How to Write Data Upgrade Scripts for Microsoft Dynamics AX 2012

Microsoft Corporation

Published: March 2014

Table of Contents

Introduction

This document describes how to use the Microsoft Dynamics™ AX Data Upgrade Framework and how to write data upgrade scripts for customer data upgrade data models (Microsoft Dynamics AX tables). The data upgrade framework can be used to perform data correction or data transformation.

The intended audience for this document is developers. If you have modified the schema of a previous version of Microsoft Dynamics AX, then you must use the upgrade framework to write a corresponding upgrade script to update the data.

This document is based on the Microsoft Dynamics AX 2012 Data Upgrade Framework. It has been updated regarding the new data upgrade framework and best practices for performance.

Microsoft Dynamics AX 2012 supports upgrading data in the following ways:

- Microsoft Dynamics AX 4.0 to Microsoft Dynamics AX 2012
- Microsoft Dynamics AX 2009 to Microsoft Dynamics AX 2012

Terms and Abbreviations

The following table provides terms and definitions that relate to the data upgrade process.

Term/Abbreviation	**Definition**
Source environment or source system	The Microsoft Dynamics AX system which is being upgraded. The supported source systems for direct upgrade to Microsoft Dynamics AX 2012 are: • Microsoft Dynamics AX 4.0 • Microsoft Dynamics AX 2009 The upgrade starts while the source environment is still live, but there is minimal impact to the live system.
Transformation	Data is transformed between source and target environment by using field mapping and joins between necessary tables. Examples include the Address table normalization and the Dimension table normalization.
Source affected table	Tables and fields within source environment that have to be updated with transformations. Examples are Dimension fields and Address fields. No update or changes are made to the source table directly, changes are made to shadow tables.
Source non-affected table	Tables that have to be copied to the target system as they are with the exception of mapping transformation.
Shadow table	Auxiliary table created on the source environment. The shadow tables will contain all fields from the source tables which have to be updated.
Dictionary table	New target Microsoft Dynamics AX 2012 tables. These tables will be

	imported into the source environment for the application pre-upgrade checklist, and upgrade script execution where needed. The definition of the table must match the target system, the dictionary tables will be copied to the target as they are.
Target affected table	Target tables affected by transformations. The table columns will be partially copied from the source tables, and partially from the shadow tables.
Target non-affected table	Tables copied unchanged from the source database. These tables already have the Microsoft Dynamics AX 2012 schema. Mapping between Microsoft Dynamics AX 4.0, Microsoft Dynamics AX 2009, and Microsoft Dynamics AX 2012 schemas must be created on copy (similar to SYNC).
Preprocess upgrade script	Preprocess upgrade script executed by the upgrade framework for a particular table in the source environment based on the template provided by an application team.
Preprocess upgrade script template	Template of a preprocess script created by an application team and registered with the upgrade framework API. Template provides the following for the upgrade framework: • Table and fields that will be involved in the application pre-upgrade checklist. • Business logic for the source data that occurs for the application pre-upgrade checklist.
Application pre-upgrade checklist tasks	Application pre-upgrade checklist tasks that require user intervention to clean-up before the upgrade, for example Address normalization. Many of the addresses are duplicate. As a result, the application pre-upgrade checklist form will provide users with the ability to decide how the master address entity should look.
Delta upgrade script or delta job	Upgrade scripts implemented by application teams to find the changes within table records since the last run of the related preprocessing upgrade scripts.
Single-user mode	No active user transaction is running on the source system. Logic will check if only one administrator user is connected to the source Microsoft Dynamics AX system.
Exception tables	System tables that are specific to the installation that will be excluded from the copy operation. For example, License tables.
Validation script	Special upgrade scripts defined to identify issues with upgrade or data upgrade. These kinds of upgrade scripts should read data from live Microsoft Dynamics AX tables and display messages to take corrective action as needed.
Upgrade throttling	Method used to pause scripts, resume scripts, or change the amount of resources assigned to execute a script.

Throttling APIs	APIs provided by the upgrade framework that can be used by upgrade scripts to support pause and resume functionality.

How to Upgrade Data for a Major Release or Service Pack

The data upgrade framework drives the data upgrade scripts that transform an older version of the Microsoft Dynamics AX database to the new version. These steps are described in later sections.

The upgrade process consists of two parts:

- Preprocess in the source environment
- Upgrade in the target environment

The following table describes the scripts that run on the source environment:

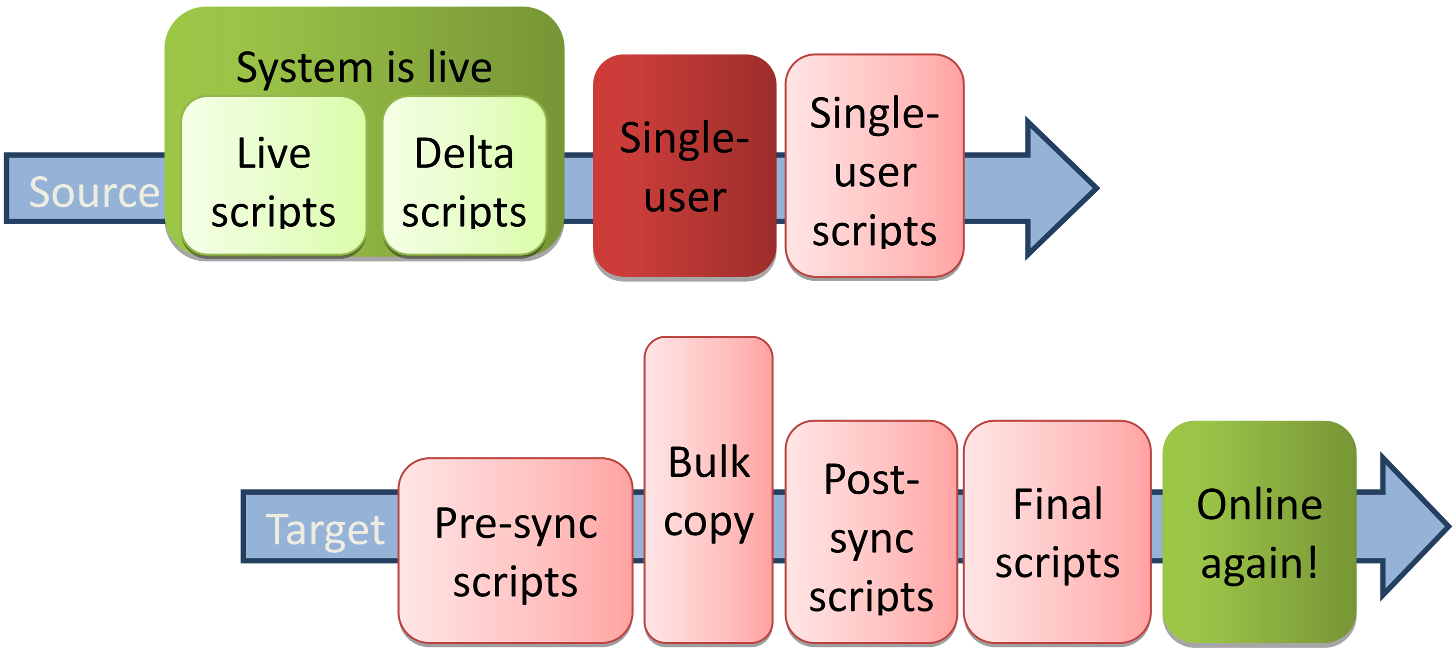
Script type	**Description**
Live	Preprocessing script that is implemented to run on a live source system. The Microsoft Dynamics AX system is available to the user and the upgrade occurs in the background. Live preprocessing scripts write prepared data into the shadow and dictionary tables that were created by importing the preprocessing XPOs. These scripts must be implemented by using row-by-row operations, and commit after a predetermined number of transactions. To schedule live preprocessing scripts, in the preprocessing upgrade checklist, click Run live preprocessing scripts. The scripts that run are specified in the initTransformationJobs method with a definition of SetLivePreProcessingScript in ReleaseUpdateTransformDB40_ and ReleaseUpdateTransformDB50_ classes.
Delta	Preprocessing scripts are implemented to check for changes to the production data that have occurred since you started running the live preprocessing scripts. Similar to the live preprocessing scripts, the delta scripts processes the updated data into the shadow and dictionary tables. Delta scripts are designed to run multiple times before entering into the single-user mode. To schedule Delta preprocessing scripts, in the preprocessing upgrade checklist, click Run delta preprocessing scripts. The scripts that run are specified in the initTransformationJobs method with a definition of SetDeltaPreProcessingScript in the ReleaseUpdateTransformDB40_ and ReleaseUpdateTransformDB50_ classes.
Single-user	Preprocessing scripts are implemented to run on the source system during single-user mode. Single-user mode occurs on the source system to make sure that the system is unavailable for business use. In single-user mode, only an upgrade user who has administrative permissions is connected. No other users can start a client session when the source system is in final preprocessing and source data is being bulk copied to the Microsoft Dynamics AX 2012 target system. Single-user preprocessing scripts must be written as set-based operations. To schedule single-user preprocessing scripts, in the preprocessing upgrade checklist, click Run single-user mode preprocessing scripts. The scripts that run are specified in the initTransformationJobs method with a definition of SetSingleuserPreProcessingScript in ReleaseUpdateTransformDB40_ and ReleaseUpdateTransformDB50_ classes.
Upgrade	Help to identify the issues that cause an upgrade to fail. These scripts read AOT

Readiness	metadata or data and then categorize the results into an error or advisory in the upgrade validation result. Errors must be fixed before you continue with the upgrade. Advisory are designed to improve the upgrade experience by pointing out issues which may improve the efficiency of upgrade, or issues which require manual actions. To schedule upgrade readiness scripts, in the preprocessing upgrade checklist, click on the Upgrade Readiness check. The scripts that run are specified in the initTransformationJobs method with a definition of SetValidatePreProcessingScript in ReleaseUpdateTransformDB40_ and ReleaseUpdateTransformDB50_ classes.

The following table describes the scripts that run on the target environment:

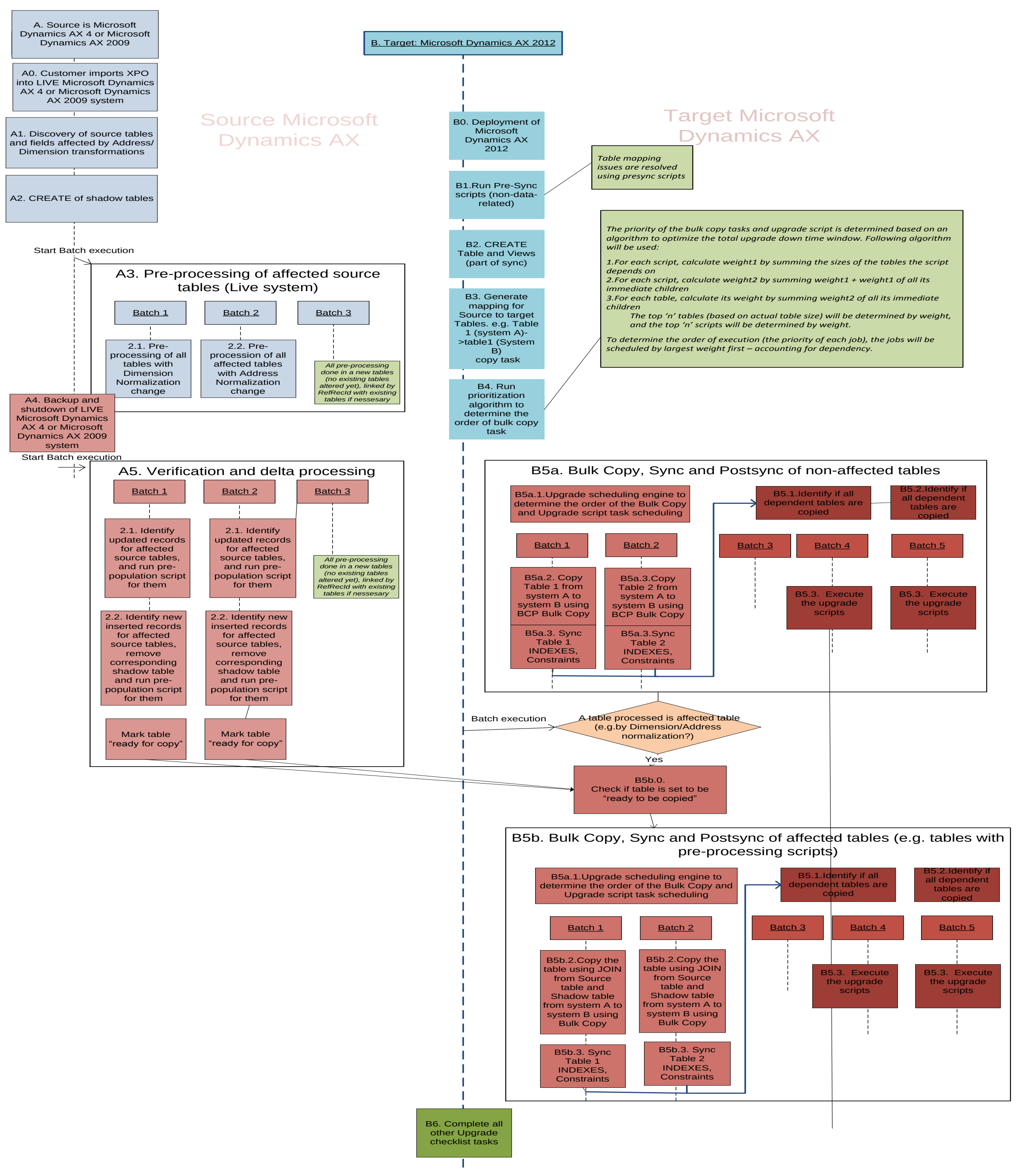
Script	**Description**
Presynchronize	Make metadata changes before the step to create tables. For example, presynchronize scripts are used to map the database table or fields on the source Microsoft Dynamics AX system to the database table or fields on the target system. This table-to-table and column-to-column mapping prepares the source-to-target upgrade model for data copy from source to target. Other kinds of presynchronize scripts disable unique index in preparation for data upgrade and avoid duplicate key error. This action is undone during the postsynchronize step. The scripts that run are methods with an UpgradeScriptStageAttribute of PreSync. They are specified in the in ReleaseUpdateDB401_, ReleaseUpdateDB41_ and ReleaseUpdateDB60_ classes.
Postsync	Contain the bulk of the data upgrade. Company-specific business data is upgraded in postsynchronization upgrade scripts. Postsynchronization also reverses the metadata changes that were made during the presynchronize step.
Final	Implemented to run outside the core data upgrade. Thus reduces the upgrade downtime for the core data upgrade. An example is upgrading metadata that is related to AIF endpoints. The scripts that run are methods with an UpgradeScriptStageAttribute of Additional, the scripts are specified in the ReleaseUpdateDB401_, ReleaseUpdateDB41_ and ReleaseUpdateDB60_ classes.

The following diagram illustrates a simplified view of the data upgrade process:



The following diagram is a sequence diagram of the data upgrade process that provides detail on when each process occurs.

In the diagram, the sequence is specified with a alphanumeric value. For example, the events that are labeled A, A1, A2 occur first, then B, B1, B2 and so on.


