTALCODEEXT VARCHAR2(4) null
COUNTY VARCHAR2(20) null
COUNTRYCODE VARCHAR2(20) null
U_ADDRESS <ak>
PK_ADDRESS <pk>
SBYN_CONTACT
COMPANYID VARCHAR2(20) <ak,fk> null
CONTACTID VARCHAR2(20) <pk> not null
CONTACTTYPE VARCHAR2(32) null
FIRSTNAME VARCHAR2(32) null
LASTNAME VARCHAR2(32) null
U_CONTACT <ak>
PK_CONTACT <pk>
FK_ADDRESS_COMPANYID
SBYN_COMPANY
SYSTEMCODE VARCHAR2(20) <ak,fk> null
LID VARCHAR2(25) <ak,fk> null
COMPANYID VARCHAR2(20) <pk> not null
COMPANYNAME VARCHAR2(40) null
COMPANYNAME_NAME VARCHAR2(40) null
COMPANYNAME_NAMEPHONCODE VARCHAR2(8) null
COMPANYNAME_ORGTYPE VARCHAR2(40) null
COMPANYNAME_ASSOCTYPE VARCHAR2(40) null
COMPANYNAME_INDUSTRY VARCHAR2(40) null
COMPANYNAME_SECTOR VARCHAR2(40) null
COMPANYNAME_ALIAS VARCHAR2(40) null
COMPANYNAME_URL VARCHAR2(40) null
COMPANYTYPE VARCHAR2(40) null
EXCHANGE VARCHAR2(30) null
STOCKSYMBOL VARCHAR2(20) null
SIC VARCHAR2(16) null
INDUSTRY VARCHAR2(40) null
SALESREGION VARCHAR2(40) null
TAXPAYERID VARCHAR2(40) null
CREDITSTANDING VARCHAR2(12) null
NOOFEMPLOYEES NUMBER null
U_COMPANY <ak>
PK_COMPANY <pk>
FK_CONTACT_COMPANYID
SBYN_PHONE
COMPANYID VARCHAR2(20) <ak,fk> null
PHONEID VARCHAR2(20) <pk> not null
PHONETYPE VARCHAR2(8) <ak> null
PHONE VARCHAR2(20) null
PHONEEXT VARCHAR2(6) null
U_PHONE <ak>
PK_PHONE <pk>
FK_PHONE_COMPANYID
To SBYN_ENTERPRISE by FK_ENTERPRISE_SYSTEMCODE_LID
FK_COMPANY_SYSTEMCODE_LID
SBYN_SYSTEMOBJECT
SYSTEMCODE VARCHAR2(20) <pk,fk> not null
LID VARCHAR2(25) <pk> not null
CHILDTYPE VARCHAR2(20) null
CREATEUSER VARCHAR2(30) null
CREATEFUNCTION VARCHAR2(20) null
CREATEDATE DATE null
UPDATEUSER VARCHAR2(30) null
UPDATEFUNCTION VARCHAR2(20) null
UPDATEDATE DATE null
STATUS VARCHAR2(15) null
PK_SBYNSYSTEMOBJECT <pk>
FK_SYSTEMOBJECT_SYSTEMCODE
SBYN_SYSTEMS
SYSTEMCODE VARCHAR2(20) <pk> not null
DESCRIPTION VARCHAR2(50) not null
STATUS CHAR not null
ID_LENGTH NUMBER not null
FORMAT VARCHAR2(60) null
INPUT_MASK VARCHAR2(60) null
VALUE_MASK VARCHAR2(60) null
CREATE_DATE DATE not null
CREATE_USERID VARCHAR2(20) not null
UPDATE_DATE DATE null
UPDATE_USERID VARCHAR2(20) null


SBYN_CONTACTSBR
COMPANYID VARCHAR2(20) <ak,fk> null
CONTACTID VARCHAR2(20) <pk> not null
CONTACTTYPE VARCHAR2(32) null
FIRSTNAME VARCHAR2(32) null
LASTNAME VARCHAR2(32) null
U_CONTACTSBR <ak>
PK_CONTACTSBR <pk>
FK_CONTACTSBR_COMPANYID
SBYN_PHONESBR
COMPANYID VARCHAR2(20) <ak,fk> null
PHONEID VARCHAR2(20) <pk> not null
PHONETYPE VARCHAR2(8) <ak> null
PHONE VARCHAR2(20) null
PHONEEXT VARCHAR2(6) null
U_PHONESBR <ak>
PK_PHONESBR <pk>
FK_PHONESBR_COMPANYID
SBYN_ADDRESSSBR
COMPANYID VARCHAR2(20) <ak,fk> null
ADDRESSID VARCHAR2(20) <pk> not null
ADDRESSTYPE VARCHAR2(8) <ak> null
ADDRESSLINE1 VARCHAR2(40) null
ADDRESSLINE1_HOUSENUMBER VARCHAR2(10) null
ADDRESSLINE1_STREETDIR VARCHAR2(5) null
ADDRESSLINE1_STREETNAME VARCHAR2(40) null
ADDRESSLINE1_STNAMEPHONCODE VARCHAR2(8) null
ADDRESSLINE1_STREETTYPE VARCHAR2(5) null
ADDRESSLINE2 VARCHAR2(40) null
CITY VARCHAR2(30) null
STATECODE VARCHAR2(10) null
POSTALCODE VARCHAR2(8) null
POSTALCODEEXT VARCHAR2(4) null
COUNTY VARCHAR2(20) null
COUNTRYCODE VARCHAR2(20) null
U_ADDRESSSBR <ak>
PK_ADDRESSSBR <pk>
FK_ADDRESSSBR_COMPANYID
SBYN_COMPANYSBR
EUID VARCHAR2(20) <ak,fk> null
COMPANYID VARCHAR2(20) <pk> not null
COMPANYNAME VARCHAR2(40) null
COMPANYNAME_NAME VARCHAR2(40) null
COMPANYNAME_NAMEPHONCODE VARCHAR2(8) null
COMPANYNAME_ORGTYPE VARCHAR2(40) null
COMPANYNAME_ASSOCTYPE VARCHAR2(40) null
COMPANYNAME_INDUSTRY VARCHAR2(40) null
COMPANYNAME_SECTOR VARCHAR2(40) null
COMPANYNAME_ALIAS VARCHAR2(40) null
COMPANYNAME_URL VARCHAR2(40) null
COMPANYTYPE VARCHAR2(40) null
EXCHANGE VARCHAR2(30) null
STOCKSYMBOL VARCHAR2(20) null
SIC VARCHAR2(16) null
INDUSTRY VARCHAR2(40) null
SALESREGION VARCHAR2(40) null
TAXPAYERID VARCHAR2(40) null
CREDITSTANDING VARCHAR2(12) null
NOOFEMPLOYEES NUMBER null
U_COMPANYSBR <ak>
PK_COMPANYSBR <pk>
FK_COMPANYSBR_EUID
SBYN_SYSTEMSBR
EUID VARCHAR2(20) <pk> not null
CHILDTYPE VARCHAR2(20) null
CREATESYSTEM VARCHAR2(20) null
CREATEUSER VARCHAR2(30) null
CREATEFUNCTION VARCHAR2(20) null
CREATEDATE DATE null
UPDATESYSTEM VARCHAR2(20) null
UPDATEUSER VARCHAR2(30) null
UPDATEFUNCTION VARCHAR2(20) null
UPDATEDATE DATE null
STATUS VARCHAR2(15) null
REVISIONNUMBER NUMBER(38) null
PK_SBYNSYSTEMSBR <pk>
FK_SYSTEMSBR_EUID
SBYN_OVERWRITE
EUID VARCHAR2(20) <pk,fk> not null
PATH VARCHAR2(200) <pk> not null
TYPE VARCHAR2(20) null
INTEGERDATA NUMBER(38) null
BOOLEANDATA NUMBER(38) null
STRINGDATA VARCHAR2(200) null
BYTEDATA CHAR(2) null
LONGDATA LONG null
DATEDATA DATE null
FLOATDATA NUMBER(38,4) null
TIMESTAMPDATA DATE null
PK_SBROVERWRITE <pk>
To SBYN_SYSTEMOBJECT by FK_ENTERPRISE_SYSTEMCODE_LID
FK_ENTERPRISE_EUID
SBYN_ENTERPRISE
SYSTEMCODE VARCHAR2(20) <pk,fk2> not null
LID VARCHAR2(25) <pk,fk2> not null
EUID VARCHAR2(20) <pk,fk1> not null
PK_ENTERPRISE <pk>



Working with the Master Index Java API (Repository)

Sun Master Index provides several Java classes and methods to use to transform and process data in a master index project. The master index API is specifically designed to help you maintain the integrity of the data in the database by providing specific methods for updating, adding, and merging records in the database.

The following topics provide information about the master index API and describe the dynamic API:

Master Index Java Class Types (Repository)

Sun Master Index provides a set of static API classes that can be used with any object structure and any Sun master index application. Sun Master Index also generates several dynamic API classes that are specific to the object structure of each master index application. The dynamic classes contain similar methods, but the number and names of methods change depending on the object structure. In addition, several methods are generated in an OTD for use in external system Collaborations and another set of methods is generated for use within an Business Process. For detailed information about the static classes and methods, see the Javadocs.

The following topics provide additional information about the different types of Java classes:

Static Master Index Java Classes (Repository)

Static classes provide the methods you need to perform basic data cleansing functions against incoming data, such as performing searches, reviewing potential duplicates, adding and updating records, and merging and unmerging records. The primary class containing these functions is the `MasterController` class, which includes the `executeMatch` method. Several classes support the `MasterController` class by defining additional objects and functions. Documentation for the static methods is provided in Javadoc format. The static classes are listed and described in the Javadocs provided with Sun Master Index.

Dynamic Master Index Object Classes (Repository)

When you generate a master index project, several dynamic methods are created that are specific to the object structure defined for the master index application. This includes classes that define each object in the object structure and that allow you to work with the data in each object. If the object structure is modified, regenerating the project updates the dynamic methods for the new structure.

Dynamic OTD Methods

When you generate a master index project, a method OTD is created that contains Java methods to help you define how records are processed into the master index database from external systems. Like the dynamic object classes, these methods are based on the object structure. They rely on the dynamic object classes to create the objects in the master index application and to define and retrieve field values for those objects. Regenerating the master index application updates the methods to reflect any changes to the object structure.

Dynamic Business Process Methods

When you generate a master index project, several methods are listed under the method OTD folder that are designed for use within a Business Process. These methods are a subset of the master index API that can be used to query a master index database using a web-based interface. As with the dynamic OTD methods, the Business Process methods are also based on the defined object structure. Regenerating a project updates these methods to reflect any changes to the object structure.

Dynamic Master Index Object Classes (Repository)

Several dynamic classes are generated for each master index application for use in Collaborations. One class is created for each parent and child object defined in the object structure.

The following topics list and describe the dynamic object classes:

- "Master Index Parent Object Classes (Repository)" on page 63
- "Master Index Child Object Classes (Repository)" on page 76

Master Index Parent Object Classes (Repository)

A Java class is created to represent the parent object defined in the object definition of the master index application. The methods in this class provide the ability to create a parent object and to set or retrieve the field values for that object.

The name of the parent object class is the same as the name of each parent object, with the word "Object" appended. For example, if the parent object in your object structure is "Person", the name of the parent class is "PersonObject". The methods in this class include a constructor

method for the parent object, and get and set methods for each field defined for the parent object. Most methods are named based on the name of the parent object and the fields and child objects defined for that object. In the following methods described for the parent object, *ObjectName* indicates the name of the parent object, *Child* indicates the name of a child object, and *Field* indicates the name of a field defined for the parent object.

Definition

`class` *ObjectName*`Object`

Methods

*ObjectName*Object

Description

This is the user-defined object name class for the parent object. You can instantiate this class to create a new instance of the parent object class.

Syntax

`new` *ObjectName*`Object()`

Parameters

None.

Returns

An instance of the parent object.

Throws

`ObjectException`

add*Child*

Description

This method associates a new child object with the parent object. The new child object is of the type specified in the method name. For example, to associate a new address object with a parent object, call `addAddress`.

Syntax

`void add`*`Child`*`(`*`Child`*`Object` *`child`*`)`

Note – The type of object passed as a parameter depends on the child object to associate with the parent object. For example, the syntax for associating an address object is as follows:

```
void addAddress(AddressObject address)
```

Parameters

Name	Type	Description
child	*Child*Object	A child object to associate with the parent object. The name and type of the parameter is specified by the child object name.

Returns

None.

Throws

`ObjectException`

addSecondaryObject

Description

This method associates a new child object with the parent object. The object node passed as the parameter defines the child object type.

Syntax

```
void addSecondaryObject(ObjectNode obj)
```

Parameters

Name	Type	Description
obj	ObjectNode	An ObjectNode representing the child object to associate with the parent object.

Returns

None.

Throws

```
SystemObjectException
```

copy

Description

This method copies the structure and field values of the specified object node.

Syntax

```
ObjectNode copy()
```

Parameters

None.

Returns

A copy of the object node.

Throws

`ObjectException`

dropSecondaryObject

Description

This method removes a child object associated with the parent object (in the memory copy of the object). The object node passed in as the parameter defines the child object type. Use this method to remove a child object before it has been committed to the database. This method is similar to `ObjectNode.removeChild`. Use `ObjectNode.deleteChild` to remove the child object permanently from the database.

Syntax

```
void dropSecondaryObject(ObjectNode obj)
```

Parameters

Name	Type	Description
obj	ObjectNode	An ObjectNode representing the child object to drop from the parent object.

Returns

None.

Throws

`SystemObjectException`

get*ObjectName*Id

Description

This method retrieves the unique identification code (primary key) of the object, as assigned by the master index application.

Syntax

`String get`*`ObjectName`*`Id()`

Parameters

None.

Returns

A string containing the unique ID of the parent object.

Throws

`ObjectException`

get*Child*

Description

This method retrieves all child objects associated with the parent object that are of the type specified in the method name. For example, to retrieve all address objects associated with a parent object, call `getAddress`.

Syntax

`Collection get`*`Child`*`()`

Parameters

None.

Returns

A collection of child objects of the type specified in the method name.

Throws

None.

get*Field*

Description

This method retrieves the value of the field specified in the method name. Each getter method is named according to the fields defined for the parent object. For example, if the parent object contains a field named FirstName, the getter method for this field is named `getFirstName`.

Syntax

`String get`*`Field`*`()`

Note – The syntax for the getter methods depends on the type of data specified for the field in the object structure. For example, the getter method for a date field would have the following syntax:

```
Date get<i>Field</i>
```

Parameters

None.