A. Source is Microsoft Dynamics AX 4 or Microsoft Dynamics AX 2009
A0. Customer imports XPO into LIVE Microsoft Dynamics AX 4 or Microsoft Dynamics AX 2009 system
A1. Discovery of source tables and fields affected by Address/ Dimension transformations
A2. CREATE of shadow tables
Start Batch execution
A3. Pre-processing of affected source tables (Live system)
Batch 1
Batch 2
Batch 3
2.1. Pre-processing of all tables with Dimension Normalization change
2.2. Pre-procession of all affected tables with Address Normalization change
All pre-processing done in a new tables (no existing tables altered yet), linked by RefRecId with existing tables if nessesary
A4. Backup and shutdown of LIVE Microsoft Dynamics AX 4 or Microsoft Dynamics AX 2009 system
Start Batch execution
A5. Verification and delta processing
Batch 1
Batch 2
Batch 3
2.1. Identify updated records for affected source tables, and run pre-population script for them
2.1. Identify updated records for affected source tables, and run pre-population script for them
All pre-processing done in a new tables (no existing tables altered yet), linked by RefRecId with existing tables if nessesary
2.2. Identify new inserted records for affected source tables, remove corresponding shadow table and run pre-population script for them
2.2. Identify new inserted records for affected source tables, remove corresponding shadow table and run pre-population script for them
Mark table "ready for copy"
Mark table "ready for copy"
Source Microsoft Dynamics AX
B. Target: Microsoft Dynamics AX 2012
Target Microsoft Dynamics AX
B0. Deployment of Microsoft Dynamics AX 2012
B1.Run Pre-Sync scripts (non-data-related)
Table mapping issues are resolved using presync scripts
B2. CREATE Table and Views (part of sync)
B3. Generate mapping for Source to target Tables. e.g. Table 1 (system A)->table1 (System B) copy task
B4. Run prioritization algorithm to determine the order of bulk copy task
B5a. Bulk Copy, Sync and Postsync of non-affected tables
B5a.1.Upgrade scheduling engine to determine the order of the Bulk Copy and Upgrade script task scheduling
B5.1.Identify if all dependent tables are copied
B5.2.Identify if all dependent tables are copied
Batch 1
Batch 2
Batch 3
Batch 4
Batch 5
B5a.2. Copy Table 1 from system A to system B using BCP Bulk Copy
B5a.3.Copy Table 2 from system A to system B using BCP Bulk Copy
B5.3. Execute the upgrade scripts
B5.3. Execute the upgrade scripts
B5a.3. Sync Table 1 INDEXES, Constraints
B5a.3.Sync Table 2 INDEXES, Constraints
Batch execution
A table processed is affected table (e.g.by Dimension/Address normalization?)
Yes
B5b.0. Check if table is set to be "ready to be copied"
B5b. Bulk Copy, Sync and Postsync of affected tables (e.g. tables with pre-processing scripts)
B5a.1.Upgrade scheduling engine to determine the order of the Bulk Copy and Upgrade script task scheduling
B5.1.Identify if all dependent tables are copied
B5.2.Identify if all dependent tables are copied
Batch 1
Batch 2
Batch 3
Batch 4
Batch 5
B5b.2.Copy the table using JOIN from Source table and Shadow table from system A to system B using Bulk Copy
B5b.2.Copy the table using JOIN from Source table and Shadow table from system A to system B using Bulk Copy
B5.3. Execute the upgrade scripts
B5.3. Execute the upgrade scripts
B5b.3. Sync Table 1 INDEXES, Constraints
B5b.3. Sync Table 2 INDEXES, Constraints
B6. Complete all other Upgrade checklist tasks

The priority of the bulk copy tasks and upgrade script is determined based on an algorithm to optimize the total upgrade down time window. Following algorithm will be used:

1.For each script, calculate weight1 by summing the sizes of the tables the script depends on
2.For each script, calculate weight2 by summing weight1 + weight1 of all its immediate children
3.For each table, calculate its weight by summing weight2 of all its immediate children
The top 'n' tables (based on actual table size) will be determined by weight, and the top 'n' scripts will be determined by weight.

To determine the order of execution (the priority of each job), the jobs will be scheduled by largest weight first – accounting for dependency.

Implementation consideration for upgrade scripts

You will be implemeting upgrade scripts under the followng scenarios:

- Upgrading ISV solution: As an ISV developer you are responsible for upgrading your ISV solution. That means you will be using upgrade framework to upgrade data associated with ISV solution.
- Upgrading custom solution: Most Dynamics AX are customized to meet unique customer requirements. These customizations are mostly done by partners or in some case by customers. Data associated with these customizations also needs to be upgraded and requires custom upgrade scripts.

The following list contains questions to ask when you develop upgrade scripts:

- When do I need to write upgrade scripts?
- What types of upgrade script do I need?
- How to implement these scripts?
- What's the best practice to implement these scripts?

When is a Data Upgrade Script Needed?

The following list describes changes that require an upgrade script:

- Change the name of a field, when field ID is different
- Change the name of a table, when table ID is different
- Delete a table and save data
- Delete a field and save data
- Add or change unique indexes
- Change a non-unique index into a unique index
- Restructure where data is stored. For example, move data from one field to another
- Correct old data inconsistencies
- Populate new tables with existing data
- Populate new fields with existing data or a default value that is different from the default value for the data type

When is a Data Upgrade Script Not Needed?

There are changes that can be made in the data model without the need for an upgrade script. The following list describes changes that can be made without an upgrade script:

- Change the name of a field, when field ID is the same
- Change the name of a table, when table ID is the same
- Add a field to a table with a default value for every field
- Add or change relations

- Add or change non-unique indexes
- Add or change delete actions
- Add or change/delete temporary table

What types of upgrade script do I need?

The first question is do I need source side or target side upgrade scripts. If source side scripts – do I need Live & Delta scripts or Single user mode? If Target side scripts – so I need the Presyncronize or Postsynchronize upgrade scripts. There are other questions – when should I implement upgrade readiness scripts or should I conisder adding upgrade scripts as Additional scripts? This section will provide with some guidance which will help you understand the various factors which determines the answer.

How big or complex is the data change?

This is determined mainly by the size of the data (if you are an ISV - think about the worst case scenario) or the complexity of the chnge in terms of the data model changes. For example, in Dyanmics Ax 2012, Addresses are normailied. What used to be in custtable and various transaction tables in previous version has been converted into the group of less than ten normalized tables. In worst case, it means addresses were stored in around fifty or more tables and some of these address were stored in large transaction tables, such as salestable, etc. If we would have taken a route to upgrade this data on target system, then it would lead to a huge downtime due to the amount of data being processed and the complexity of the scripts. Complexity in this case means writing code which will go through all transaction tables and then building a normalized set of Address tables on a Dynamics Ax 2012 system. This means huge upgrade downtime. Therefore, we choose a route of building a Live and delta preprocessing upgrade scripts. There were smaller changes such as upgrading address book, which can be accomplished quickly using a set based operations – and for that we chose the postsynchronization upgrade scripts.

If the changes are complex and the data set which are impacted by these changes are large then you should consider implementing Live preprocessing upgrade scripts. You can have a combination of Live preprocessing, single user scripts and postsync scripts. Postsync scripts are appropriate where changes are staright forward, changes can be easily implemented using the set based operations and the expected impact on the total upgrade downtime is low.

Most important part of any upgrade script development is to minimize the upgrade downtime for the customer. Therefore, it means upgrade scripts design should be implemented with performance in mind. The development cost of implementing the preprocessing scripts are a little higher compared to postsync scripts. Therefore, make sure that you analyze the design carefully while deciding the approach to upgrade script implementation.

In summary, here are scenarios when you may need preprocessing scripts:

- Upgrade does not fit into the upgrade window
- Massive data updates
- Processing big transactional tables

Here are some of the implementation consideration and usage of various types of upgrade scripts.

Upgrade readiness scripts:

Upgrade readiness scripts are implemented to identify the following issues:

- Data quality issues which might fail your upgrade scripts: Upgrade scripts are written based on certain assumption on valid data, like you don't expect a duplicate for your unique index or certain data constraints. If user is not aware of these data issues then upgrade scripts, either Preprocessing scripts or Target side scripts would fail. In many cases it's hard to know immediately why the scripts are failing and needs some debugging. By identifying these conditions upfront, the data upgrade process runs much smoothly, saving on valuable upgrade test cycles. In most cases these scripts should be of type "Error" (upgrade readiness script type) – that means user must fix these issues before proceeding with the upgrade.
- Unsupported scenarios: There may be scenarios which are unsupported. For example, you expect to have the multisite enabled for all company accounts before data upgrade script can run. You may also have scenario where certain functionality has been deprecated or if there is no out of the box support for upgrade. For example, there is no automated reports upgrade (X++ reports) from earlier version to Dynamics AX 2012 and therefore an upgrade advisory type readiness script informs about it to the user. The intent is to make sure customer or partners upgrading their system are aware of all issues which needs to be accounted and planned for upgrade. These scripts can be of type Advisory or Error, depending upon the intent of the scripts. If you don't want the upgrade user to proceed with upgrade without fixing issues then it should be of type "Error". If the intent of the script is to just inform user, like, upgrade of X++ report is not automated, then this can be defined as of upgrade script type "Advisory".
- Best practice or required setup of the system: For running upgrade you may recommend certain performance setup, like, Microsoft shipped an script which ask user to run a script which update database statistics to improve the performance of upgrade or making sure source database has the RCSI (read committed snapshot isolation) setting turned on.
- Upgrade readiness scripts should be designed to run multiple times. These scripts can potentially be run on a Live production system. Note the performance of the script. Script in most cases will only be doing a read from tables and populating the results in the log table.

Live and Delta preprocessing scripts:

Live upgrade scripts are designed to run on a Live Dynamics AX system (Source system) which is being upgraded. These scripts are designed with following goals:

- Do row-by-row operations to minimize impact on the live production system
- No update or changes are ever made to the production data, all changes should happen in staging tables, such as Shadow or dictionary tables.
- "While select" statement should be implemented with performance in mind, for example:
  - Avoid complex joins in while select statements which may lead to database sorting
  - "Where" condition within the statement should be based on primary or unique indexes
  - "Order by" clause within the statement should contain the primary or unique index
  - Design scripts to avoid any unnecessary round-trip to the database

    - Use indexes on shadow table fields which are being used for joining with source table
    - Use cache lookups on shadow tables
    - Use field lists within select statements
- Script should use the upgrade throttling APIs to support pause/resume functionality
- Every Live script should have a corresponding Delta script
    - Each Delta script is designed to identify the changes since the Live preprocessing. For example, a not exist join on RecID and RecVersion column will identify the delta changes.
    - "While Select" best practice for Live scripts are applicable for Delta script too
- Virtual company consideration: Design scripts to support Virtual company or table collection on the tables. Usually, the recommendation is to design the script with an assumption to have the Shadow table for the script be part of the same table collection as the source table (there is an Upgrade readiness script to identify this situation). Supporting virtual company in this context will require implementing a "not exist" condition within the "While select" loop to avoid reprocessing the rows which were processed in the context of other shared company. Otherwise, script will fail with records exist error.
- Configuration key consideration: make user to tie your upgrade scripts to appropriate configuration key to prevent it from running if the corresponding feature is not installed.

Single user scripts:

Single user scripts are designed to be run during the upgrade downtime. Therefore, these scripts should be implemented using the set base operations.

Preprocessing forms:

Preprocessing forms are required when you need user interaction to remove the data ambugity. Here are some scenarios which explains when a preprocessing form should be implemented.

Scenario 1: Application Concept between Microsoft Dynamics Ax version has changed

For example:

Multisite activation: Multisite feature was introduced in Microsoft Dynamics AX 2009 and to upgrade a Microsoft Dynamics Ax 4.0 to Microsoft Dynamics Ax 2012, a site structure needs to be predefined. This predefined site structure is used during upgrade to correctly upgrade to multisite enabled system on Microsoft Dynamics Ax 2012

Global address book: Organization of Addresses in Microsoft Dynamics Ax 2012 are drastically changed. In earlier version addresses were part of the custtable and variou transactional tables which stored the shipped, invoiced, or billed addresses. In Microsoft Dynamics AX 2012 all address are consilidated into standard set of reference data for country name, address format, region, etc. It means if there were three country defined as UK, England, Great Britain – all meaning the same country but in new Microsoft Dynamics AX system it will be a set of ambigous data. A user needs to correctly map these country to correct set of standard country.

Scenario 2: Company based schema is changed to shared tables

In earlier version of Microsoft Dynamics Ax most tables and concepts were bound by DataAreaID, i.e. company construct was associated with each concept within the table structure. In Microsoft Dynamics Ax 2012, most table structure are changed to shared table concept. It means data which were unique within a context of a single company may have duplicates if merged with all the company accounts. For example: Map items to products, define units of measure. These forms identify the duplicates and recommend ways to fix these duplicates so correctly merge the data.

Consideration for moving company specfic data to shared data:

Here are some of the things that have to be considered when data from company specific tables, meaning tables with the SaveDataPerCompany = yes are upgraded to new Shared tables, meaning tables with SaveDataPerCompany = No.

The most common issue is that the data cannot be merged automatically because N records have to result in 1 record. This means that the user will have to map company specific records to new shared records.

Example – create products based on inventtable records

ItemId should be copied to productNumber and name to productName

ItemID	Name	DataAreaId
A	My a	Dmo
A	M a with different name	Ext

In the above the two items could be merged if name was not copied, but since name is copied, determine which name to use or if the items actually should be considered the same. If they cannot, the user needs to provide a new product number for one of them and tell which item should map to what product. This is where a form is created which shows the duplicates and then provide a way to map this correctly using options such as, map by ID, map 1 to 1, etc.

How do you show the data from all the companies to the user so he can make decisions on the merging and input values?

In Microsoft Dynamics AX 4.0 you did not have crossCompany support. That means that if you want to work on the data from different companies in some ordered way, you have to create shared tables where you copy all the company specific data. These tables can also provide the functionality to enter new values or choose mappings.

During the Microsoft Dynamics Ax 2012 upgrade scripts you could be using these shared DEL_ tables to select from and insert into your new shared Microsoft Dynamics Ax 2012 tables. Consider performance and de-normalized DEL_ tables to make the upgrade script as fast as possible.

In order to get your form to show up in the upgrade preprocessing checklist you need to create a class that extends SysCheckList and add it into the SysCheckList.checkListItems() method.

Merging/Mapping algorithms

If you are merging data and the user should provide input to resolve merge conflicts or do other mappings you most likely need a form, were this can be inputted. But consider the size of the data you are working with. If there is lot of records you most likely need to consider adding some defaults mapping algorithms like:

1. Map all records to a new unique record (here you are not merging anything)
2. Map all records with the same natural key , e.g. all items with same item ID

Validating mappings

Most likely you need to provide some validation of the mappings that have been done. Examples could be that items with color dimension active cannot be mapped to the same product as items without color dimension active.

Depending on how much data there is, you might output these validations in different formats. For the dimension group upgrade an X++ report was chosen and for the product upgrade a whole error logging form and system was developed. It will depend on how much data you expect and how complicated the mapping is. For something like items there can be millions of records, hence it cannot be expected that customers correct all issues in one go and manually one-by-one.

Validating changes to data

Once the customer has finished your upgrade step and provided the data you need, data can still change. New records may be added, others deleted and some changed. That means that you will have to take this into consideration. Most likely you need to create a delta-script that detects such cases.

A word on virtual companies

If a table is shared between 3 companies there will only be one set of records for that table and they have the dataareaid = virtual companies dataareaid. So if there is a chance your tables are shared you should consider what you present to the user when asking him to map and merge. Do you show one record for each actual company or only the one for the virtual company? Experience has shown that writing the actual Microsoft Dynamics Ax 2012 upgrade scripts are easiest if you use the virtual dataareaid but if you have several tables with a mix of non-virtual and virtual you might need to be extra careful when writing your joins on your preprocessing shared tables.

Performance

Reember, these forms are run when Dyanmics Ax system is Live in production. So make sure query used to populate the form or map the data is written appropriately to have the minimum impact on the Dyanmics AX system.

Attribute based upgrade framework (target side scripts only):

> Note: This section is only applicable for target side scripts, i.e. the scripts implemented on Dynamics Ax 2012 code base. The preprocessing upgrade scripts on the source system still use the API-based model.

Earlier version of upgrade framework has an API based upgrade script definition, such initpresync, initpostsync, etc. It required implementing the definition and body of the script separately. It was also very challenging to write any kind of tool which can do the dependency anlaysis for missing upgrade script dependency or redundant dependecy between scripts.

The new upgrade framework in Dynamics Ax 2012 uses X++ attributes to define the upgrade scripts. This architecture simplifies coding upgrade scripts, reduces the possibility of bugs, improves supportability, and makes extending the upgrade framework easier. The biggest advantage of this feature is that users can run a dependency analyzer tool to optimize the upgrade script dependencies.

Here are some of the salient feature of the new framework:

- Upgrade framework uses attributes values to schedule the scripts correctly
- Equivalent upgrade attributes are defined for all upgrade script parameters, which used to required API calls in earlier version
- A dependency analyzer tool is created that analyzes the upgrade attributes to determine script dependencies (Dependency analyzer tool is available through the Upgrade cockpit General tab.)

API based upgrade framework is deprecated and only used for backward compatability to support two version upgrade. All new upgrade scripts on Dyanmics AX 2012 should use new attribute based model. API based upgrade framework will be removed in future versions.

Presynchronization scripts:

Presynchronization scripts in earlier version were mainly implemented for metadata changes which required disabling unique indexes to avoid duplicates. Microsoft Dynamics AX 2012 has included the concept of a Source to Target model. Data from the Source system is copied to the target system. To do this, the upgrade framework uses an algorithem to map the source tables to corresponding target tables.

Fixing table/field mapping errors:

The mapping algorithm follows the following precedence:

1. Check special mapping – this is custom mappiong as defined within presynchronize scripts

2. Try to map by name, e.g. SalesLine -> SalesLine (Salesline table exist on both source and target system)

3. Try map by DEL_ prefix, e.g. SalesLine -> DEL_SalesLine (salesline table on traget doesn't exist but DEL_Salesline table exists)

4. Try to map by ID (Salesline or DEL_salesline table on target doesn't exist but the ID of the Salesline table on source and target is same)

5. Otherwise, mapping error is reported and must be fixed!

The following list describes common mapping errors when you launch the table and field mapping form from the data upgrade checklist.

- Target table is not empty: If target table is not empty then the source table is not copied to target.