Returns

The value of the specified field. The type of data returned depends on the data type defined in the object definition.

Throws

`ObjectException`

getChildTags

Description

This method retrieves a list of the names of all child object types defined for the object structure.

Syntax

```
ArrayList getChildTags()
```

Parameters

None.

Returns

An array of child object names.

Throws

None.

getMetaData

Description

This method retrieves the metadata for the parent object.

Syntax

```
AttributeMetaData getMetaData()
```

Parameters

None.

Returns

An AttributeMetaData object containing the parent object's metadata.

Throws

None.

getSecondaryObject

Description

This method retrieves all child objects that are associated with the parent object and are of the specified type.

Syntax

```
Collection getSecondaryObject(String type)
```

Parameters

Name	Type	Description
type	String	The child type of the objects to retrieve.

Returns

A collection of child objects of the specified type.

Throws

`SystemObjectException`

isAdded

Description

This method retrieves the value of the "add flag" for the parent object. The add flag indicates whether the object will be added.

Syntax

```
String isAdded()
```

Parameters

None.

Returns

A Boolean value indicating whether the add flag is set to true or false.

Throws

```
ObjectException
```

isRemoved

Description

This method retrieves the value of the "remove flag" for the parent object. The remove flag indicates whether the object will be removed.

Syntax

```
String isRemoved()
```

Parameters

None.

Returns

A Boolean value indicating whether the remove flag is set to true or false.

Throws

```
ObjectException
```

isUpdated

Description

This method retrieves the value of the "update flag" for the parent object. The update flag indicates whether the object will be updated.

Syntax

```
String isUpdated()
```

Parameters

None.

Returns

A Boolean value indicating whether the update flag is set to true or false.

Throws

`ObjectException`

set*ObjectName*Id

Description

This method sets the value of the *ObjectName*Id field in the parent object.

Syntax

`void set`*`ObjectName`*`Id(Object value)`

Parameters

Name	Type	Description
value	Object	An object containing the value of the *ObjectName*Id field.

Returns

None.

Throws

`ObjectException`

set*Field*

Description

This method sets the value of the field specified in the method name. Each setter method is named according to the fields defined for the parent object. For example, if the parent object contains a field named "DateOfBirth", the setter method for this field is named `setDateOfBirth`. A setter method is created for each field in the parent object, including any fields containing standardized or phonetic data.

Syntax

`void set`*`Field`*`(Object value)`

Parameters

Name	Type	Description
value	Object	An object containing the value of the field specified by the method name.

Returns

None.

Throws

`ObjectException`

setAddFlag

Description

This method sets the "add flag" of the parent object. The add flag indicates whether the object will be added.

Syntax

```
void setAddFlag(boolean flag)
```

Parameters

Name	Type	Description
flag	Boolean	An indicator of whether the add flag is set to true or false.

Returns

None.

Throws

None.

setRemoveFlag

Description

This method sets the "remove flag" of the parent object. The remove flag indicates whether the object will be removed.

Syntax

```
void setRemoveFlag(boolean e)
```

Parameters

Name	Type	Description
e	Boolean	An indicator of whether the remove flag is set to true or false.

Returns

None.

Throws

None.

setUpdateFlag

Description

This method sets the "update flag" of the parent object. The update flag indicates whether the object will be updated.

Syntax

```
void setUpdateFlag(boolean flag)
```

Parameters

Name	Type	Description
flag	Boolean	An indicator of whether the update flag is set to true or false.

Returns

None.

Throws

None.

structCopy

Description

This method copies the structure of the specified object node.

Syntax

```
ObjectNode structCopy()
```

Parameters

None.

Returns

A copy of the structure of the object node.

Throws

`ObjectException`

Master Index Child Object Classes (Repository)

One Java class is created for each child object defined in the object definition of the master index application. If the object definition contains three child objects, three child object classes are created. The methods in these classes provide the ability to create the child objects and to set or retrieve the field values for those objects.

The name of each child object class is the same as the name of the child object, with the word "Object" appended. For example, if a child object in your object structure is named "Address", the name of the corresponding child class is `AddressObject`. The methods in these classes include a constructor method for the child object, and get and set methods for each field defined for the child object. Most methods are named based on the name of the child object and the fields defined for that object. In the methods listed below, *Child* indicates the name of the child object and *Field* indicates the name of each field defined for that object.

Definition

`class` *Child*`Object`

Methods

*Child*Object

Description

This method represents the child object class. This class can be instantiated to create a new instance of a child object class.

Syntax

`new` *Child*`Object()`

Parameters

None.

Returns

An instance of the child object.

Throws

`ObjectException`

copy

Description

This method copies the structure and field values of the specified object node.

Syntax

```
ObjectNode copy()
```

Parameters

None.

Returns

A copy of the object node.

Throws

`ObjectException`

get*Child*Id

Description

This method retrieves the unique identification code (primary key) of the object, as assigned by the master index application.

Syntax

`String get`*`Child`*`Id()`

Parameters

None.

Returns

A string containing the unique ID of the child object.

Throws

`ObjectException`

get*Field*

Description

This method retrieves the value of the field specified in the method name. Each getter method is named according to the fields defined for the child object. For example, if the child object contains a field named "TelephoneNumber", the getter method for this field is named `getTelephoneNumber`. A getter method is created for each field in the object, including fields that store standardized or phonetic data.

Syntax

`String get`*`Field`*`()`

Note – The syntax for the getter methods depends on the type of data specified for the field in the object structure. For example, the getter method for a date field would have the following syntax:

```
Date getField
```

Parameters

None.

Returns

The value of the specified field. The type of data returned depends on the data type defined in the object definition.

Throws

`ObjectException`

getMetaData

Description

This method retrieves the metadata for the child object.

Syntax

```
AttributeMetaData getMetaData()
```

Parameters

None.

Returns

An AttributeMetaData object containing the child object's metadata.

Throws

None.

getParentTag

Description

This method retrieves the name of the parent object of the child object.

Syntax

```
String getParentTag()
```

Parameters

None.

Returns

A string containing the name of the parent object.

Throws

None.

set*Child*Id

Description

This method sets the value of the *Child*Id field in the child object.

Syntax

```
void set<i>Child</i>Id(Object value)
```

Parameters

Name	Type	Description
value	Object	An object containing the value of the *Child*Id field.

Returns

None.

Throws

```
ObjectException
```

set*Field*

Description

This method sets the value of the field specified in the method name. Each setter method is named according to the fields defined for the parent object. For example, if the parent object contains a field named "CompanyName", the setter method for this field is named *setCompanyName*.

Syntax

```
void set<i>Field</i>(Object value)
```

Parameters

Name	Type	Description
value	Object	An object containing the value of the field specified by the method name.

Returns

None.

Throws

`ObjectException`

structCopy

Description

This method copies the structure of the specified object node.

Syntax

```
ObjectNode structCopy()
```

Parameters

None.

Returns

A copy of the structure of the object node.

Throws

`ObjectException`

Dynamic Master Index OTD Methods (Repository)

A set of Java methods are created in an OTD for use in the master index Collaborations. These methods wrap static Java API methods, allowing them to work with the dynamic object classes. Many OTD methods return objects of the dynamic object type, or they use these objects as parameters. In the following methods described for the OTD methods, *ObjectName* indicates the name of the parent object.