- Table name is not matched: target table doesn't have the corresponding source table.
- Field name is not matched: Some fields on source and target and not matched. Even if there is a single field mismatch between tables, the whole table is not copied over unless the mapping issue is fixed.
- Field name in the shadow table does not match: Shadow table is joined with the corresponding source table during bulk copy and data inserted within the mapped target table. If the source table field used within the shadow table doesn't match with target then table is shown as a mapping error.
- Field data type changed: If the field type has changed or if the field length is different, it is considered a mapping error. Many times, an EDT difference between the source and target system may lead to this issue. The simple fix is to make the EDT definition the same.

The following list describes Presynchronize script implementation considerations:

- Write pre-synchronization scripts to provide special mapping.
- If necessary, write data transformation scripts, like postsynchronize scripts. In some cases you may need to write or alter the source side preprocessing scripts.
- Remember that pre-synchronization scripts are running on EMPTY database.
- Don't use pre-synchronization scripts to update the data.
- Use pre-synchronization scripts to change metadata or register table or field mappings.
- Provide configuration keys.
- Use attribute-based models.

Postsynchronization scripts:

The following list describes the purpose for Postsynchronization scripts:

- Executed when the data is copied over from the source system.
- Performs final data transformation.
- Re-enables unique indexes if disabled by pre-synchronization scripts.

The following list describes implementation considerations:

- Use an attribute-based model.
- Use set-based operations to speed up performance.
- Provide UpgradeScriptTableAttribute attribute for all tables used (even for ReleaseUpdateDB41 class & ReleaseUpdateDB401 class to support two version upgrade). This is a special attribute which tells the upgrade framework if the create, read, update, or delete operations happen on the tables being used within the script. If this attribute is missing, the postsynchronize script may run before the data for this particular table is copied over to the target table.
- Provide configuration keys.
- Provide script dependencies.
- Watch out for parallelism and race conditions.

The Preprocess Upgrade Checklist (souce environment)

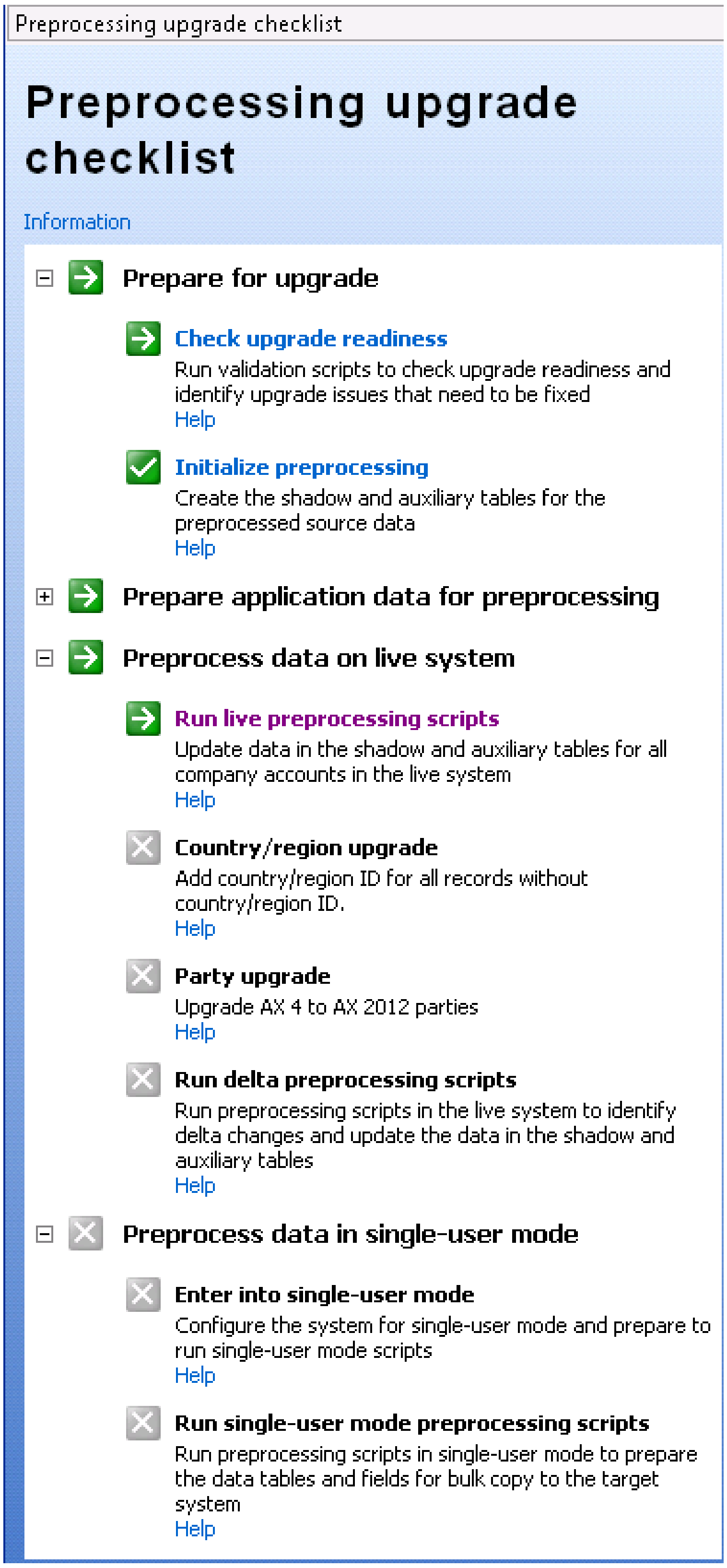
The preprocess upgrade checklist is a navigation pane that guides you through the preprocess data upgrade steps in the source environment. Use the following steps to access the preprocess upgrade checklist.

1. Import the preprocess XPO into the source environment. The XPOs are located in the retail\CD\DatabaseUpgrade directory on the setup CD-ROM.
2. Open the checklist manually. Navigate to the SysCheckList_PreUpgrade40/50 menu item.

Data upgrade is performed using the preprocess upgrade checklist in the following order:

1. Prepare for upgrade
2. Prepare application data for preprocessing
3. Preprocess data on live system
4. Preprocess data in single user mode

The following screenshot illustrates the preprocess upgrade checklist:



Preprocessing upgrade checklist

Preprocessing upgrade checklist

Information

⊟ Prepare for upgrade

Check upgrade readiness
Run validation scripts to check upgrade readiness and identify upgrade issues that need to be fixed
Help

Initialize preprocessing
Create the shadow and auxiliary tables for the preprocessed source data
Help

⊞ Prepare application data for preprocessing

⊟ Preprocess data on live system

Run live preprocessing scripts
Update data in the shadow and auxiliary tables for all company accounts in the live system
Help

Country/region upgrade
Add country/region ID for all records without country/region ID.
Help

Party upgrade
Upgrade AX 4 to AX 2012 parties
Help

Run delta preprocessing scripts
Run preprocessing scripts in the live system to identify delta changes and update the data in the shadow and auxiliary tables
Help

⊟ Preprocess data in single-user mode

Enter into single-user mode
Configure the system for single-user mode and prepare to run single-user mode scripts
Help

Run single-user mode preprocessing scripts
Run preprocessing scripts in single-user mode to prepare the data tables and fields for bulk copy to the target system
Help

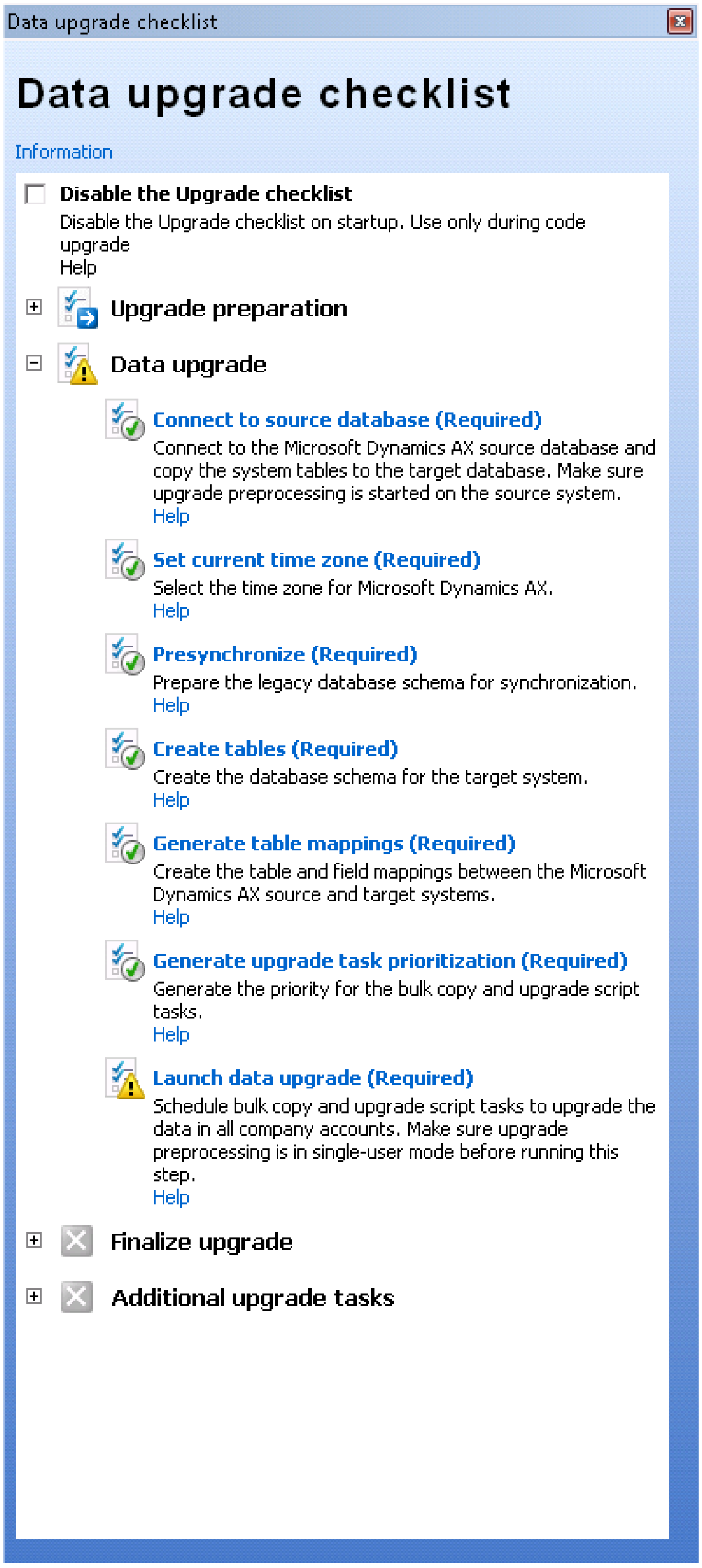
Figure 1. The preprocessing upgrade checklist

The Upgrade Checklist (target environment)

The upgrade checklist is a navigation pane that guides you through the data upgrade steps in the Microsoft Dynamics AX 2012 target environment. It is invoked automatically when Microsoft Dynamics AX starts after a service pack or major release is installed. Data upgrade is performed using the Upgrade Checklist in the following order:

1. Presynchronize
2. Postsynchronize
3. Upgrade additional features

The following screenshot illustrates the upgrade checklist:



Data upgrade checklist

Data upgrade checklist

Information

Disable the Upgrade checklist
Disable the Upgrade checklist on startup. Use only during code upgrade
Help

Upgrade preparation

Data upgrade

Connect to source database (Required)
Connect to the Microsoft Dynamics AX source database and copy the system tables to the target database. Make sure upgrade preprocessing is started on the source system.
Help

Set current time zone (Required)
Select the time zone for Microsoft Dynamics AX.
Help

Presynchronize (Required)
Prepare the legacy database schema for synchronization.
Help

Create tables (Required)
Create the database schema for the target system.
Help

Generate table mappings (Required)
Create the table and field mappings between the Microsoft Dynamics AX source and target systems.
Help

Generate upgrade task prioritization (Required)
Generate the priority for the bulk copy and upgrade script tasks.
Help

Launch data upgrade (Required)
Schedule bulk copy and upgrade script tasks to upgrade the data in all company accounts. Make sure upgrade preprocessing is in single-user mode before running this step.
Help

Finalize upgrade

Additional upgrade tasks

Figure 2. The upgrade checklist

How to Add New Items to the Upgrade Checklist

This section describes how to add new items to the upgrade checklist. You will create a class that extends the SysCheckListItem class. Use the following steps to add an item to the upgrade checklist.

Create a new class that extends the **SysCheckListItem** class and implements the **SysCheckListInterfaceUpgrade** class. Name the class with the **SysCheckListItem_**<name> format where <name> is a unique name.

Override the **getCheckListGroup** method and return the name of the group that the checklist item should appear in.

For example, Upgrade preparation. You may use an existing group name or create a new group.

Nested groups can be created by using the GroupName\SubGroupName format.

All checklist items must belong to a group. Create a new action menu item and name the menu action item the same as the class created in Step 1. Set the following properties:

Label: The text that should appear in the checklist for the item

HelpText: The help text that should appear below the checklist item text

ObjecType: Class

Object: Select the name of the class created in Step 1

SecurityKey: AdminSetup

In the class, override the **getHelpLink** method and provide a link to a .chm topic.

The following code illustrates an example of providing a link to a .chm topic:

```
#define.TopicId('AxShared.chm::/html/7b533e0b-f64d-410e-99ae-0296ace-50900.htm')
return SysCheckListItem::sharedGuide(#TopicId);
```

1. Override the **getMenuItemName** method and return the name of the action menu item created in Step 3.
2. Override the **getMenuItemType** method and return the type of the action menu item created in Step 3.

   The following code illustrates an example of returning an action menu item:

   ```
   return MenuItemType::Action;
   ```
3. If necessary, override the **isRunnable** method. Determine what conditions the checklist item should appear in the checklist. Return **True** to show the checklist item, or return **False** to hide it.
4. Override the **new** method. Specify where your item should appear in the checklist and what other checklist items your checklist item depends on.
   a) Call this.**placeAfter** to specify the order of your checklist item.
   The following code example illustrates the checklist item placed after the **Detect code upgrade conflicts** checklist item.

   ```
   this.placeAfter(classnum(SysCheckListItem_SysUpgradeDetectCon));
   ```
   b) Call this.**addDependency** to specify which checklist items your checklist item depends on.

   The following code example illustrates the checklist item depends on the **Set current time zone** checklist item. The **Set current time zone** checklist item must be completed before this checklist item is enabled.

```
this.addDependency(classnum(SysCheckListItem_BaseTimezoneUpgrade));
```

5. Override the **main** method. This is the method that is executed when the checklist item is clicked in the checklist.
6. In class SysCheckList, add an entry corresponding to your checklist item to the list in method **checkListItems**. The list contains all the possible checklist items. Add your entry in the correct order in which it should appear in the list.

The Data Upgrade Framework for the Source Environment

The data upgrade framework for the source environment gives developers the infrastructure to insert data upgrade scripts written in X++ to be executed in the source environment before the actual upgrade in Microsoft Dynamics AX 2010 starts.

Framework and application scripts are shipped as a separate XPO file which customers have to manually import into their live Microsoft Dynamics AX 4.0 or Microsoft Dynamics AX 2009 environment before the upgrade process starts.

All source preprocessing scripts must be derived from the ReleaseUpdateTransformDB class.

Defining transformation and preprocessing scripts

Before writing upgrade scripts, you must define **transformation**. Transformation and preprocessing on the source environment are used when the upgrade impact of a change is major, and using usual post-syncronize upgrade script in the target environment is not an option for performance reasons.

By defining transformation, you provide:

- the framework that the tables are going to use
- the additional tables that the framework needs to create in the source system
- the scripts to run in the source environment

Later, that information will be used when the upgrade framework copies the data to the target system.

You cannot change any existing tables or any data in the existing source system. The only tables in the source environment which you can insert records into are "shadow tables" created by the framework or the "dictionary tables" backported from Microsoft Dynamics AX 2010.

The following lists the required steps to define transformation for each version, Microsoft Dynamics AX 4 and Microsoft Dynamics AX 2009:

a) Import the preprocessing upgrade framework XPO file (from the installation CD) into the Microsoft Dynamics AX 4 and Microsoft Dynamics AX 2009 development environments.
b) Create a class derived from ReleaseUpdateTransformDB, or modify the existing class. For each version, a set of classes exists - one upgrade class per module. They are named ReleaseUpdateTransformDB<version>_<module>, for example ReleaseUpdateTransformDB41_Bank.
c) Create or modify the initTransformationJobs method. Add the definition of the new transformation.
d) Export your new class and all nessesary tables into the standalone XPO file.

To define a valid transformation, you have to do the following:

1. Create transformation

ReleaseUpdateTransformDB.New()

```
        ReleaseUpdateTransformDB transformation = new ReleaseUpdateTransformDB();
```

Instantiate a new Table Transformation class ***object***

2. Define source table

- Hardcode the source table names (which may require end customers to update that code to fit their customizations) or
- Create automated discovery based on their business rules, for example find all fields with certain EDT or relationships.

void ReleaseUpdateTransformDB.SetSourceTable(Tablename _sourceTablename)

Description: Register transformation class for specific source affected table.