Dynamic Master Index OTD Methods (Repository)

The following topics list and describe the dynamic OTD methods.

activateEnterpriseRecord

Description

This method changes the status of a deactivated enterprise object back to active.

Syntax

```
void activateEnterpriseRecord(String euid)
```

Parameters

Name	Type	Description
euid	String	The EUID of the enterprise object to activate.

Returns

None.

Throws

- `RemoteException`
- `ProcessingException`
- `UserException`

activateSystemRecord

Description

This method changes the status of a deactivated system object back to active.

Syntax

```
void activateSystemRecord(String systemCode, String localId)
```

Parameters

Name	Type	Description
systemCode	String	The processing code of the system associated with the system record to be activated.
localID	String	The local identifier associated with the system record to be activated.

Returns

None.

Throws

- `RemoteException`
- `ProcessingException`
- `UserException`

addSystemRecord

Description

This method adds the system object to the enterprise object associated with the specified EUID.

Syntax

```
void addSystemRecord(String euid, SystemObjectBean systemObject)
```

Parameters

Name	Type	Description
euid	String	The EUID of the enterprise object to which you want to add the system object.
systemObject	SystemObjectBean	The Bean for the system object to be added to the enterprise object.

Returns

None.

Throws

- `RemoteException`
- `ProcessingException`
- `UserException`

deactivateEnterpriseRecord

Description

This method changes the status of an active enterprise object to inactive.

Syntax

```
void deactivateEnterpriseRecord(String euid)
```

Parameters

Name	Type	Description
euid	String	The EUID of the enterprise object to deactivate.

Returns

None.

Throws

- `RemoteException`
- `ProcessingException`
- `UserException`

deactivateSystemRecord

Description

This method changes the status of an active system object to inactive.

Syntax

```
void deactivateSystemRecord(String systemCode, String localId)
```

Parameters

Name	Type	Description
systemCode	String	The system code of the system object to deactivate.
localid	String	The local ID of the system object to deactivate.

Returns

None.

Throws

- `RemoteException`
- `ProcessingException`
- `UserException`

executeMatch

`executeMatch` is one of two methods you can call to process an incoming system object based on the configuration defined for the Manager Service and associated runtime components (the second method is "executeMatchUpdate" on page 86). This process searches for possible matches in the database and contains the logic to add a new record or update existing records in the database. One of the two execute match methods should be used for inserting or updating a record in the database.

The following runtime components configure `executeMatch`.

- The Query Builder defines the blocking queries used for matching.
- The Threshold file (`master.xml`) specifies which blocking query to use and specifies matching parameters, including duplicate and match thresholds.
- The pass controller and block picker classes specify how the blocking query is executed.

Note – If `executeMatch` determines that an existing system record will be updated by the incoming record, it replaces the entire existing record with the information in the new record. This could result in loss of data; for example, if the incoming record does not include all address information, existing address information could be lost. To avoid this, use the `executeMatchUpdate` method instead.

Syntax

```
MatchColResult executeMatch(SystemObjectBean systemObject)
```

Parameters

Name	Type	Description
systemObject	SystemObjectBean	The Bean for the system object to be added to or updated in the enterprise object.

Returns

A match result object containing the results of the matching process.

Throws

- `RemoteException`
- `ProcessingException`
- `UserException`

executeMatchUpdate

Like "executeMatch" on page 85, `executeMatchUpdate` processes the system object based on the configuration defined for the Manager Service and associated runtime components. It is configured by the same runtime components as `executeMatch`. One of these two execute match methods should be used for inserting or updating a record in the database.

The primary difference between these two methods is that when `executeMatchUpdate` finds that an incoming record matches an existing record, only the changed data is updated. With `executeMatch`, the entire existing record is replaced by the incoming record. The `executeMatchUpdate` method differs from `executeMatch` in the following ways:

- If a partial record is received, `executeMatchUpdate` only updates fields whose values are different in the incoming record. Unless the `clearFieldIndicator` field is used, empty or null fields in the incoming record do not update existing values.
- The `clearFieldIndicator` field can be used to null out specific fields.
- Child objects in the existing record are not deleted if they are not present in the incoming record.
- Child objects in the existing record are updated if the same key field value is found in both the incoming and existing records.
- To allow a child object to be removed from the parent object when using `executeMatchUpdate`, a new "delete" method is added to each child object bean.

Syntax

```
MatchColResult executeMatchUpdate(SystemObjectbean systemObject)
```

Parameters

Name	Type	Description
systemObject	SystemObjectBean	The Bean for the system object to be added to or updated in the enterprise object.

Returns

A match result object containing the results of the matching process.

Throws

- `RemoteException`
- `ProcessingException`
- `UserException`

findMasterController

This method obtains a handle to the `MasterController` class, providing access to all of the methods of that class. For more information about the available methods in this class, see the Javadoc provided with Sun Master Index.

Syntax

```
MasterController findMasterController()
```

Parameters

None.

Returns

A handle to the `com.stc.eindex.ejb.master.MasterController` class.

Throws

None.

getEnterpriseRecordByEUID

Description

This method returns the enterprise object associated with the specified EUID.

Syntax

`Enterprise`*`ObjectName`* `getEnterpriseRecordByEUID(String euid)`

Parameters

Name	Type	Description
euid	String	The EUID of the enterprise object you want to retrieve.

Returns

An enterprise object associated with the specified EUID or null if the enterprise object is not found.

Throws

- `RemoteException`
- `ProcessingException`
- `UserException`

getEnterpriseRecordByLID

Description

This method returns the enterprise object associated with the specified system code and local ID pair.

Syntax

`Enterprise`*`ObjectName`* `getEnterpriseRecordByLID(String system, String localid)`

Parameters

Name	Type	Description
system	String	The system code of a system associated with the enterprise object to find.
localid	String	A local ID associated with the specified system.

Returns

An enterprise object or null if the enterprise object is not found.

Throws

- `RemoteException`
- `ProcessingException`
- `UserException`

getEUID

Description

This method returns the EUID of the enterprise object associated with the specified system code and local ID.

Syntax

```
String getEUID(String system, String localid)
```

Parameters

Name	Type	Description
system	String	A known system code for the enterprise object.
localid	String	The local ID corresponding with the given system.

Returns

A string containing an EUID or null if the EUID is not found.

Throws

- `RemoteException`
- `ProcessingException`
- `UserException`

getLIDs

Description

This method retrieves the local ID and system pairs associated with the given EUID.

Syntax

`System`*`ObjectName`*`PK[] getLIDs(String euid)`

Parameters

Name	Type	Description
euid	String	The EUID of the enterprise object whose local ID and system pairs you want to retrieve.

Returns

An array of system object keys (System*ObjectName*PK objects) or null if no results are found.

Throws

- `RemoteException`
- `ProcessingException`
- `UserException`

getLIDsByStatus

Description

This method retrieves the local ID and system pairs that are of the specified status and that are associated with the given EUID.

Syntax

`System`*`ObjectName`*`PK[] getLIDsByStatus(String euid, String status)`

Parameters

Name	Type	Description
euid	String	The EUID of the enterprise object whose local ID and system pairs to retrieve.
status	String	The status of the local ID and system pairs you want to retrieve.

Returns

An array of system object keys (System*ObjectName*PK objects) or null if no system object keys are found.

Throws

- `RemoteException`
- `ProcessingException`
- `UserException`

getSBR

Description

This method retrieves the single best record (SBR) associated with the specified EUID.

Syntax

SBR*ObjectName* `getSBR(String euid)`

Parameters

Name	Type	Description
euid	String	The EUID of the enterprise object whose SBR you want to retrieve.

Returns

An SBR object or null if no SBR associated with the specified EUID is found.

Throws

- `RemoteException`
- `ProcessingException`
- `UserException`

getSystemRecord

Description

This method retrieves the system object associated with the given system code and local ID pair.

Syntax

```
System<i>ObjectName</i> getSystemRecord(String system, String localid)
```

Parameters

Name	Type	Description
system	String	The system code of the system object to retrieve.
localid	String	The local ID of the system object to retrieve.

Returns

A system object containing the results of the search or null if no system objects are found.

Throws

- `RemoteException`
- `ProcessingException`
- `UserException`

getSystemRecordsByEUID

Description

This method returns the active system objects associated with the specified EUID.

Syntax

```
System<i>ObjectName</i>[] getSystemRecordsByEUID(String euid)
```

Parameters

Name	Type	Description
euid	String	The EUID of the enterprise object whose system objects you want to retrieve.

Returns

An array of system objects associated with the specified EUID.

Throws

- `RemoteException`
- `ProcessingException`
- `UserException`

getSystemRecordsByEUIDStatus

Description

This method returns the system objects of the specified status that are associated with the given EUID.

Syntax

`System`*`ObjectName`*`[] getSystemRecordsByEUIDStatus(String euid, String status)`

Parameters

Name	Type	Description
euid	String	The EUID of the enterprise object whose system objects you want to retrieve.
status	String	The status of the system objects you want to retrieve.

Returns

An array of system objects associated with the specified EUID and status, or null if no system objects are found.

Throws

- `RemoteException`
- `ProcessingException`
- `UserException`

lookupLIDs

Description

This method first looks up the EUID associated with the specified source system and source local ID. It then retrieves the local ID and system pairs of the specified status that are associated with that EUID and are from the specified destination system. Note that both systems must be of the specified status or an error will occur.

Syntax

```
System<ObjectName>PK[] lookupLIDs(String sourceSystem, String sourceLID,
String destSystem, String status)
```

Parameters

Name	Type	Description
sourceSystem	String	The system code of the known system and local ID pair.
sourceLID	String	The local ID of the known system and local ID pair.
destSystem	String	The system of origin for the local ID and system pairs you want to retrieve.
status	String	The status of the local ID and system pairs to retrieve.

Returns

An array of system object keys (System*ObjectName*PK objects).

Throws

- `RemoteException`
- `ProcessingException`
- `UserException`

mergeEnterpriseRecord

Description

This method merges two enterprise objects, specified by their EUIDs.

Syntax

```
Merge<ObjectName>Result mergeEnterpriseRecord(String fromEUID, String toEUID,
boolean calculateOnly)
```

Parameters

Name	Type	Description
fromEUID	String	The EUID of the enterprise object that will not survive the merge.
toEUID	String	The EUID of the enterprise object that will survive the merge.
calculateOnly	boolean	An indicator of whether to commit changes to the database or to simply compute the merge results. Specify **false** to commit the changes.

Returns

A merge result object containing the results of the merge.

Throws

- `RemoteException`
- `ProcessingException`
- `UserException`

mergeSystemRecord

Description

This method merges two system objects, specified by their local IDs, from the specified system. The system objects can belong to a single enterprise object or to two different enterprise objects.

Syntax

```
Merge<ObjectName>Result mergeSystemRecord(String sourceSystem, String sourceLID,
String destLID, boolean calculateOnly)
```

Parameters

Name	Type	Description
sourceSystem	String	The processing code of the system to which the two system objects belong.
sourceLID	String	The local ID of the system object that will not survive the merge.
destLID	String	The local ID of the system object that will survive the merge.
calculateOnly	boolean	An indicator of whether to commit changes to the database or to simply compute the merge results. Specify **false** to commit the changes.

Returns

A merge result object containing the results of the merge.

Throws

- `RemoteException`
- `ProcessingException`
- `UserException`

searchBlock

Description

This method performs a blocking query against the database using the blocking query specified in the Threshold file and the criteria contained in the specified object bean.

Syntax

`Search`*`ObjectName`*`Result searchBlock(`*`ObjectName`*`Bean searchCriteria)`

Parameters

Name	Type	Description
searchCriteria	*ObjectName*Bean	The search criteria for the blocking query.

Returns

The results of the search.

Throws

- `RemoteException`
- `ProcessingException`
- `UserException`

searchExact

Description

This method performs an exact match search using the criteria specified in the object bean. Only records that exactly match the search criteria are returned in the search results object.

Syntax

`Search`*`ObjectName`*`Result searchExact(`*`ObjectName`*`Bean searchCriteria)`

Parameters

Name	Type	Description
searchCriteria	*ObjectName*Bean	The search criteria for the exact match search.

Returns

The results of the search stored in a Search*ObjectName*Result object.

Throws

- `RemoteException`
- `ProcessingException`
- `UserException`

searchPhonetic

Description

This method performs a search using phonetic values for some of the criteria specified in the object bean. This type of search allows for typographical errors and misspellings.

Syntax

`Search`*`ObjectName`*`Result searchPhonetic(`*`ObjectName`*`Bean searchCriteria)`

Parameters

Name	Type	Description
searchCriteria	*ObjectName*Bean	The search criteria for the phonetic search.

Returns

The results of the search.

Throws

- `RemoteException`
- `ProcessingException`
- `UserException`

transferSystemRecord

Description

This method transfers a system record from one enterprise record to another enterprise record.

Syntax

```
void transferSystemRecord(String toEUID, String systemCode, String localID)
```

Parameters

Name	Type	Description
toEUID	String	The EUID of the enterprise record to which the system record will be transferred.
systemCode	String	The processing code of the system record to transfer.
localID	String	The local ID of the system record to transfer.

Returns

None.

Throws

- `RemoteException`
- `ProcessingException`
- `UserException`

updateEnterpriseRecord

Description

This method updates the fields in an existing enterprise object with the values specified in the fields the enterprise object passed in as a parameter. When updating an enterprise object, attempting to change a field that is not updateable will cause an exception. This method does not update the SBR; the survivor calculator updates the SBR once the changes are made to the associated system records.

Syntax

```
void updateEnterpriseRecord(Enterprise ObjectName enterpriseObject)
```

Parameters

Name	Type	Description
enterpriseObject	Enterprise*ObjectName*	The enterprise object containing the values that will update the existing enterprise object.

Returns

None.

Throws

- `RemoteException`
- `ProcessingException`
- `UserException`

updateSystemRecord

Description

This method updates the existing system object in the database with the given system object.

Syntax

```
void updateSystemRecord(System ObjectName systemObject)
```

Parameters

Name	Type	Description
systemObject	System*ObjectName*	The system object to be updated to the enterprise object. **Note –** In the method OTD, "Object" in the parameter name is changed to the name of the parent object. For example, if the parent object is "Person", the name of this parameter will appear as "systemPerson".

Returns

None.

Throws

- `RemoteException`
- `ProcessingException`
- `UserException`

Dynamic Business Process Methods (Repository)

A set of Java methods are created in the master index application for use in Business Processes. These methods include a subset of the dynamic OTD methods, which are documented above. Many of these methods return objects of the dynamic object type or they use these objects as parameters. In the descriptions for these methods, *ObjectName* indicates the name of the parent object.