Example:

```
transformation.SetSourceTable(tablestr(LedgerTable));
```

3. Define source fields

void ReleaseUpdateTransformDB.addSourceField(identifiername _fieldName)

Description: Registers a field on the source affected table to be used for transformation. By registering field a source, user tells upgrade framework NOT to copy that field from the source database, but copy a fields registered as target field from the shadow table instead

Example:

```
        transformation.addSourceField(fieldstr(LedgerTable, AccountNum));
```

4. Define Shadow table

Shadow tables should be included in the preprocessing XPO files so they can be created during the XPO import.

5. Define target fields

void ReleaseUpdateTransformDB.addTargetField(extendedtypeId _typeId, identifiername _fieldName)

Description: Registers a field on the target affected table to be used for transformation. That field will be automatically created in the shadow table with TypeID and FieldName provided by the user. Later, when processing BULK COPY of the data from source environment to target, this field will be automatically copied to the target affected table from the shadow table

Example:

```
        transformation.addTargetField(typeid(Description), 'NewAccountNum');
```

6. Per Company To Global Table

If your table is global in Microsoft Dynamics AX 2012 but per company in the previous version, you need to follow these steps:

- Call the API SetPercompanyToGlobal. For example, in the ReleaseUpdateTransformDB50_Basic. initTransformationJobs():transformation_DirPartyTable.SetPerCompanyToGlobal(NoYes::Yes);SaveDataPerCompany is set to No in the shadow table
- Create a field name **Shadow_DataAreaID** in the shadow table which should be populated with the old DataAreaId of the previous main table record.
    - Add a relation from the shadow or dictionary table to the main table on Shadow_DataAreaId and RefRecId fields.
- Implement the logic to copy data from the source table to the shadow table as appropriate

7. Schedule preprocess scripts

- Data upgrade scripts in the source environment are executed in Microsoft Dynamics AX 4.0 and Microsoft Dynamics AX 2009 environments. You have to create a separate version of your script for Microsoft Dynamics AX 4.0 and Microsoft Dynamics AX 2009 upgrades.
- Application teams can provide four kinds of scripts for process their transformation, which have to be scheduled in the initTransformationJobs method.
- Scripts called when the source system is live will result in live preprocessing. These scripts must be written using row-by-row operations.
- Scripts which will be called when system is in the single user mode, but also optionally can be run by a user when the system is live = Delta processing Scripts called when the source system is only in Single User mode = Single User Preprocessing. These scripts must be written as set-based operations.
- Scripts called in a test environment to validate to identify upgrade readiness and data validation issues for data upgrade process. These scripts can be run in a Test Run mode which can be scheduled by clicking on the first item in the preprocessing checklist.

void ReleaseUpdateTransformDB.SetLivePreProcessingScript(ClassId _scriptClassId, identifiername _scriptName, ReleaseUpdateScriptType _scriptType = ReleaseUpdateScriptType::SharedScript, container _configKeys = connull(), boolean _requiresXact = true)

Description: Register preprocessing script template (for the LIVE environment) to be used for the transformation.

Example:

```
        transformation.SetLivePreProcessingScript(classidget(this), methodstr(ReleaseUpdateTransformDB50_Invent,
checkAndPreparePreUpgrade));
```

void ReleaseUpdateTransformDB.SetDeltaPreProcessingScript(ClassId _scriptClassId, identifiername _scriptName, ReleaseUpdateScriptType _scriptType =

ReleaseUpdateScriptType::SharedScript, container _configKeys = connull(), boolean _requiresXact = true)

Description: Register preprocessing script template (for the Singe User Mode and optionally for LIVE environment) to be used for the transformation.

Example:

```
        transformation.SetDeltaPreProcessingScript(classidget(this), methodstr(ReleaseUpdateTransformDB50_Invent,
checkPreUpgrade));
```

void ReleaseUpdateTransformDB.SetSingleUserPreProcessingScript(ClassId _scriptClassId, identifiername _scriptName, ReleaseUpdateScriptType _scriptType = ReleaseUpdateScriptType::SharedScript, container _configKeys = connull(), boolean _requiresXact = true)

Description: Register preprocessing script template (for the Singe User Mode environment) to be used for the transformation.

Example:

```
        transformation.SetSingleUserPreProcessingScript(classidget(this), methodstr(ReleaseUpdateTransformDB50_Invent,
checkPreUpgrade));
```

void ReleaseUpdateTransformDB.SetValidationPreProcessingScript(ClassId _scriptClassId, identifiername _scriptName, ReleaseUpdateScriptType _scriptType = ReleaseUpdateScriptType::SharedScript, container _configKeys = connull(), boolean _requiresXact = true)

Description: Register preprocessing script template (for the Live validation environment) to be used for validation. Validation scripts are only run in a test environment and not in a production environment. See the section *Logging Data Issues in Validation Scripts* for more information about this script type.

```
        transformation.SetValidationPreProcessingScript(classidget(this), methodstr(ReleaseUpdateTransformDB50_Invent,
validatePreUpgrade));
```

8. Save transformation

Save()

Description: Saves the transformation

Example:

```
    TableTransformation.save ();
```

9. Define dependencies between transformations

static void addTransformDependency(RefRecId _firstId, RefRecId _thenId)

Description: Adds dependency between transformations.

Example:

```
        ReleaseUpdateTransformDB::addTransformDependency(Transformation1. getTransformationId(),Transformation2.
getTransformationId());
```

static void addTransformDependencyByTable(tableId _tableId, RefRecId _thenId)

Description: Adds dependency between one transformation and all other transformations that affect the specified table, including transformations from other modules; i.e. ‘cross module’. These dependencies are resolved in ReleaseUpdateTransformDB::initJobs.

Example:

```
        ReleaseUpdateTransformDB::addTransformDependencyByTable(tablenum(<table>),Transformation2.
getTransformationId());
```

The first parameter is the source table of the tranformation, not the shadow table.

static void addTransformDependencyByMethod(classId _classId, IdentifierName _methodName, RefRecId _thenId)

Description: Adds dependency between one transformation and another transformation by class/method. This allows ‘cross module’ transformation dependencies to be defined. These dependencies are resolved in ReleaseUpdateTransformDB::initJobs.

Example:

```
        ReleaseUpdateTransformDB::addTransformDependencyByMethod(classnum(<class>), <methodName>,Transformation2.
getTransformationId());
```

10. Supporting Pause/Resume Functionality and Committing Data in Batches

During an upgrade you may want to pause an upgrade script and later resume the script from where it left off. In addition, you may want to improve performance by committing processed data in batches. The Upgrade Preprocessing Framework has API support for these scenarios. This is called script throttling. In most cases, live preprocessing scripts should be implemented with throttling.

The ReleaseUpdateTransactionManager class implements the following methods that support script throttling.

When calling the Set* methods in initTransformationJobs to register your upgrade scripts, be sure to set the requiresXact flag to false.

Example:

```
SetLivePreProcessingScript(_classnum(ReleaseUpdateTransformDB_Admin),
methodstr(ReleaseUpdateTransformDB_Admin, myScriptName,
ReleaseUpdateScriptType::SharedScript, connull(), false);
```

ReleaseUpdateTransactionManager::newTransactionManager(classId _upgradeScriptClassId, identifierName _upgradeScriptMethodName, identifierName _helperMethodName = '', freeText _blockId = '')

Description: Creates a new instance of the transaction manager and associates it with your upgrade script. An optional helper method name and block ID can be specified for multiple code blocks. The _blockId can be any value that is appropriate for your situation.

When you use multiple instances of ReleaseUpdateTransactionManager in your upgrade script, if one instance receives a pause command be sure to properly exit the upgrade script. Do not allow the script to continue. ReleaseUpdateTransactionManager::newTransactionManager will throw an exception if it detects another instance was paused. See the example below.

bool initialize()

Description: Initializes the transaction manager. Returns 'true' if the script is in resume mode.

validateTransaction(container _controlData)

Description: Call this method at the beginning of the while select loop in the upgrade script. The method determines if a ttsbegin is needed. For Microsoft Dynamics 2009, the ttsbegin and matching ttscommit must be within the same code block. This method accounts for that limitation.

container geLasttIterationState

Description: Returns information that the script uses to determine at what point it should resume. In most cases, this can be the last RecId processed by the script, but can be any set of values that make sense for your situation (control data).

boolean doIterationAndContinue(container _controlData)

Description: Call at the end of the while select loop in the upgrade script. The method keeps track of how many iterations the while select has processed and automatically performs a ttscommit and ttsbegin when the iteration limit has been reached. For example, if a script should process batches in rows of 10,000, this method will count the rows, then commit when it reaches 10,000. The method returns false if the script has been paused by the user. Your script should 'break' at this point.

The pattern for Delta scripts is different. In the following example, the RecVersion was added to the *Order By* clause because *Order By* does not work with RecId due to a bug in Microsoft Dynamics AX.

The following code is an example of using the throttling API in a Live script:

```
public void throttlingAPITest()

{

    testTableSource     srcTable;

    testTable2Source    srcTable2;

    testTableShadow     shadowTable;

    RefRecId            lastRecId;

    boolean             isResuming;
    ReleaseUpdateTransactionManager transactionManager1;
    ReleaseUpdateTransactionManager transactionmanager2;
    ;
    // Do not delete_from the shadow table here
    // Create a new instance of the transaction manager
    transactionManager1 =
ReleaseUpdateTransactionManager::newTransactionManager(classidget(this),
        methodstr(ReleaseUpdateTransformDBTest, throttlingAPITest), '', '1');
    // Determine if this script is resuming from a previous pause
    isResuming = transactionManager1.initialize();
    // Retrieve the last RecId processed
    [lastRecId] = transactionManager1.getLastIterationState();

    while select *
        from srcTable
        order by RecId, RecVersion
        where (other conditions) && (srcTable.RecId > lastRecId || !isResuming) // Understand your script
logic and implement accordingly – don not copy and paste as this example uses RecId to track the last
processed row. Your scenario may vary.
    {
        // validateTransaction will start the transaction, if needed
        transactionManager1.validateTransaction([srcTable.RecId]);

        shadowTable.clear();
        shadowTable.Name = srcTable.Name;
        shadowTable.insert();

        // doIterationAndContinue checks if a commit should be performed
        if (!transactionManager1.doIterationAndContinue([srcTable.RecId]))
        {
            // If doIterationAndContinue returns false, exit
            return;
        }
    }
    // Complete the final transaction
    transactionManager1.endTransaction([srcTable.RecId]);
    // Create a new instance of the transaction manager for loop 2
    transactionManager2 =
ReleaseUpdateTransactionManager::newTransactionManager(classidget(this),
        methodstr(ReleaseUpdateTransformDBTest, throttlingAPITest), '', '2');
    // Determine if this script is resuming from a previous pause
    isResuming = transactionManager2.initialize();
    // Retrieve the last RecId processed
    [lastRecId] = transactionManager2.getLastIterationState();

    while select *
        from srcTable2
        order by RecId, RecVersion
        where (srcTable.RecId > lastRecId) || !isResuming
    {
        // validateTransaction will start the transaction, if needed
        transactionManager2.validateTransaction([srcTable2.RecId]);

        shadowTable.clear();
        shadowTable.Name2 = srcTable2.Name;
        shadowTable.insert();

        // doIterationAndContinue checks if a commit should be performed
        if (!transactionManager2.doIterationAndContinue([srcTabl2e.RecId]))
        {
            // If doIterationAndContinue returns false, exit
            return;
        }
    }
    // Complete the final transaction
    transactionManager2.endTransaction([srcTable2.RecId]);
}
```

The RecID might not be unique, select a combination of fields to track the processed rows in your query. For example, with non-shared tables, DataAreaID might be used in your query condition:

```
while select crosscompany AddressCounty order by DataAreaId, RecId, RecVersion

where (other conditions) && ((AddressCounty.DataAreaId > lastDataAreaId ||
(AddressCounty.RecId > lastRecId && AddressCounty.DataAreaId == lastDataAreaId)) ||
!isResuming)
```

In this case, DataAreaId should be an additional input/output when calling the throttling APIs, for example:

```
…

[lastRecId, lastDataAreaId] = transactionManager.getLastIterationState();

…

if (!transactionManager.doIterationAndContinue([AddressCounty.RecId,
AddressCounty.DataAreaId]))

…
```

DataAreaId is used for a shared script only. For a standard script, which runs per company, DataAreaId is redundant in the above query.

Example of API use in a Delta script:

```
isResuming = transactionManager.initialize();//isResuming might not be used but initialize() should be
called.


delete_from shadow_InventCostTrans

notexists join RecId, recVersion from InventCostTrans

where   InventCostTrans.RecId        == shadow_InventCostTrans.RefRecId &&

        InventCostTrans.recVersion    == shadow_InventCostTrans.RecVersionId;


while select RecId, recVersion, Dimension from InventCostTrans

order by InventCostTrans.RecId, InventCostTrans.recVersion

notexists join shadow_InventCostTrans

where   InventCostTrans.RecId        == shadow_InventCostTrans.RefRecId &&

        InventCostTrans.recVersion    == shadow_InventCostTrans.RecVersionId ~~&&~~

~~        ((InventCostTrans.RecId > lastRecId) || !isResuming) ~~// • Do not add a where clause for checking
last recid processed here. Deltas may have occurred in rows prior to this. The not exists join takes care
of this.

{

    // validateTransaction will start the transaction, if needed

    transactionManager.validateTransaction([InventCostTrans.RecId]);
    shadow_InventCostTrans.RefRecId            = InventCostTrans.RecId;
    shadow_InventCostTrans.RecVersionId        = InventCostTrans.recVersion;
    shadow_InventCostTrans.DefaultDimension    =
DimensionConversionHelper::getNativeDefaultDimension(InventCostTrans.Dimension);

    shadow_InventCostTrans.insert();

    // doIterationAndContinue checks if a commit should be performed
    if (!transactionManager.doIterationAndContinue([InventCostTrans.RecId]))
    {
        // If doIterationAndContinue returns false, exit
        return;
    }
}

// Complete the final transaction
transactionManager.endTransaction([InventCostTrans.RecId]);
```

11. Logging Data Issues in Validation Scripts

Validation scripts in preprocessing have a unique feature that allows them to identify data issues discovered and alert the user. The script can rely on the framework to present these issues to the user, and can optionally provide a custom UI to display or fix these issues. To provide these features, validation scripts can use the following API:

ReleaseUpdateValidationLogger

ReleaseUpdateValidationMessages (macros)

log(ReleaseUpdateMessageId _messageId, ReleaseUpdateLogResultType _logResult, freeText _messageText, freeText _resolution, SysInfoAction _action = null, ReleaseUpdateActionType _actionType = ReleaseUpdateActionType::None)

logDetail(ReleaseUpdateMessageId _messageId, freeText _message)

ReleaseUpdateValidationLogger::newValidationLogger(classId _upgradeScriptClassId, identifierName _upgradeScriptMethodName)

Description: Creates a new instance of the validation logger and associates it with your upgrade script.

> ***log(ReleaseUpdateMessageId _messageId, ReleaseUpdateLogResultType _logResult, freeText _messageText, freeText _resolution, SysInfoAction _action = null, ReleaseUpdateActionType _actionType = ReleaseUpdateActionType::None)***

Description: Logs a unique message. The message is tracked by _messageId (entered in ReleaseUpdateValidationMessages) and only logged once per script, even if the log message is called multiple times with the same messageId.

logDetail(ReleaseUpdateMessageId _messageId, freeText _message)

Description:Logs a detail message associated with a previously logged messageId (log).

The following is an example of a log entry and its corresponding detail:

Log:	There are invalid states in table x
Detail:	Row with RecId 1 has invalid state 'Wash'
	Row with RecId 2 has invalid state 'Mass'
	Row with RecId 100 has invalid state 'Fla'

The following is an example of a script using the validation APIs:

```
#ReleaseUpdateMessageIds

MyAddresses                     myAddresses;
SysInfoAction_ValidationForm    action;
ReleaseUpdateLoggerInterface    logger;
;

action = new SysInfoAction_ValidationForm::newFormname(formstr(MyAddressesResolveState));

logger = ReleaseUpdateValidationLogger::newValidationLogger(
                                    classidget(this),
                                    methodstr(ReleaseUpdateTransformDB41_Admin,
                                    validateMyAddresses));

while select *
    from myAddresses
{
    if (!this.IsValidState(myAddresses.State))
    {
            logger.log(
                #myAddressesStateInvalid,                  // MessageId
                ReleaseUpdateLogResult::Error,             // Type
                "The following rows have invalid states:",  // Message Text
                "Update these rows to use valid states.",   // Resolution
                action,                                  // SysInfoAction
                ReleaseUpdateActionType::Fix)             // ReleaseUpdateActionType

            logger.logDetail(
                #myAddressesStateInvalid,
                strFmt("RecId: %1 (%2)",
                myAddresses.RecId,
                myAddresses.State))
        }
    }
```

12. Modify a subset of records in a table during upgrade

Using the upgrade framework, you can upgrade a subset of records in a table while keeping the rest unchanged.

For example, the MarkupTrans table has the Keep field which should only be changed for a subset of records.

The recommended solution is to write the upgrade scripts as if the field should be upgraded for the entire table.

The difference is that the **select** statement in the upgrade scripts should have a **where** clause to indicate that only records that should be upgraded are selected. Shadow records are then created for the records that should be upgraded.

When registering the upgrade script in the transformation, it should set the shadow table join type to be left outer join:

```
    transformation.setShadowTableJoinType(ReleaseUpdateJoin::LeftOuterJoin);
```

During bulk copy, the framework will then automatically set the upgraded field to the upgraded value if one exists otherwise it will keep the original value for the field.

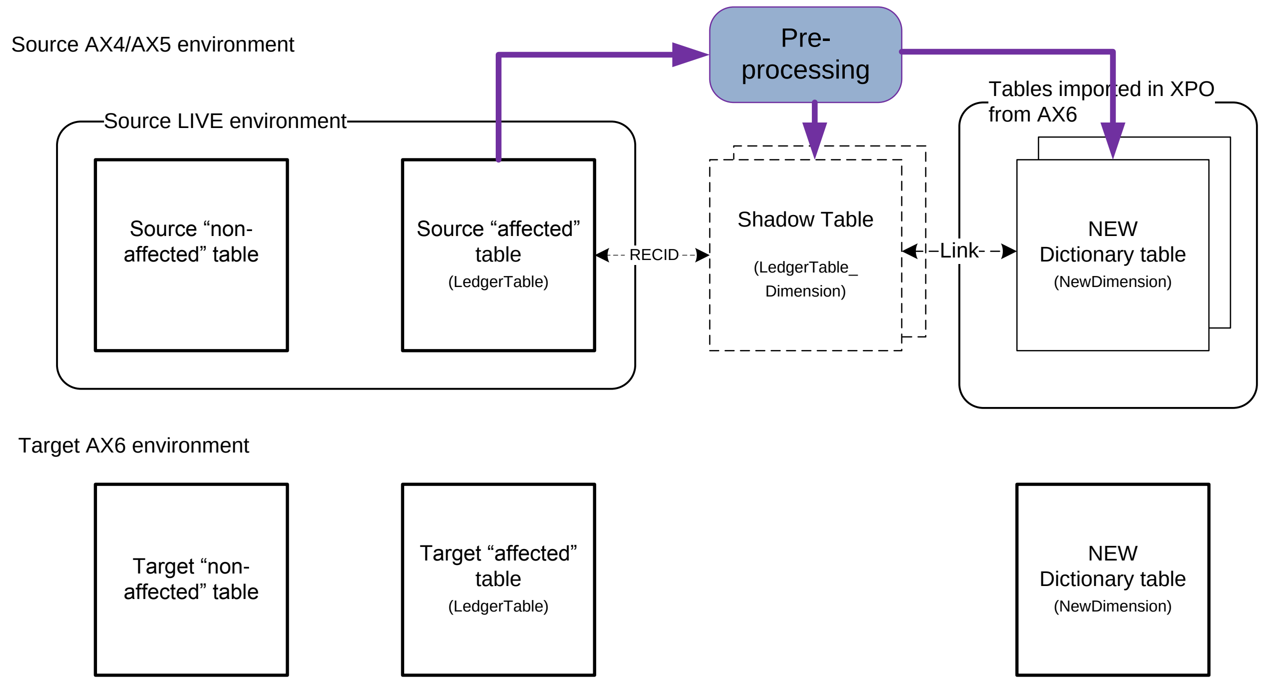
This approach assumes that the fields on a record which determine whether it should have a record in the shadow table cannot change in the source environment. Otherwise, it is a challenge to identify modified records.

Writing Data Upgrade Scripts for Source Environment

This section contains information on writing Live preprocessing scripts and writing Delta and single user mode preprocessing scripts.

Writing Live preprocessing scripts

Preprocessing scriptsc can Direct SQL or X++ application scripts executed from within AX by the framework:


Source AX4/AX5 environment
Pre-processing
Tables imported in XPO from AX6
Source LIVE environment
Source "non-affected" table
Source "affected" table (LedgerTable)
RECID
Shadow Table (LedgerTable_Dimension)
Link
NEW Dictionary table (NewDimension)
Target AX6 environment
Target "non-affected" table
Target "affected" table (LedgerTable)
NEW Dictionary table (NewDimension)

Scripts must correctly accept source affected table names and field names, and respective shadow table name, and perform necessary transformation of data from source tables/fields:

1. **Records in new tables**: should be created directly in the new "dictionary" table (for example, NewDimension table) which was imported from target system into the source system
2. **Modified fields of the existing "source" table**: saved in a "shadow" table, linked with the source table using RecId and RecVersionId link

```
preUpgradeScript(SourceTableName, SourceFieldNames[], ShadowTableName, ShadowFieldNames[])
{
// application logic goes here


//insert into new dictionary tables
WHILE SELECT FROM SourceTableName
{
```

```
            // Check if record exist in Dictionary Table1
            SELECT FROM Dictionary Table1 where <some condition based on SourceTableName >

            // IF it does not exist, insert it
            INSERT INTO DictionaryTable1 <some fields from SourceTableName>

            // Check if record exist in Dictionary Table2
            SELECT FROM Dictionary Table2 where <some condition based on SourceTableName >

            // IF it does not exist, insert it
            INSERT INTO DictionaryTable2 <some fields from SourceTableName>

            // insert into "shadow table" modified source fields link to the Dictionary tables and
            // RecId/RecVersionId link to the source table

            S = INSERT INTO ShadowTableName
            (REFRECID, REFDATAAREAID,  RECVERSIONID, shadowFieldName[1], shadowFieldName[2])
            SourceTableName.RECID,
            SourceTableName.DATAAREAID,
            SourceTableName.RECVERSIONID
            DictionaryTable1.targetField1,
            DictionaryTable2.targetField2
            executeStatement(S);
}
}
```

If direct SQL is used, then two versions of the script has to be implemented, Oracle and MS SQL.

Writing Delta and single user mode preprocessing scripts

Purpose of these scripts is to check the state of the source tables since preprocessing script was run on the LIVE system, and resolve all discrepancies occurred since.

These will be Direct SQL or application scripts executed from within AX by the framework. It'll be script's responsibility to identify new or modified records in the source table and take appropriate actions (for example, rerun the business logic for new records). However, framework will provide some guidance on this, for example:

There are 3 types of changes which may have happened to a record

1. How to identify new records
   Script can identify new records by using shadow table RefRecIds.

For instance,

```
            SELECT SourceTable.RecIDs NOT EXIST ShadowTable.RefRecId
```

If such record was found, we can execute preprocessing script on it

2. Record updated

Script can find updated records using RecVersion field. A search like provided below can be used:

```
SELECT SourceTable, ShadowTable where SourceTable.RECID = TargetTable.RECID and SourceTable.RecVersion !=
ShadowTable.RecVersion
```

If such record was found, we will delete the record from shadow table and re-run preUpgradeScript() on it. However, that approach may leave some orphan records in the "dictionary" tables. Also, RecVersion may not be very reliable due to third party components writing directly to the database

3. Record deleted

Script can identify new records by using shadow table RefRecIds, but usually there is no need no need as later we will be joining Shadow table and Source table on RecID (see section 4). However, orphan records may be left in the "dictionary" tables

As soon as table's delta processing completed for all transformations, the table must be marked as **ready to be copied**

The Data Upgrade Framework for the Target Environment

The data upgrade framework gives developers the infrastructure to insert data upgrade scripts written in X++. The data upgrade framework manages the dependencies of the scripts, schedules them to be run in parallel by batch clients, and provides progress reports on the running scripts. The data upgrade framework has a built-in error recovery mechanism that helps to ensure system integrity when the upgrade has to be resumed after an error.

With the exception of the base ReleaseUpdateDB class, the ReleaseUpdateDB* classes contain implementations of data upgrade scripts. The scripts provide abstract methods and utility functions for data upgrade classes.

Classes with preprocessing upgrade scripts are derived from the class ReleaseUpdateTransformDB, and have different types of scripts and ways of scheduling them.

The following diagram illustrates the class diagram of the upgrade script classes.

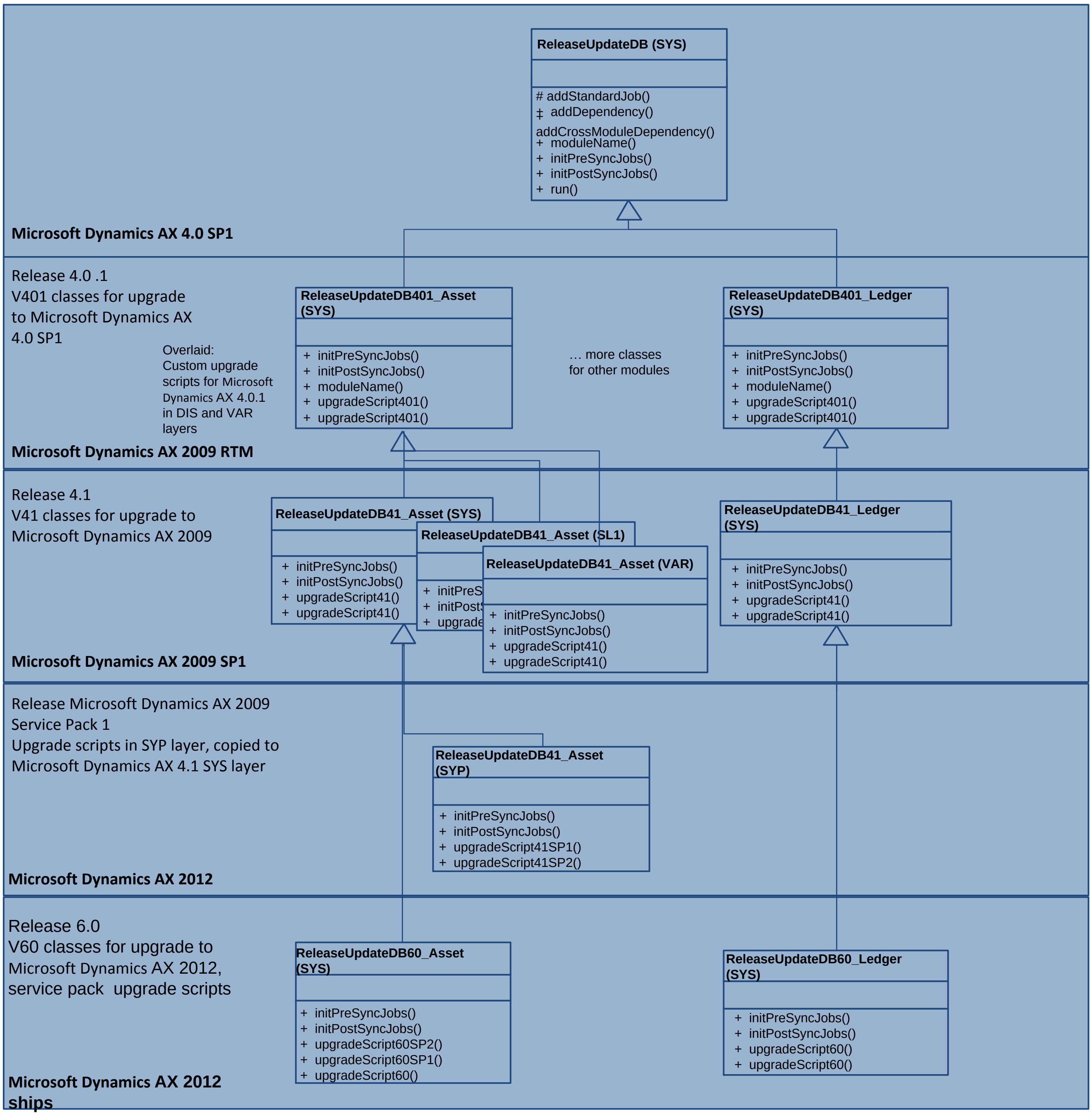

ReleaseUpdateDB (SYS)
# addStandardJob()
‡ addDependency()
addCrossModuleDependency()
+ moduleName()
+ initPreSyncJobs()
+ initPostSyncJobs()
+ run()
Microsoft Dynamics AX 4.0 SP1
Release 4.0 .1
V401 classes for upgrade to Microsoft Dynamics AX 4.0 SP1
Overlaid: Custom upgrade scripts for Microsoft Dynamics AX 4.0.1 in DIS and VAR layers
ReleaseUpdateDB401_Asset (SYS)
+ initPreSyncJobs()
+ initPostSyncJobs()
+ moduleName()
+ upgradeScript401()
+ upgradeScript401()
… more classes for other modules
ReleaseUpdateDB401_Ledger (SYS)
+ initPreSyncJobs()
+ initPostSyncJobs()
+ moduleName()
+ upgradeScript401()
+ upgradeScript401()
Microsoft Dynamics AX 2009 RTM
Release 4.1
V41 classes for upgrade to Microsoft Dynamics AX 2009
ReleaseUpdateDB41_Asset (SYS)
+ initPreSyncJobs()
+ initPostSyncJobs()
+ upgradeScript41()
+ upgradeScript41()
ReleaseUpdateDB41_Asset (SL1)
+ initPreS
+ initPostS
+ upgrade
ReleaseUpdateDB41_Asset (VAR)
+ initPreSyncJobs()
+ initPostSyncJobs()
+ upgradeScript41()
+ upgradeScript41()
ReleaseUpdateDB41_Ledger (SYS)
+ initPreSyncJobs()
+ initPostSyncJobs()
+ upgradeScript41()
+ upgradeScript41()
Microsoft Dynamics AX 2009 SP1
Release Microsoft Dynamics AX 2009 Service Pack 1
Upgrade scripts in SYP layer, copied to Microsoft Dynamics AX 4.1 SYS layer
ReleaseUpdateDB41_Asset (SYP)
+ initPreSyncJobs()
+ initPostSyncJobs()
+ upgradeScript41SP1()
+ upgradeScript41SP2()
Microsoft Dynamics AX 2012
Release 6.0
V60 classes for upgrade to Microsoft Dynamics AX 2012, service pack upgrade scripts
ReleaseUpdateDB60_Asset (SYS)
+ initPreSyncJobs()
+ initPostSyncJobs()
+ upgradeScript60SP2()
+ upgradeScript60SP1()
+ upgradeScript60()
ReleaseUpdateDB60_Ledger (SYS)
+ initPreSyncJobs()
+ initPostSyncJobs()
+ upgradeScript60()
+ upgradeScript60()
Microsoft Dynamics AX 2012 ships

Figure 2. Data Upgrade Script Classes

Data Upgrade Scripts by Module

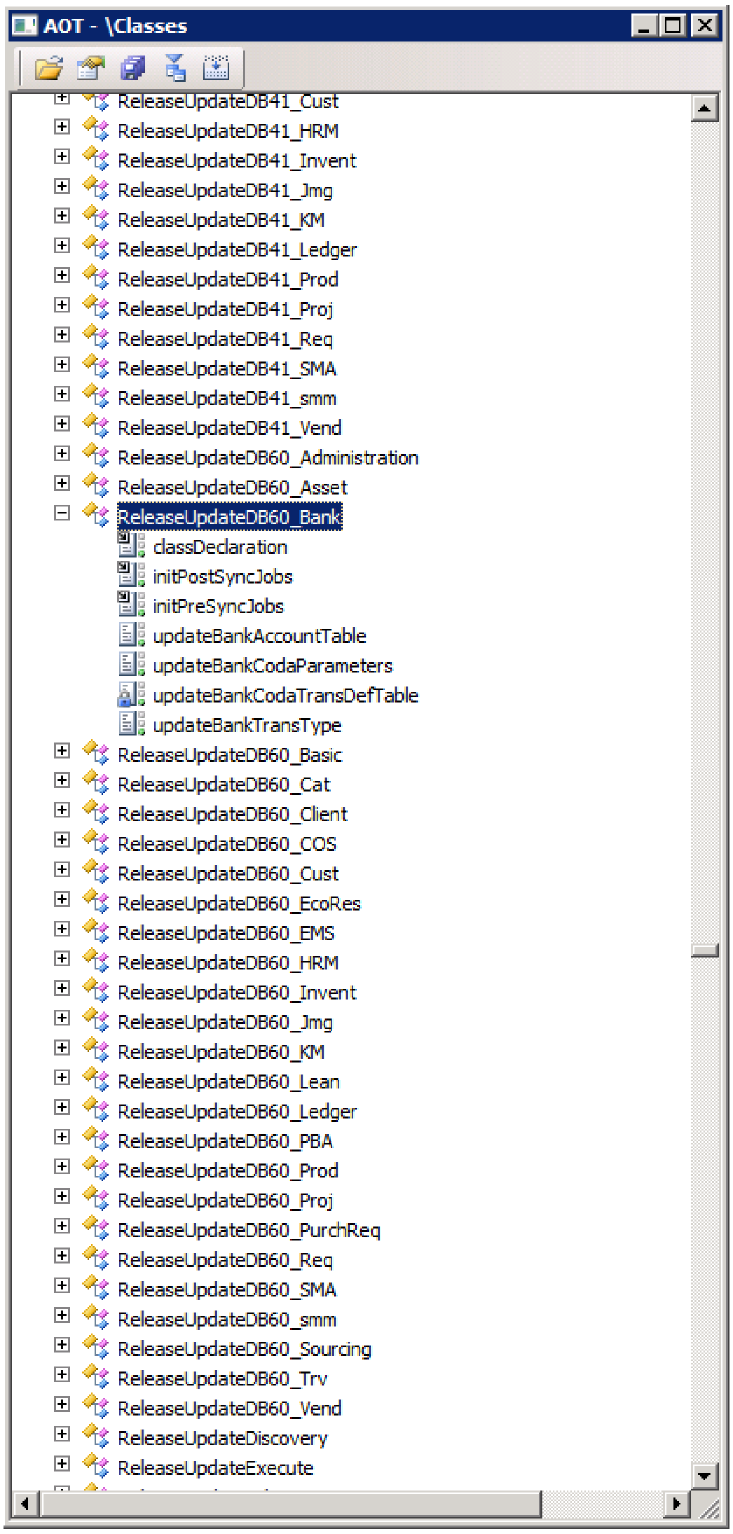
Data upgrade scripts are inserted into the data upgrade as methods of a ReleaseUpdateDB<NN>_<module> class, where <NN> is the version of Microsoft Dynamics AX being upgraded to, and <module> is the module name the script belongs to. These classes are derived from the base class ReleaseUpdateDB and are connected to the data upgrade framework.