The following methods are available for Business Processes. They are described in the previous section, "Dynamic Master Index OTD Methods (Repository)" on page 81.

Master Index Helper Classes (Repository)

Helper classes include objects that can be passed as parameters to an OTD method or a Business Process method. They also include the methods that you can access through the system*ObjectName* variable in client Collaborations or Business Processes (where *ObjectName* is the name of a parent object. The helper classes include:

System*ObjectName* Master Index Class (Repository)

In order to run `executeMatch` or `executeMatchUpdate` in a Java Collaboration or Business Process, you must define a variable of the class type System*ObjectName*, where *ObjectName* is the name of a parent object. This class is passed as a parameter to the execute match methods. The class contains a constructor method and several get and set methods for system fields. It also includes one field that specifies the value of the clear field character (for more information, see "ClearFieldIndicator Field" on page 102). In the methods described in this section, *ObjectName* indicates the name of the parent object, *Child* indicates the name of a child object, and *Field* indicates the name of a field defined for the parent object.

Definition

`class SystemObjectName`

Fields

Methods

- "getClearFieldIndicator" on page
- "get*Field*" on page 103
- "get*ObjectName*" on page 104
- "set*Field*" on page 105
- "set*ObjectName*" on page 106

Inherited Methods

The following methods are inherited from `java.lang.Object`.

- equals
- hashcode
- notify
- notifyAll
- toString
- wait()
- wait(long arg)
- wait(long timeout, int nanos)

ClearFieldIndicator Field

The `ClearFieldIndicator` field allows you to specify whether to treat a field in the parent object as null when performing an update from an external system. When an update is performed in the master index application, empty fields typically do not overwrite the value of an existing field. You can specify to nullify a field that already has an existing value in the master index application by entering an indicator in that field. This indicator is specified by the `ClearFieldIndicator` field. By default, the `ClearFieldIndicator` field is set to double-quotes (""), so if a field is set to double-quotes, that field will be blanked out. If you do not want to use this feature, set the clear field indicator to null.

System*ObjectName*

Description

This method is the user-defined system class for the parent object. You can instantiate this class to create a new instance of the system class.

Syntax

`new System`*`ObjectName`*`()`

Parameters

None.

Returns

An instance of the System*ObjectName* class.

Throws

`ObjectException`

getClearFieldIndicator

Description

This method retrieves the value of the `ClearFieldIndicator` field.

Syntax

```
Object getClearFieldIndicator()
```

Parameters

None.

Returns

An object containing the value of the `ClearFieldIndicator` field.

Throws

None.

get*Field*

Description

This method retrieves the value of the specified system field. There are getter methods for the following fields: LocalId, SystemCode, Status, CreateDateTime, CreateFunction, and CreateUser.

Syntax

```
String get Field()
```

or

```
Date get Field()
```

Parameters

None.

Returns

The value of the specified field. The type of value returned depends on the field from which the value was retrieved.

Throws

`ObjectException`

get*ObjectName*

Description

This method retrieves the parent object Java Bean for the system record.

Syntax

ObjectName`Bean get`*ObjectName*`()`

Parameters

None.

Returns

A Java Bean containing the parent object.

Throws

None.

setClearFieldIndicator

Description

This method sets the value of the clear field character (in the `ClearFieldIndicator` field). By default, this is set to double quotes (“”).

Syntax

```
void setClearFieldIndicator(String value)
```

Parameters

Name	Type	Description
value	String	The value that should be entered into a field to indicate that any existing values should be replaced with null.

Returns

None.

Throws

None.

set*Field*

Description

This method sets the value of the specified system field. There are setter methods for the following fields: LocalId, SystemCode, Status, CreateDateTime, CreateFunction, and CreateUser.

Syntax

```
void set<i>Field</i>(value)
```

Parameters

Name	Type	Description
value	varies	The value to set in the specified field. The type of value depends on the field into which the value is being set.

Returns

None.

Throws

`ObjectException`

set*ObjectName*

Description

This method sets the parent object Java Bean for the system record.

Syntax

`void set`*`ObjectName`*`(`*`ObjectName`*`Bean object)`

Parameters

Name	Type	Description
object	*ObjectName*Bean	The Java Bean for the parent object.

Returns

None.

Throws

`ObjectException`

Master Index Parent Beans (Repository)

A Java Bean is created to represent each parent object defined in the object definition of the master index application. The methods in these classes provide the ability to create a parent object Bean and to set or retrieve the field values for that object Bean.

The name of each parent object Bean class is the same as the name of each parent object, with the word "Bean" appended. For example, if a parent object in your object structure is "Person", the name of the associated parent Bean class is "PersonBean". The methods in this class include a constructor method for the parent object Bean, and get and set methods for each field defined for the parent object. Most methods are named based on the name of the parent object and the fields and child objects defined for that object. In the methods described in this section, *ObjectName* indicates the name of the parent object, *Child* indicates the name of a child object, and *Field* indicates the name of a field defined for the parent object.

Definition

`final class` *`ObjectName`*`Bean`

Methods

Inherited Methods

The following methods are inherited from `java.lang.Object`.

- equals
- hashcode
- notify
- notifyAll
- toString
- wait()
- wait(long arg)
- wait(long timeout, int nanos)

*ObjectName*Bean

Description

This method is the user-defined object Bean class. You can instantiate this class to create a new instance of the parent object Bean class.

Syntax

`new` *ObjectName*`Bean()`

Parameters

None.

Returns

An instance of the parent object Bean.

Throws

`ObjectException`

count*Child*

Description

This method returns the total number of child objects contained in a system object. The type of child object is specified by the method name (such as Phone or Address).

Syntax

`int count`*`Child`*`()`

Parameters

None.

Returns

An integer indicating the number of child objects in a collection.

Throws

None.

countChildren

Description

This method returns a count of the total number of child objects belonging to a system object.

Syntax

`int countChildren()`

Parameters

None.

Returns

An integer representing the total number of child objects.

Throws

None.

countChildren

Description

This method returns a count of the total number of child objects of a specific type that belong to a system object.

Syntax

```
int countChildren(String type)
```

Parameters

Name	Type	Description
type	String	The type of child object to count, such as Phone or Address.

Returns

An integer representing the total number of child objects of the specified type.

Throws

None.

delete*Child*

Description

This method removes the specified child object from the system object. The type of child object to remove is specified by the name of the method, and the specific child object to remove is specified by its unique identification code assigned by the master index application.

Syntax

```
void delete<Child>(String <Child>Id)
```

Parameters

Name	Type	Description
*Child*Id	String	The unique identification code of the child object to delete.

Returns

None.

Throws

`ObjectException`

get*Child*

Description

This method retrieves an array of child object Beans. Each getter method is named according to the child objects defined for the parent object. For example, if the parent object contains a child object named "Address", the getter method for this field is named `getAddress`. A getter method is created for each child object in the parent object.

Syntax

Child`Bean[] get`*Child*`()`

Parameters

None.

Returns

An array of Java Beans containing the type of child objects specified by the method name.

Throws

None.

get*Child*

Description

This method retrieves a child object Bean based on its index in a list of child objects. Each getter method is named according to the child objects defined for the parent object. For example, if the parent object contains a child object named "Address", the getter method for this field is named `getAddress`. A getter method is created for each child object in the parent object.