When you create upgrade scripts for your version of Microsoft Dynamics AX, you can use any of the new classes in the following table according to your script application module and the version you are developing.

Version 401	Version 41	Version 60
ReleaseUpdateDB401_Administration	ReleaseUpdateDB41_Administration	ReleaseUpdateDB60_Administration
ReleaseUpdateDB401_Bank	ReleaseUpdateDB41_Asset	ReleaseUpdateDB60_Asset
ReleaseUpdateDB401_COS	ReleaseUpdateDB41_Bank	ReleaseUpdateDB60_Bank
ReleaseUpdateDB401_Cust	ReleaseUpdateDB41_Basic	ReleaseUpdateDB60_Basic
ReleaseUpdateDB401_Ledger	ReleaseUpdateDB41_COS	ReleaseUpdateDB60_Cat
ReleaseUpdateDB401_Proj	ReleaseUpdateDB41_Cust	ReleaseUpdateDB60_Client
ReleaseUpdateDB401_Vend	ReleaseUpdateDB41_HRM	ReleaseUpdateDB60_COS
	ReleaseUpdateDB41_Invent	ReleaseUpdateDB60_Cust
	ReleaseUpdateDB41_Jmg	ReleaseUpdateDB60_EcoRes
	ReleaseUpdateDB41_KM	ReleaseUpdateDB60_EMS
	ReleaseUpdateDB41_Ledger	ReleaseUpdateDB60_HRM
	ReleaseUpdateDB41_Prod	ReleaseUpdateDB60_Invent
	ReleaseUpdateDB41_Proj	ReleaseUpdateDB60_Jmg
	ReleaseUpdateDB41_Req	ReleaseUpdateDB60_KM
	ReleaseUpdateDB41_SMA	ReleaseUpdateDB60_Lean
	ReleaseUpdateDB41_smm	ReleaseUpdateDB60_Ledger
	ReleaseUpdateDB41_Trv	ReleaseUpdateDB60_PBA
	ReleaseUpdateDB41_Vend	ReleaseUpdateDB60_Prod
		ReleaseUpdateDB60_Proj
		ReleaseUpdateDB60_PurchReq
		ReleaseUpdateDB60_Req
		ReleaseUpdateDB60_SMA
		ReleaseUpdateDB60_smm
		ReleaseUpdateDB60_Sourcing
		ReleaseUpdateDB60_Trv
		ReleaseUpdateDB60_Vend

The naming convention ReleaseUpdateDB60 means upgrade to Microsoft Dynamics AX 2012. Pre-synchronization, Post-synchronization, and Additional features upgrade methods coexist in these classes.

The following screenshot illustrates a view of the upgrade script classes in the AOT.

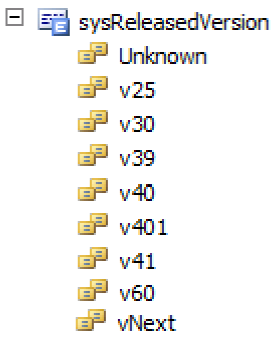

AOT - \Classes

```
ReleaseUpdateDB41_Cust
ReleaseUpdateDB41_HRM
ReleaseUpdateDB41_Invent
ReleaseUpdateDB41_Jmg
ReleaseUpdateDB41_KM
ReleaseUpdateDB41_Ledger
ReleaseUpdateDB41_Prod
ReleaseUpdateDB41_Proj
ReleaseUpdateDB41_Req
ReleaseUpdateDB41_SMA
ReleaseUpdateDB41_smm
ReleaseUpdateDB41_Vend
ReleaseUpdateDB60_Administration
ReleaseUpdateDB60_Asset
ReleaseUpdateDB60_Bank
    classDeclaration
    initPostSyncJobs
    initPreSyncJobs
    updateBankAccountTable
    updateBankCodaParameters
    updateBankCodaTransDefTable
    updateBankTransType
ReleaseUpdateDB60_Basic
ReleaseUpdateDB60_Cat
ReleaseUpdateDB60_Client
ReleaseUpdateDB60_COS
ReleaseUpdateDB60_Cust
ReleaseUpdateDB60_EcoRes
ReleaseUpdateDB60_EMS
ReleaseUpdateDB60_HRM
ReleaseUpdateDB60_Invent
ReleaseUpdateDB60_Jmg
ReleaseUpdateDB60_KM
ReleaseUpdateDB60_Lean
ReleaseUpdateDB60_Ledger
ReleaseUpdateDB60_PBA
ReleaseUpdateDB60_Prod
ReleaseUpdateDB60_Proj
ReleaseUpdateDB60_PurchReq
ReleaseUpdateDB60_Req
ReleaseUpdateDB60_SMA
ReleaseUpdateDB60_smm
ReleaseUpdateDB60_Sourcing
ReleaseUpdateDB60_Trv
ReleaseUpdateDB60_Vend
ReleaseUpdateDiscovery
ReleaseUpdateExecute
```

Figure 3. Upgrade Classes in the AOT

SYS Versions and Data Upgrade of Interim SYS releases

The SYS layer contains the core functionality of Microsoft Dynamics AX. A modification to this layer is shipped to partners and customers in beta versions (for example, Microsoft Dynamics AX 4.0 TAP3), final release version (for example, Microsoft Dynamics AX 4.0), and refresh versions of major releases (for example, Microsoft Dynamics AX 4.0.1), referred to here as interim SYS releases. The data upgrade framework supports upgrades that span multiple SYS releases by providing the infrastructure to incrementally upgrade from one SYS release to another, later release as shown in the following screenshot.



sysReleasedVersion
- Unknown
- v25
- v30
- v39
- v40
- v401
- v41
- v60
- vNext

SYS versions are defined in the Base Enum SysReleaseVersion

Each ReleaseUpdateDB* class (except for the base ReleaseUpdateDB class and preprocessing upgrade classes) is associated with a SYS version and named accordingly. The class hosts the data upgrade scripts that upgrade the SYS data model from the previous SYS version to the current SYS version.

Upgrade scripts can span more than one SYS release. Therefore, each data upgrade script class inherits upgrade scripts from the class of the same module in the most recent previous release. When you need upgrade scripts for a new interim release, and when the upgrade script class for the corresponding module does not yet exist, you create the class that uses the right naming convention and ensure this class inherits upgrade scripts from the previous version of the upgrade script class of the same module.

For example, in Figure 2, the Ledger module has upgrade scripts for version 4.0.1, 4.1, and 6.0 but does not have an upgrade script for release version 4.0.1. For the Asset module, there are upgrade scripts for versions 401, 41, 60 (Microsoft AX 2012). Therefore the class ReleaseUpdateDB41_Asset must inherit from ReleaseUpdateDB401_Asset, which in turn inherits from ReleaseUpdateDB.

```
Public class ReleaseUpdateDB41_Asset extends ReleaseUpdateDB401_Asset
{
}
```

```
Public class ReleaseUpdateDB60_Asset extends ReleaseUpdateDB41_Asset
{
}
```

In order to incrementally upgrade from a SYS release that is two or more versions earlier, the initPreSyncJobs, initPostSyncJobs and initAdditionalJobs methods must be overridden and you must call “#initSyncJobsPrefix” to include the previous upgrade. The initPreSyncJobs, initPostSyncJobs and initAdditionalJobs jobs detect the earlier (“from”) version of the upgrade and skips if necessary.

```
void initPostSyncJobs()
{

        #initSyncJobsPrefix
}
```

Finally, the purpose of an individual script is to upgrade a tables data from SysVer -1 to SysVer. Each script is used to upgrade the data to the current version.

Writing Data Upgrade Scripts for ISV or customized solution in the Target Environment

An ISV solution has two options to write upgrade scripts:

Option 1: Use the upgrade framework to write upgrade scripts in the same way as the core upgrade scripts have been created in the SYS layer. Choosing this option, the ISV solution should be installed before running the major version upgrade, all ISV scripts will be loaded and scheduled along with the core upgrade scripts during the major upgrade.

Option 2: Use the upgrade framework to write upgrade scripts for a minor upgrade after the major version upgrade has been completed. Choosing this option, ISV scripts can be implemented in the same way as option 1 with some differences:

- The preSync scripts must use the enum ReleaseUpdateScriptStage::PreSync**Update**
- The postSync scripts must use the enum ReleaseUpdateScriptStage::PostSync**Update**
- The Additional scripts must use the enum ReleaseUpdateScriptStage::Additional**Update**
- The script classes should implement initPreSync**Update**Jobs(),initPostSync**Update**Jobs(), and initAdditional**Update**Jobs(). All of these methods should contain a single call to the local macro, for example:

  ```
  void initPostSyncUpdateJobs()
  {
     #initSyncJobsPrefix
            }
  ```

- After installing the scripts, ISV installer should call the ReleaseUpdateDB::RegisterForMinorUpgradeScript method in Microsoft Dynamics AX to register the scripts. The following code example registers three scripts. Use ttsbegin and ttscommit to make sure all or none of the script is registered.

  ```
  ttsbegin;

  ReleaseUpdateDB::RegisterForMinorUpgradeScript(
    classStr(ReleaseUpdateDB60_ISV01),  methodStr(ReleaseUpdateDB60_ISV01, Script01));

  ReleaseUpdateDB::RegisterForMinorUpgradeScript(
    classStr(ReleaseUpdateDB60_ISV01),  methodStr(ReleaseUpdateDB60_ISV01, Script02));

  ReleaseUpdateDB::RegisterForMinorUpgradeScript(
  ```

```
classStr(ReleaseUpdateDB60_ISV01),  methodStr(ReleaseUpdateDB60_ISV01, Script03));

ttscommit;
```

- After the scripts have been registered, the next time Microsoft Dynamics AX starts, the Update checklist will display to allow users to schedule and run the ISV scripts, the checklist can also be opened manually using the display menu item SysCheckList_Update.

Create a single upgrade script that combines changes across multiple product versions

When upgrading to version *n* (target) from version *n-2* (source), you can sometimes provide an algorithm that upgrades data directly from the source to the target version without upgrading to the interim version. We call these algorithms combined upgrade scripts. In cases for which you can create a combined upgrade script, follow the best practices below:

- Place the algorithm in the upgrade class for the source version, replacing the original algorithm. For example, if you are upgrading from version 4.0 to 6.0, put the combined algorithm in the ReleaseUpdateDB401 class.
- Put a condition in a script in the upgrade class for the target version, setting it to execute only if you are not upgrading from the source version. For example, change the script in the 6.0 version to

```
public void updateCustTrans()
{
   if (ReleaseUpdateDB::getFromVersion() != sysReleasedVersion::v40)
   {
               Original script logic for upgrade from 5.0 to 6.0
   }
}
```

Using Configuration Key to Remove Obsolete Objects after Upgrade

After the upgrade is finished, you can disable the configuration keys “Keep update objects” (SysDeletedObjects40, SysDeletedObjects41 and SysDeletedObjects60 for Microsoft Dynamics AX 2012). After database synchronization is complete, all obsolete components of the data model will be removed and performance will be improved. The components that are removed are those needed to perform the data upgrade, but provide no value when the process is completed.

Data Upgrade Scripts in the target environment

Data upgrade scripts comprise the majority of the data upgrade framework. For each version, a set of classes exists - one upgrade class per module. Currently, there are 26 application modules for upgrade scripts. They are named ReleaseUpdateDB<version>_<module>, for example ReleaseUpdateDB60_Bank.

Each of these classes contains scripts for pre-synchronization, post-synchronization and additional upgrades.

The scripts are scheduled by their attributes specified at the beginning of every script method.

The methods initPreSyncJobs, initPostSyncJobs and initAdditionalJobs are still kept for version checking.

Each class can handle your upgrade script in one of four different ways - Start, Shared, Standard, and Final. Choose the right one so that the script runs at the correct time and in the correct manner:

Pre-synchronization	Post-synchronization	Additional upgrade
Start (allow duplicates)	-	
Shared/Normal	**Shared/Normal**	Shared/normal
-	Final (undo allow duplicates)	

- **Presynchronize Start scripts**

  (Executed first)

  Start scripts are used to change indexes that have become unique in order to allow duplicates. This is a modification of meta data and must be undone in a post-synchronization final script (see below). Start scripts are run once versus once per company as with normal scripts.

- **Presynchronize Shared scripts**

  (Executed once in parallel with pre-synchronization normal scripts)

  Shared scripts are used mainly for cleanup jobs such as deleting duplicate records for tables that have changed an index from allowing duplicates to being unique. Shared scripts are run at the same time as normal scripts. The only way to ensure that a shared script is run before another shared script or a normal script is to set up a dependency between the scripts. To perform this operation, see Writing Data Upgrade Scripts below. Shared scripts are run only once, as compared to normal scripts, which are run once per company

- **Presynchronize Standard scripts**

  (Executed for each company account in parallel with pre-synchronization shared scripts)

  Normal scripts are run once per company and are used for company-specific clean up jobs, rebuilding indexes, or deleting company-specific data that will be regenerated later.

- **Presynchronize Final scripts**

  Used very rarely. Pre-synchronization start, shared and normal scripts manage dependencies better.

- **Postsynchronize Start scripts**

Used very rarely. Post-synchronization shared, normal and final scripts manage dependencies better.

- **Postsynchronize Shared scripts**

  (Executed once in parallel with post-synchronization normal scripts)

  Shared scripts are run once and used to update non company-specific tables.

- **Postsynchronize Standard scripts**

  (Executed for each company account in parallel with post-synchronization shared scripts.)

  Standard scripts are run once per company and are used to update company specific tables. (~90% of all scripts are of this type)

- **Postsynchronize Final scripts**

  (Executed last)

  Final scripts are used to undo changes to indexes that were made to allow duplicates using the pre-synchronization start script. Final scripts are run only once, as compared to normal scripts, which are run once per company.

- **Upgrade additional features scripts**

  Upgrade additional features scripts are used to upgrade of the non-core functionality after the functional data upgrade

Writing Data Upgrade Scripts for Target Environment

To create a script you need to create a method on the appropriate class. For example, for Microsoft Dynamics AX 6.0 the class is ReleaseUpdateDB60_<module>. You must also inform the framework how to handle the script. This is done by by providing attributes for their upgrade script methods like following:

```
[UpgradeJobTypeAttribute(Standard),
UpgradeJobTitleAttribute ( "@SYS97795"),
UpgradeJobConfiguratuionKeyAttribute (configurationKeyStr(ProjBasic)),
UpgradeJobConfiguratuionKeyAttribute (configurationKeyStr(HRMBasic)),
UpgradeJobDependsOnJobAttribute (ReleaseUpdateDB60_Proj, updateProjOnAccountPosting)]
void updateProjCategory()
{
   ProjCategory    projCategory;
   ttsbegin;
   update_recordset projCategory
       setting Active    = NoYes::Yes;
   ttscommit;
}
```

Here is the list of available attributes:

- Shared/ Standard / Start/ Final
- PreSync/PostSync/Additional
- Configuration keys associated with the script
- Script description(label)
- Does the script requires its own transaction
- (optional) Dependencies on another scripts or tables
- Table names
- Type of access to each table : Create, Read, Update, Delete

UpgradeScriptDescriptionAttribute: one attribute per method allowed
Value: String
UpgradeScriptStageAttribute: one attribute per method allowed
Values: enum values PreSync, PostSync, Additional
UpgradeScriptTypeAttribute: one attribute per method allowed
Values: enum values Standard, Shared, Start, Final
UpgradeScriptTransactionAttribute: one attribute per method
Value: enum Yes/No
UpgradeScriptTableAttribute: several attributes allowed
Value: String (table name),

enum: Yes/No (Create),

enum: Yes/No (Read),

enum: Yes/No (Update),

enum: Yes/No (Delete)

for example,

[UpgradeScriptTableAttribute(tableStr(LedgerTable), false, true, true, false)]

For legacy compatibility, Microsoft Dynamics AX supports adding upgrade scripts, by adding a line in the initPreSyncJobs or initPostSyncJobs or initAdditionalJobs method on the class. Each of these ReleaseUpdateDBxx_xxx classes contains three separate methods you can modify to schedule your jobs – initPreSyncJobs, initPostSyncJobs and initAdditionalJobs. To run the job in the pre-synchronize phase, add it to the initPreSyncJobs method, otherwise add it to the initPostSyncJobs method or to the initAdditionalJobs method for the additional feature upgrade.

This method is not recommended and is mainly preserved for legacy support. The following example shows the script templates you can use:

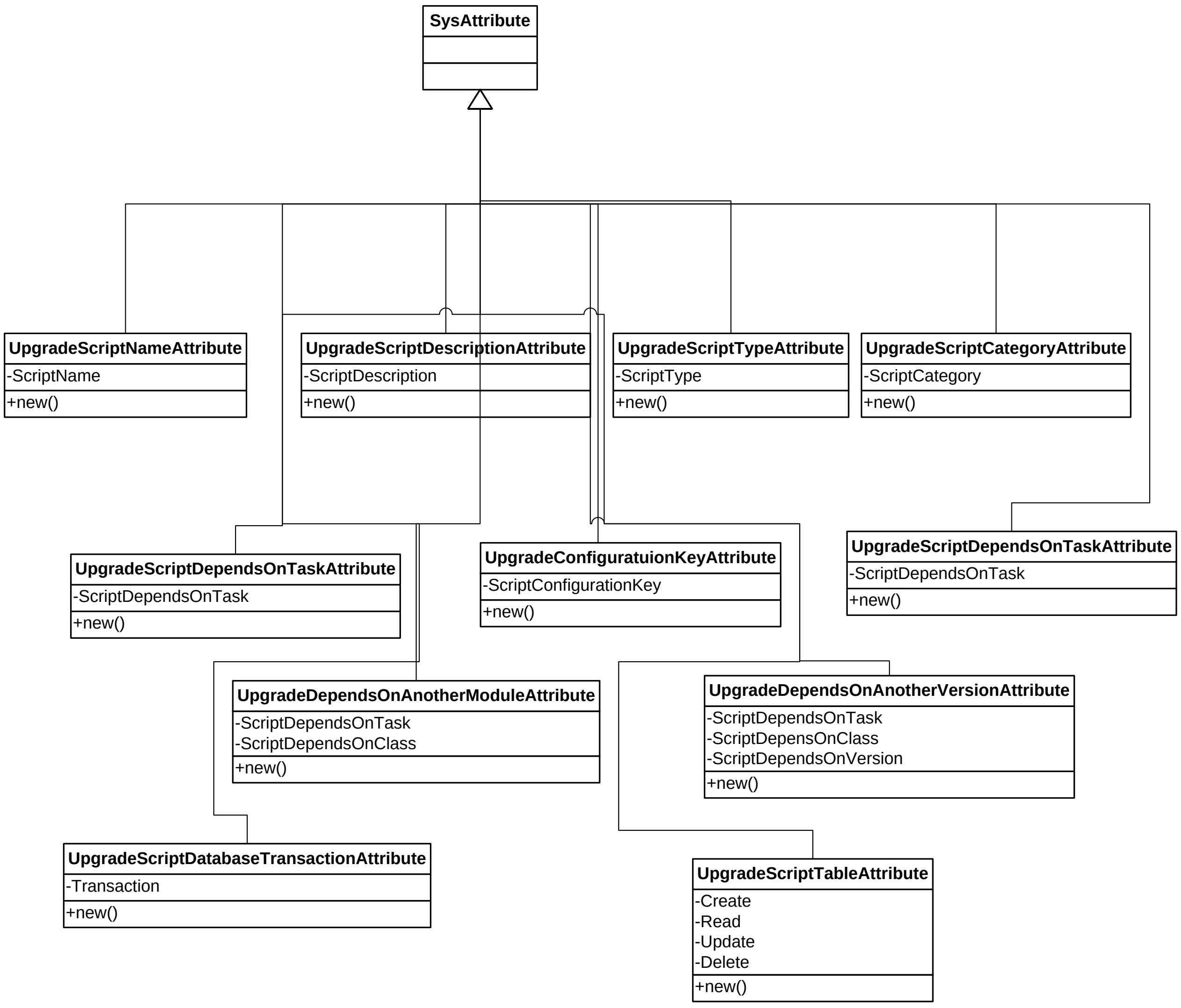
```
this.addStartJob(methodStr(<ClassName>, <MethodName>), "description", [configurationkeynum(ConfigurationKey1), ...,
configurationkeynum(ConfigurationKey1) ]);

this.addSharedJob(methodStr(<ClassName>, <MethodName>), "description", [configurationkeynum(ConfigurationKey1), ...,
configurationkeynum(ConfigurationKey1) ]);

this.addStandardJob(methodStr(<ClassName>, <MethodName>), "description",[configurationkeynum(ConfigurationKey1), ...,
configurationkeynum(ConfigurationKey1) ]);

this.addFinalJob(methodStr(<ClassName>, <MethodName>), "description",[configurationkeynum(ConfigurationKey1), ...,
configurationkeynum(ConfigurationKey1) ]);
```

The following diagram illustrates a class diagram of the attribute based upgrade model.


SysAttribute
UpgradeScriptNameAttribute
-ScriptName
+new()
UpgradeScriptDescriptionAttribute
-ScriptDescription
+new()
UpgradeScriptTypeAttribute
-ScriptType
+new()
UpgradeScriptCategoryAttribute
-ScriptCategory
+new()
UpgradeScriptDependsOnTaskAttribute
-ScriptDependsOnTask
+new()
UpgradeConfiguratuionKeyAttribute
-ScriptConfigurationKey
+new()
UpgradeScriptDependsOnTaskAttribute
-ScriptDependsOnTask
+new()
UpgradeDependsOnAnotherModuleAttribute
-ScriptDependsOnTask
-ScriptDependsOnClass
+new()
UpgradeDependsOnAnotherVersionAttribute
-ScriptDependsOnTask
-ScriptDepensOnClass
-ScriptDependsOnVersion
+new()
UpgradeScriptDatabaseTransactionAttribute
-Transaction
+new()
UpgradeScriptTableAttribute
-Create
-Read
-Update
-Delete
+new()

Upgrade script configuration keys

You can provide an optional set of configuration keys associated with an upgrade script - [*configurationkeynum (<config key name1,config key name2, ... , config key name n>]*. The script will be scheduled to run if at least one configuration key associated with script is enabled during upgrade.

You can do this by specifying attribute for your upgrade script:

UpgradeConfiguratuionKeyAttribute : several attributes allowed

(*attribute will be joined with OR; for complex OR and AND between configuration keys – upgrade script should handle it with if/else condition within upgrade script*)

Value: Configuration key

For compatibility purposes, legacy way of specifying configuration key is still supported, but not recommended:

```
this.addFinalJob(methodstr(ReleaseUpdateDB60_Administration, allowDupSysExpImpTableGroupIdx), "@SYS97945",
[configurationkeynum(Asset), configurationkeynum(Bank) ]);
```

Also, you can specify a set of configuration keys on the module level by using the setModuleConfigKey function. The module configuration key set is joined with each upgrade script configuration key set for that module.

```
this.setModuleConfigKey([configurationkeynum(ConfigurationKey1),…,configurationkeynum(ConfigurationKey1) ])
```

If you are using setModuleConfigKey, it should be called from the InitPreSyncJobs, initPostSyncJobs and InitAdditionalJobs methods separately.

Script Dependencies

You can also add dependencies between your scripts. This can be useful to avoid locking and for enforcing a logical flow of your scripts. To add a dependency, you can add appropriate attribute before your upgrade script (recommended) or include call to addDependency method in the appropriate InitXXXJobs method:

- If you have a dependency **between the scripts inside a module**, use the **UpgradeDependsOnTaskAttribute or** addDependency method.

```
this.addDependency(methodStr(<ClassName>, <MethodName>),
             methodStr(<ClassName>, <MethodName>));
```

where the first method must be executed before the second method executes.

- If the script is **dependent on another module script**, you can use the **UpgradeDependsOnModuleAttribute or** addCrossModuleDependency method to ensure a correct execution sequence between scripts placed in the different classes:

```
this.addCrossModuleDependency(classnum(<ClassName>), methodStr(<ClassName>, <MethodName>),
                 classnum <ClassName>, methodStr(<ClassName>, <MethodName>));
```

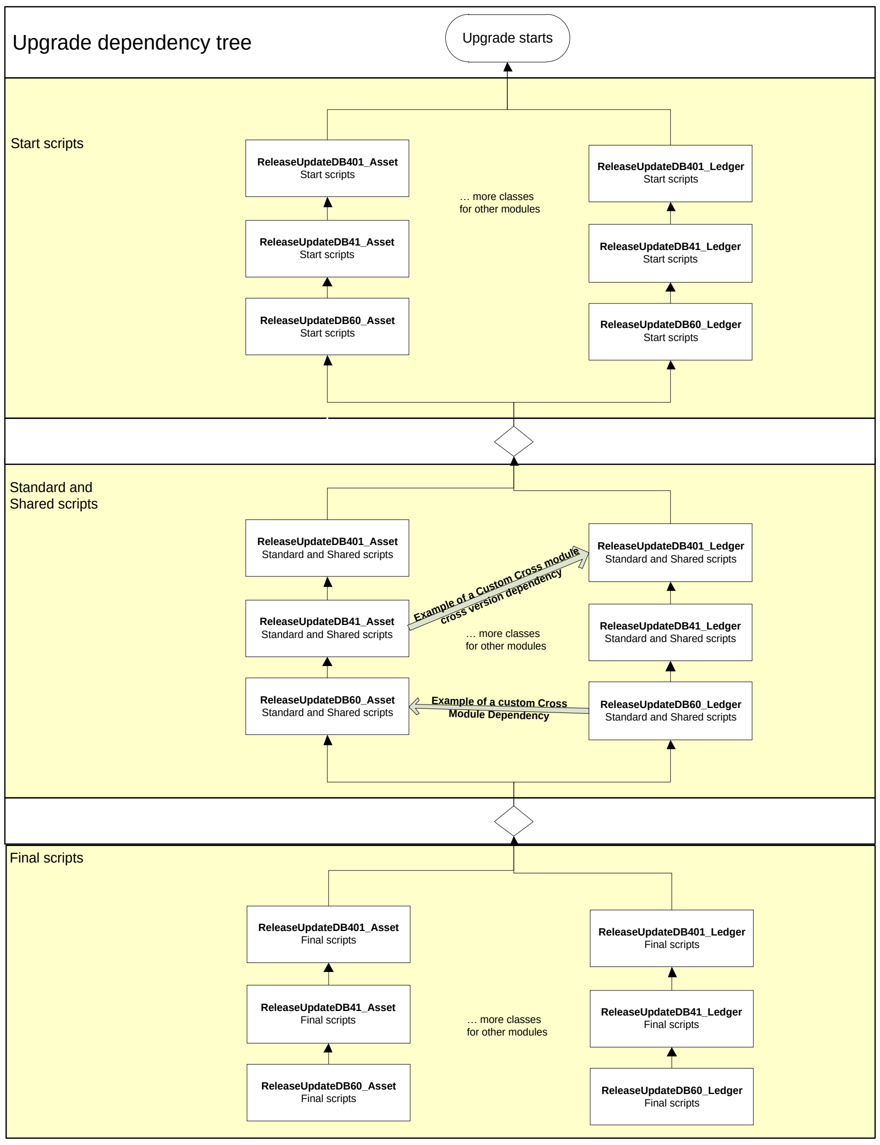
- If the script is **dependent on another module script from a previous version**, you can use the **UpgradeDependsOnVersionAttribute** or addCrossVersionModuleDependency method to ensure that the correct execution sequence between scripts placed in the different versions and modules:

```
this.addCrossVersionModuleDependency(
                classnum(<ClassName>),
                 methodStr(<ClassName>, <MethodName>),
                 SysReleaseVersion::<version>,
                classnum <ClassName>,
                methodStr(<ClassName>, <MethodName>),
                SysReleasedVersion::<version>);
```

- If there are scripts depending on your script and you want to set the dependency but do not want to change the other scripts, you can use the **UpgradeTaskDependsOnMeAttribute** or **UpgradeModuleDependsOnMeAttribute**.

4. If a script is **dependent on another script from a previous version but located in the same module**, then you do not need a dependency, the upgrade framework will provide an implicit dependency in this case.

Here is an example of the dependency tree:


Upgrade dependency tree
Upgrade starts
Start scripts
ReleaseUpdateDB401_Asset Start scripts
ReleaseUpdateDB41_Asset Start scripts
ReleaseUpdateDB60_Asset Start scripts
… more classes for other modules
ReleaseUpdateDB401_Ledger Start scripts
ReleaseUpdateDB41_Ledger Start scripts
ReleaseUpdateDB60_Ledger Start scripts
Standard and Shared scripts
ReleaseUpdateDB401_Asset Standard and Shared scripts
ReleaseUpdateDB41_Asset Standard and Shared scripts
ReleaseUpdateDB60_Asset Standard and Shared scripts
Example of a Custom Cross module cross version dependency
… more classes for other modules
Example of a custom Cross Module Dependency
ReleaseUpdateDB401_Ledger Standard and Shared scripts
ReleaseUpdateDB41_Ledger Standard and Shared scripts
ReleaseUpdateDB60_Ledger Standard and Shared scripts
Final scripts
ReleaseUpdateDB401_Asset Final scripts
ReleaseUpdateDB41_Asset Final scripts
ReleaseUpdateDB60_Asset Final scripts
… more classes for other modules
ReleaseUpdateDB401_Ledger Final scripts
ReleaseUpdateDB41_Ledger Final scripts
ReleaseUpdateDB60_Ledger Final scripts

Precautions When You Write Data Scripts Before Synchronization

Pre-synchronization data upgrade scripts are executed before the new version of Microsoft Dynamics AX Object Data (AOD) is synchronized to the Microsoft Dynamics AX database and before the data is copied over from the source system to target system. This means that the executed code will use a new version of metadata, but the database will still be empty.

The following lists tables that are synchronized during AOS startup before the upgrade checklist starts:

SysSetupLog	SysProgress	ReleaseUpdateBulkCopyTable
SysSetupCompanyLog	SysBPParameters	ReleaseUpdateBulkCopyField
SysRecordTemplateTable	SysRemoveFields	DEL_SysSetupLog
SysRecordTemplateSystemTable	SysRemoveTables	ReleaseUpdateTransformTable
SysTraceTable	SysRemoveConfig	DEL_ReleaseUpdateTransformTable
SysTraceTableSQL	SysRemoveLicense	ReleaseUpdateTransformSourceField
SysTraceTableSQLExecPlan	SysLicenseCodeSort	ReleaseUpdateTransformTargetField
SysTraceTableSQLTabRef	DocuParameters	ReleaseUpdateBulkCopyTableExceptions

SysUserLog	EPStateStore	DEL_SysUpgradeTimeZone
SysUserInfo	EPStateStoreSettings	DEL_SysUpgradeBaseTimeZone
SysUtilElementsLog	EPServerStateCleanupSetting	WorkflowWorkItemDelegationParameters
SysCompanyUserInfo	EPWebSiteParameters	WorkflowWorkItemCommentTable
SysInetCSS	EPGlobalParameters	SysUpgradeTreeNodeConflictInfo
SysInetThemeTable	SysBCProxyUserAccount	SysUpgradeParameters
SysImageTable	SRSServers	SysUpgradeOverriddenEstimates
SysPersonalization	AifWebsites	ReleaseUpdateBulkCopyParameters
LanguageTable	Currency	ReleaseUpdateBulkFlags
SysSignatureSetup	SysSQMSettings	ReleaseUpdateDiscoveryStatus
SysDataBaseLog	SysSecurityFormTable	DEL_ReleaseUpdateDictionaries
SysExceptionTable	SysSecurityFormControlTable	DEL_SqlDictionary
DEL_Batch	SysEvent	DEL_SystemSequences
DEL_BatchGroup	KMConnectionType	DEL_SysLastValue
BatchJob	SalesParmUpdate	ReleaseUpdateBulkCopyLog
BatchConstraints	SalesParmSubTable	ReleaseUpdateSpecialTableMapping
Batch	PurchParmUpdate	ReleaseUpdateSpecialFieldMapping
BatchGroup	PurchParmSubTable	ReleaseUpdateBulkRefRecIdPatch
SysServerConfig	SysVersionControlParameters	ReleaseUpdateExtendedDataTypes
SysClusterConfig	ReleaseUpdateScripts	ReleaseUpdateDataAreaOffsets
BatchServerConfig	ReleaseUpdateScriptDepende	ReleaseUpdateConfiguration
BatchJobAlerts	ReleaseUpdateJobStatus	DataArea
BatchServerGroup	ReleaseUpdateScriptsUsedTa	DEL_CompanyDomainList
BatchJobHistory	ReleaseUpdateBulkTableInfo	ReleaseUpdateConfigKey
BatchHistory	ReleaseUpdateSysDeleted	NumberSequenceDatatype
BatchConstraintsHistory	DocuOpenFile	NumberSequenceDatatypeParameterType
BatchGlobal	TimezoneInfo	SysXppAssembly
		SecurityRoleAllTasksView

For these special tables, you cannot use pre-synchronization Start scripts. If you change the field ID on one of these tables, code changes must be made directly in the \Classes\Application\syncApplTables() method.