Syntax

Child`Bean get`*Child*`(int i)`

Parameters

Name	Type	Description
i	int	The index of the child object to retrieve from a list of child objects.

Returns

A Java Bean containing the child object specified by the index value. The method name indicates the type of child object returned.

Throws

`ObjectException`

get*Field*

Description

This method retrieves the value of the field specified in the method name. Each getter method is named according to the fields defined for the parent object. For example, if the parent object contains a field named "FirstName", the getter method for this field is named `getFirstName`.

Syntax

`String get`*Field*`()`

Note – The syntax for the getter methods depends on the type of data specified for the field in the object structure. For example, the getter method for a date field would have the following syntax:

```
Date getField
```

Parameters

None.

Returns

The value of the specified field. The type of data returned depends on the data type defined in the object definition.

Throws

`ObjectException`

get*ObjectName***Id**

Description

This method retrieves the unique identification code (primary key) of the object, as assigned by the master index application.

Syntax

`String get`*`ObjectName`*`Id()`

Parameters

None.

Returns

A string containing the unique ID of the parent object.

Throws

`ObjectException`

set*Child*

Description

This method adds a child object to the system object.

Syntax

```
void set<Child>(int index, <Child>Bean child)
```

Parameters

Name	Type	Description
index	integer	The index number for the new child object.
child	*Child*Bean	The Java Bean containing the child object to add.

Returns

None.

Throws

None.

set*Child*

Description

This method adds an array of child objects of one type to the system object.

Syntax

```
void set<Child>(<Child>Bean[] children)
```

Parameters

Name	Type	Description
children	*Child*Bean[]	The array of child objects to add.

Returns

None.

Throws

None.

set*Field*

Description

This method sets the value of the field specified in the method name. Each setter method is named according to the fields defined for the parent object. For example, if the parent object contains a field named "DateOfBirth", the setter method for this field is named `setDateOfBirth`. A setter method is created for each field in the parent object, including any fields containing standardized or phonetic data.

Syntax

```
void set<i>Field</i>(value)
```

Parameters

Name	Type	Description
value	varies	The value of the field specified by the method name. The type of value depends on the field being populated.

Returns

None.

Throws

`ObjectException`

set*ObjectName***Id**

Description

This method sets the value of the *ObjectName*Id field in the parent object.

Note – This ID is set internally by the master index application. Do not set this field manually.

Syntax

```
void set<i>ObjectName</i>Id(String value)
```

Parameters

Name	Type	Description
value	String	The value of the *ObjectName*Id field.

Returns

None.

Throws

`ObjectException`

Master Index Child Beans (Repository)

A Java Bean is created to represent each child object defined in the object definition of the master index application. The methods in these classes provide the ability to create a child object Bean and to set or retrieve the field values for that object Bean.

The name of each child object Bean class is the same as the name of each child object, with the word "Bean" appended. For example, if a child object in your object structure is named "Address", the name of the corresponding child class is `AddressBean`. The methods in this class include a constructor method for the child object Bean, and get and set methods for each field defined for the child object. Most methods have dynamic names based on the name of the child object and the fields defined for that object. In the following methods, *Child* indicates the name of a child object and *Field* indicates the name of a field defined for the child object.

Definition

`final class` *Child*`Bean`

Methods

Inherited Methods

The following methods are inherited from `java.lang.Object`.

- equals
- hashcode
- notify
- notifyAll
- toString
- wait()
- wait(long arg)
- wait(long timeout, int nanos)

*Child*Bean

Description

This method is the user-defined object Bean class. You can instantiate this class to create a new instance of the child object Bean class.

Syntax

`new` *Child*`Bean()`

Parameters

None.

Returns

An instance of the child object Bean.

Throws

`ObjectException`

delete

Description

This method removes the child object from the object being processed. This is used with the `executeMatchUpdate` function to update a system object by deleting one of the child objects from the object being processed.

Syntax

```
void delete()
```

Parameters

None.

Returns

None.

Throws

`ObjectException`

get*Field*

Description

This method retrieves the value of the field specified in the method name. Each getter method is named according to the fields defined for the child object. For example, if the child object contains a field named "ZipCode", the getter method for this field is named `getZipCode`.

Syntax

`String get`*`Field`*`()`

Note – The syntax for the getter methods depends on the type of data specified for the field in the object structure. For example, the getter method for a date field would have the following syntax:

```
Date getField
```

Parameters

None.

Returns

The value of the specified field. The type of data returned depends on the data type defined in the object definition.

Throws

`ObjectException`

get*Child***Id**

Description

This method retrieves the unique identification code (primary key) of the object, as assigned by the master index application.

Syntax

`String get`*`Child`*`Id()`

Parameters

None.

Returns

A string containing the unique ID of the child object.

Throws

`ObjectException`

set*Field*

Description

This method sets the value of the field specified in the method name. Each setter method is named according to the fields defined for the child object. For example, if the child object contains a field named "Address", the setter method for this field is named `setAddress`. A setter method is created for each field in the child object, including any fields containing standardized or phonetic data.

Syntax

`void set`*`Field`*`(value)`

Parameters

Name	Type	Description
value	varies	The value of the field specified by the method name. The type of value depends on the data type of the field being populated.

Returns

None.

Throws

`ObjectException`

set*Child*Id

Description

This method sets the value of the *Child*Id field in the child object.

Note – This ID is set internally by the master index application. Do not set this field manually.

Syntax

`void set`*`Child`*`Id(String value)`

Parameters

Name	Type	Description
value	String	The value of the *Child*Id field.

Returns

None.

Throws

`ObjectException`

DestinationEO Master Index Class (Repository)

This class represents an enterprise object involved in a merge. This is the enterprise object whose EUID was kept in the final merge result record. A DestinationEO object is used when unmerging two enterprise objects.

Definition

```
class DestinationEO
```

Methods

getEnterprise*ObjectName*

Description

This method retrieves the surviving enterprise object from a merge transaction in order to allow the records to be unmerged.

Syntax

`Enterprise`*`ObjectName`* `getEnterprise`*`ObjectName`*`()`

where *ObjectName* is the name of the parent object.

Parameters

None.

Returns

The surviving enterprise object from a merge transaction.

Throws

`ObjectException`

Search*ObjectName*Result Master Index Class (Repository)

This class represents the results of a search. A Search*ObjectName*Result object (where *ObjectName* is the name of the parent object) is returned as a result of a call to "searchBlock" on page 96, "searchExact" on page 97, or "searchPhonetic" on page 97."searchBlock" on page 96

Definition

```
class SearchObjectNameResult
```

Methods

getEUID

Description

This method retrieves the EUID of a search result record.

Syntax

```
String getEUID()
```

Parameters

None.

Returns

A string containing an EUID.

Throws

None.

getComparisonScore

Description

This method retrieves the weight that indicates how closely a search result record matched the search criteria.

Syntax

```
Float getComparisonScore()
```

Parameters

None.

Returns

A comparison weight.

Throws

None.

get*ObjectName*

Description

This method retrieves an object bean for a search result record.

Syntax

ObjectName`Bean get`*ObjectName*`()`

where *ObjectName* is the name of the parent object.

Parameters

None.

Returns

An object bean.

Throws

None.

SourceEO Master Index Class (Repository)

This class represents an enterprise object involved in a merge. This is the enterprise object whose EUID was not kept in the final merge result record. A SourceEO object is used when unmerging two enterprise objects.

Definition

```
class SourceEO
```

Methods

getEnterprise*ObjectName*

Description

This method retrieves the non-surviving enterprise object from a merge transaction in order to allow the records to be unmerged.

Syntax

`Enterprise`*`ObjectName`* `getEnterprise`*`ObjectName`*`()`

where *ObjectName* is the name of the parent object.

Parameters

None.

Returns

The non-surviving enterprise object from a merge transaction.

Throws

None.

System*ObjectName*PK Master Index Class (Repository)

This class represents the primary keys in a system object, which include the processing code for the originating system and the local ID of the object in that system. The class is named for the primary object. For example, if the primary object is named "Person", this class is named

SystemPersonPK. If the primary object is named "Company", this class is named SystemCompanyPK. The methods in these classes provide the ability to create an instance of the class and to retrieve the system processing code and the local ID.

Definition

class System*ObjectName*PK

where *ObjectName* is the name of the parent object.

Methods

System*ObjectName*PK

Description

This method is the user-defined system primary key object. This object contains a system code and a local ID. Use this constructor method to create a new instance of a system primary key object.

Syntax

new System*ObjectName*PK()

where *ObjectName* is the name of the parent object.

Parameters

None.

Returns

An instance of the system primary key object.

Throws

None.

getLocalId

Description

This method retrieves the local identifier from a system primary key object.

Syntax

```
String getLocalId()
```

Parameters

None.

Returns

A string containing a local identifier.

Throws

None.

getSystemCode

Description

This method retrieves the system's processing code from a system primary key object.

Syntax

```
String getSystemCode()
```

Parameters

None.

Returns

A string containing the processing code for a system.

Throws

None.

Master Index Match Types and Field Names (Repository)

You can select a Match Type for each field defined in the Sun Master Index wizard. Each match type defines a different type of standardization, normalization, phonetic encoding, and matching logic in the Match Field file. The following topics describe each match type and how each affects the logic in the Match Field file.

Master Index Match and Standardization Types (Repository)

For each field that will be used for matching in the master index application, you can select a match type in the wizard. When you select a match type for a field, Sun Master Index automatically adds that field to the match string in the Match Field file and, in many cases, generates additional fields in the object definition that are not visible on the wizard. These fields are used for searching and matching and they should not be modified.

If new fields are generated, they are automatically incorporated into the configuration files and the database script that creates the master index tables. These fields store standardized, normalized, or phonetic versions of the field, depending on the type of matching you choose. In addition, these fields are assigned a match type in the match string in the Match Field file. They might also be defined for standardization in the Match Field file, in which case they will also be assigned a standardization type.

Note – The match types specified in the Match Field file for the fields in the match string are not always the same as the match types you specify in the wizard. Information about match types is provided in the following sections. For more information, see *Understanding the Sun Match Engine*.

Sun Match Engine Match Types (Repository)

The Sun Master Index wizard match types fall into four primary categories.

The actual standardization and match types entered into the Match Field file vary for each match type you select in the wizard. The match and standardization types for each type of field

are listed in the following descriptions. The match types entered into the Match Field file correspond to the match types defined in the match configuration file, `MatchConfigFile.cfg`.

Person Match Types

The Person match types include PersonLastName and PersonFirstName. These match types are used to normalize and phonetically encode name fields for person matching. For each field with one of these match types, the wizard adds two fields to the object structure for phonetic and standardized versions. If you specify a field with a person match type for blocking in the wizard, the phonetic version of the name is automatically added to the blocking query. The following fields are created when you specify one of the Person match types for a field (*field_name* refers to the name of the field specified for Person matching).

- *field_name*_Std – This field contains the normalized version of the name.
- *field_name*_Phon – This field contains the phonetic version of the name.

The corresponding standardization and match types in the Match Field file are listed in Table 24.

TABLE 24 Person Name Standardization and Match Types

eView Wizard Match Type	Match Field File Standardization Type	Match Field File Match Type
PersonLastName	PersonName	LastName
PersonFirstName	PersonName	FirstName

BusinessName Match Types

The BusinessName match type is designed to help parse, normalize, and phonetically encode a business name. BusinessName matching adds several fields to the object structure and to the match string. If you specify a business name field for blocking, each parsed business name field is added to the blocking query. The corresponding standardization type in the Match Field file for all fields selected for BusinessName matching is also BusinessName. The actual match type assigned to each field varies depending on the type of information in each field.

Table 25 lists the fields created when you select the BusinessName match type for a field along with their corresponding match types in the Match Field file (*field_name* refers to the name of the field selected for BusinessName matching).

Note – Only specify this type of matching for one business name field; otherwise, the wizard will create duplicate entries in the object structure. If more than one field contains the business name, you can add those fields to the standardization structure in the Match Field file after the wizard creates the configuration files.

TABLE 25 BusinessName Match Types

Field Name	Description	Added to the Match String?	Match Field File Match Type
*field_name*_Name	The parsed and normalized version of the business name.	Yes	PrimaryName
*field_name*_NamePhon	The phonetic version of the business name.	No	
*field_name*_OrgType	The parsed organization type of the business name.	Yes	OrgTypeKeyword
*field_name*_AssocType	The association type for the business.	Yes	AssocTypeKeyword
*field_name*_Industry	The name of the industry for the business.	Yes	IndustryTypeKeyword
*field_name*_Sector	The name of the industry sector (industries are a subset of sectors).	Yes	IndustrySectorList
*field_name*_Alias	An alias for the business name.	No	
*field_name*_Url	The business' web site URL.	Yes	Url

Address Match Types

The Address match type is designed to help parse, normalize, and phonetically encode an address for matching or standardizing address information. Address matching adds several fields to the object structure and to the match string. If you specify an address field for blocking, the parsed fields are added to the blocking query. The corresponding standardization type for fields selected for Address matching is Address. The actual match type assigned to each field varies depending on the type of information in each field.

The fields created when you select the Address match type for a field are listed below along with their corresponding match types in the Match Field file (*field_name* refers to the name of the field selected for Address matching).

Note – Only specify this type of matching for one street address field; otherwise, the wizard will create duplicate entries in the object structure. If more than one field contains the street address, you can define the additional fields in the standardization structure in the Match Field file after the wizard creates the configuration files.

TABLE 26 Address Match Types

Field Name	Description	Added to Match String?	Match Field File Match Type
*field_name*_HouseNo	The parsed street number of the address.	Yes	HouseNumber
*field_name*_StDir	The parsed and normalized street direction of the address.	Yes	StreetDir
*field_name*_StName	The parsed and normalized street name of the address.	Yes	StreetName
*field_name*_StPhon	The phonetic version of the street name.	No	
*field_name*_StType	The parsed and normalized street type of the address, such as Boulevard, Street, Drive, and so on.	Yes	StreetType

If you want to search on street addresses but do not want to use these fields for matching, select the Address match type for only one street address field in the wizard. When the wizard is complete, you can remove the address fields from the match string in the Match Field file.

Miscellaneous Match Types

Several additional match types are defined in the wizard for the Sun Match Engine. These match types are used to indicate matching on a string, date, or number fields other than those described above or to indicate matching on a field that contains a single character (such as the gender field, which might accept "F" for female or "M" for male). These match types do not define standardization for the specified field and do not add any fields to the object structure. If you specify one of these match types for a field in the wizard, the field is added to the match string with a match type of String, Date, Number, or Char.