Changes in Application classes are risky and should be made with caution. The following code illustrates changes to an Application class:

```
if (!this.isRunningMode())
{
    ttsbegin;
    if (isConfigurationkeyEnabled(configurationkeynum(CRSEGermany)))
    {
        ReleaseUpdateDB::changeFieldByName('TaxRepresentative', 41, 0, 75);
    }
    ttscommit;
}
syncTable(tablenum(CompanyInfo));
```

Best Practices for Writing Data Upgrade Scripts

Best Practice Checks

In Microsoft Dynamics AX 2010, there are now several best practice checks that help to verify the completeness of upgrade script attributes.

How An Upgrade Script Is Identified

Any method that is decorated with the following attributes is considered an upgrade script:

- UpgradeScriptDescriptionAttribute
- UpgradeScriptTypeAttribute
- UpgradeScriptStageAttribute
- UpgradeScriptTableAttribute

There are conditions that further identify a method as an upgrade script that are not considered for performance reasons when you check for best practices. During an actual upgrade, the additional conditions are considered and validated.

The following lists BPError codes when a violation occurs and the resolution.

BPError code and condition/message	Resolution
BPErrorMethodIsStatic An upgrade script cannot be a static method.	Remove the static modifier.
BPErrorMethodHasParams An upgrade script cannot accept parameters.	Remove the parameters.
BPErrorMissingReqDesc For scripts that upgrade to Microsoft Dynamics AX 2012, the UpgradeScriptDescriptionAttribute attribute is required.	Add the required attribute.
BPErrorMissingReqType For scripts that upgrade to Microsoft Dynamics AX 2012, the UpgradeScriptTypeAttribute attribute is required.	Add the required attribute.
BPErrorMissingReqStage For scripts that upgrade to Microsoft Dynamics AX 2012, the UpgradeScriptStageAttribute attribute is required.	Add the required attribute.
BPErrorMissingReqTable One or more UpgradeScriptTableAttribute attributes are required for all upgrade scripts. In addition, any table referenced in the method must have a corresponding UpgradeScriptTableAttribute attribute.	Add the required attribute for any tables referenced by the upgrade script. The following upgrade scripts are exempt from this requirement: • deleteDuplicatesUsingIds • deleteDuplicatesUsingNames • indexAllowDup

	• indexAllowNoDup
BPErrorTableNotFound The table name specified in the UpgradeScriptTableAttributes attribute is not valid. Use the tableStr(<table>) method to catch this issue at compile time.	Ensure the table specified exists and that the name is typed correctly.
BPErrorConfigurationKeyNotFound The configuration key specified in UpgradeScriptConfigKeyAttribute attribute is not valid. Use the configurationKeyStr(<configurationKey>) method to catch this issue at compile time.	Ensure the configuration key exists and that the name is typed correctly.
BPErrorClassNameNotFound The class name specified in UpgradeDependsOnModuleAttribute attribute or UpgradeDependsOnVersionAttribute attribute is not valid. Use the classId(<class>) method to catch this issue at compile time.	Ensure the class exists and that the name is typed correctly.
BPErrorMethodNameNotFound The method name specified in UpgradeDependsOnModuleAttribute attribute, UpgradeDependsOnTaskAttribute attribute, or UpgradeDependsOnVersionAttribute attribute is not valid. Use the methodStr(<class>, <method>) method to catch this issue at compile time.	Ensure the method exists and that the name is typed correctly.
BPErrorInvalidScriptVersion The module for an upgrade script must have a version macro defined in the class declaration: #define.version(sysReleasedVersion::v60)	Ensure the version macro is defined and that it is not Unknown or vNext.

Transaction and Idempotency

It is an important requirement that each data upgrade script be *idempotent*. Idempotent means that if the execution fails, it must be able to execute successfully with the desired results upon reexecution.

The data upgrade framework guarantees idempotency by enclosing each script within a transaction, ensuring that the script is only executed once. Although this is a simple and robust way to ensure idempotency it results in a performance decline when an upgrade script has complex logic in a loop on a large table. In Microsoft Dynamics AX 4.0, this mechanism is optional such that an individual script can be run without the transaction at the highest level. When this option is chosen, the individual script must implement its own idempotency logic.

Another important consideration for implementing idempotency is that you can upgrade from many different versions. For example, if you write an upgrade script for SP2 of version N, when version N+1 is shipped, your customers are upgrading from both Version N SP1 and Version N SP2. This means that some customers already are upgraded and others are not. If your upgrade script is idempotent, you can just reuse it for the upgrade to version N+1.

If an upgrade script contains an error, it is easier to resolve the problem if the script is idempotent.

Coding Best Practices

Indicating Progress

To supply progress status, you can use a simplified version operation progress by calling:

```
this.tableProgress(<tableId>);
```

Include the table-ID for the table you have just updated. This should only be called once in each outermost loop (even if you are updating several tables in the inner loops).

Documenting Scripts

You should include meaningful comments in each data upgrade script to explain the functionality of the script.

Deleting a Table or Field from the Data Model

It is not possible to delete data from the data model as this would be the equivalent of deleting customer data. This also applies to fields that were never used or fields that appear in the UI (unless they are temporary). Removing a field or table requires careful planning and execution as follows:

1) Prefix the name of the item to be removed with "DEL_" and move it to the upgrade model, using the following steps:

    a) For fields and indexes:

        - Rename them in the AOT using F2 or the PropertySheet. The field/index will automatically be moved into the upgrade model in the next build.
        - In the rare case where you want the DEL_ fields/indexes to remain in the Foundation, you can add the table to the exclusion list here: //depot/main/source/application/CombineXPOs/NoDelExclusionList.txt.

          This should only be used when production code needs the tables in a post-upgrade scenario.

    b) For tables, enums and extended data types:

        - Turn-off version control.
        - Rename the element in the AOT using F2 or the PropertySheet.
        - Right-click the element and click **Move to model**. Click **Foundation Upgrade**, click **Ok**.
        - In CoreXT type:

          CD “source\application\sys\data dictionary”

          For tables:

          SD integrate “tables\<tablename>.xpo” “..\..\sysupgrade\data dictionary\tables\DEL_<tablename>.xpo”

          SD delete “tables\<tablename>.xpo”

          For enums:

SD integrate “base enums\<enumname>.xpo” “..\..\sysupgrade\data dictionary\base enums\DEL_<enumname>.xpo”

SD delete “base enums\<enumname>.xpo”

For extended data types:

SD integrate “extended data types\<typename>.xpo” “..\..\sysupgrade\data dictionary\extended data types\DEL_<typename>.xpo”

SD delete “extended data types\<typename>.xpo”

i) Turn-on version control again

2) Set the configuration property to: "SysDeletedObjectsXX" where XX is the next version, for example "60" (for Microsoft Dynamics AX 2012)
3) Implement the upgrade script that will transform the data into the new data model. Verify the current model is **Foundation Upgrade** when you create the class.
4) Test the upgrade script
5) Benchmark the upgrade script
6) *(New for Interim Upgrade):* **When deleting a field from a table**, do not delete the table or field permanently from the AOT. They need to stay in the source until the release, where they are deleted is no longer supported by upgrade. For example, if a Microsoft Dynamics AX 4.0 field is renamed in Microsoft Dynamics AX 4.1 as a DEL_field, it needs to stay in the source until Microsoft Dynamics AX 2012.
7) When deleting a table, after you rename the table to DEL_TableName then the upgrade process will copy the data from the previous version to this Del_ table. It is not a good practice to have two copies of the same table. The old table & field can be deleted after creating the DEL_ table & fields. These DEL_ tables should be tied to SysDeleted configuration keys.

Unique Indexes

It is important that the database can synchronize without errors when the customer upgrades. Three scenarios require special attention when dealing with index changes:

- Removing a field from a unique index
- Adding a new unique index
- Making a non-unique index unique, (setting the AllowDuplicates property to false)

All these scenarios make an index more restrictive and will cause the synchronization to fail if not handled properly.

The easiest solution is to delete the data that collides with the index. This should only be done in situations where it doesn't make sense to keep the duplicate records. This is performed using the following steps:

- Create a start pre-synchronization upgrade script. This will change the index to allow duplicates:

```
DictIndex dictIndex = new
        DictIndex(TableNum(<TableName>),indexNum(<TableName>,<IndexName>));
;
ReleaseUpdateDB::indexAllowDup(dictIndex);
```

- Create a normal upgrade script. This will move the data according to the new data model.
- Create a final post-synchronization upgrade script. This will change the index to not allow duplicates:

```
DictIndex dictIndex = new
        DictIndex(TableNum(<TableName>),indexNum(<TableName>,<IndexName>));
;
ReleaseUpdateDB::indexAllowNoDup(dictIndex);
```

Consideration for date effective table in upgrade script

Some Microsoft Dynamics AX 2010 tables with date effective indexes that are disabled then re-enabled after synchronization (see section 'Unique Indexes') will also require that the property 'ValidTimeStateKey' is also set to 'Yes' along with re-enabling of the index. To set the 'ValidTimeStateKey' property on the index, follow this pattern:

```
public void allowNoDupAssetParmDeprRtsDEDateEffcIdx()
{
    DictIndex dictIndex = new DictIndex(tablenum(AssetParametersDeprRates_DE),
     indexnum(AssetParametersDeprRates_DE, DateEffcIdx));
    dictIndex.modify(true, false, true);
    dictIndex.setAlternateKey(true, true);
    // Set the ValidTimeStateMode to the appropriate value for your index;
    // i.e. what the value was before the index was disabled.
    dictIndex.setValidTimeStateKey(true, ValidTimeStateMode::Gap, true);
    appl.dbSynchronize(dictIndex.tableid(), false);
}
```

Fixing table/field mapping errors

Preventing Copying of Table Data

Situations may occur where you may not want to copy the contents of a table from the source environment to the target environment, for example, if the table will be used in the new version (but not if the table has become obsolete). This can be useful when the contents of the table are auto-generated. To do this, create a pre-synchronization shared script using the pattern:

```
ReleaseUpdateDB::addBulkCopyTableException(tableStr(<table>), ReleaseUpdateBulkCopyTableExceptionType::DoNotCopy);
```

Another option is to create a pre-synchronization standard script using the delete_from construct.

Applying Field Options

Situations may occur when you need to perform operations on specific table fields during copy & sync. To do this, create a pre-synchronization shared script using the following pattern:

```
ReleaseUpdateDB::addBulkCopyFieldOption(tableStr(<table>), fieldStr(<table>, <field>),
ReleaseUpdateBulkCopyFieldOption::LTrimTarget);
```

ReleaseUpdateBulkCopyFieldOption supports the following elements:

LTrimTarget

Applies the Sql LTRIM operator to the source value as it is copied to the target, allowing previously right aligned data to be left-aligned in the target environment.

DoNotCopy

Removes the specified field from the source table's SELECT statement during the bulk copy phase, preventing the field's data from being copied to the target system.

Truncating a Table During Copy & Sync

Situations may occur when you need to truncate the contents of a table in the target environment during copy& sync.

- **If the table is not empty at the time of mapping, it is considered a mapping error**
- **If the table is not empty, the data will NOT be copied from the source system**
- **Watch for tables where data can be created accidentally:**
- **SalesParameters::find()**
- **InventDim::inventDimIdBlank()**
- **DO NOT ignore this error**

To do this, create a pre-synchronization shared script using the pattern:

```
 ReleaseUpdateDB::addBulkCopyTableException(tableStr(<table>),
ReleaseUpdateBulkCopyTableExceptionType::TruncateOnCopy);
```

This will truncate the data on the target table and then copy the source table data.

Example:

```
[
UpgradeScriptDescriptionAttribute("@SYS115198"),
UpgradeScriptStageAttribute(ReleaseUpdateScriptStage::PreSync),
UpgradeScriptTypeAttribute(ReleaseUpdateScriptType::SharedScript),
UpgradeScriptTableAttribute(tableStr(ReleaseUpdateBulkCopyTableExceptions),
true,true,false,false)
]
public void deletePrePopulatedData()
{
    ReleaseUpdateDB::addBulkCopyTableException(tableStr(DirNameSequence),
        ReleaseUpdateBulkCopyExceptionType::TruncateOnCopy);
}
```

Specifying the Shadow/Source Table JOIN Type

By default, shadow tables are joined to the source table during the bulk copy operation using a simple 'JOIN'. Specify a different join type using the following pattern in the preprocessing script:

```
ReleaseUpdateTransformDB.setShadowTableJoinType(_shadowJoinType);
```

_shadowJoinType is of type ReleaseUpdateJoin and can be one of the following values:

```
ReleaseUpdateJoin::Join              // 'JOIN' (default)
ReleaseUpdateJoin::FullOuterJoin     // 'FULL JOIN'
ReleaseUpdateJoin::LeftOuterJoin     // 'LEFT JOIN'
ReleaseUpdateJoin::RightOuterJoin    // 'RIGHT JOIN'
```

Mapping a Table with name and fieldid or tableid changed

When a table or field's name is changed, and there is a possible conflict between new and existing table or field names, in order to preserve the table and its data, you must call the following methods in a pre-synchronization Start script:

static void void ReleaseUpdateBulkCopyMap**.***addTableNameMapping(tableName _oldTableName, tableName _newTableName, NoYes _system = NoYes::No)*

static void ReleaseUpdateBulkCopyMap**.***addFieldNameMapping(tableName _oldTableName, fieldName _oldFieldName, tableName _newTableName, fieldName _newFieldName, NoYes _system = NoYes::No)*

FIELDTYPE of the sourface and target mapping should match. If there is a mismatch in fieldtype the framework do not copy the data.

The following is an xample of mapping a field:

```
[
UpgradeScriptDescriptionAttribute("@SYS53630"),
UpgradeScriptConfigKeyAttribute(configurationkeystr(LogisticsBasic)),
UpgradeScriptStageAttribute(ReleaseUpdateScriptStage::PreSync),
UpgradeScriptTypeAttribute(ReleaseUpdateScriptType::SharedScript),
UpgradeScriptTableAttribute(tablestr(InventDim),    false,  false,  true,   false)
]
public void updateFieldMappingInventDim()
{
    ReleaseUpdateBulkCopyMap::addFieldNameMapping(
        'Shadow_InventDim', 'ShadowRecId',
        tablestr(InventDim), fieldstr(InventDim, RecId));
}
```

Mapping a Table with Table ID or Field ID Changed

When a table or field's id is changed, no upgrade scripts are needed.

However, for legacy compatibilty, we support the following functions to accommodate ID of a table or field change:

ReleaseUpdateDB::ChangeTableID (for table ID changes)

ReleaseUpdateDB::ChangeFieldID (for field ID changes)

You can also use the following methods to address tables and fields by name:

ReleaseUpdateDB::ChangeTableByName (for table ID changes)

ReleaseUpdateDB::ChangeFieldByName (for field ID changes)

For a few special tables that are listed in the "Precautions When You Write Data Scripts Before Synchronization" section, you cannot use the pre-synchronization Start script. Please refer to that section for more details and code samples.

Deleting Configuration Keys

Configuration keys should **not** be deleted. Configuration key changes are not handled by code upgrade, therefore, changes will not be detected at code upgrade time. If a customization has been set up to use a Microsoft Corporation shipped configuration key in custom tables, and if the configuration key is deleted, the table will be lost during synchronization.

Referencing Number Sequences within upgrade scripts

If a number sequence has to be referenced within a X++ upgrade script, it is recommended to code that reference as a separate method insetad of hardcoding it within the script itself, which will make the process of changing it easier for a user running the upgrade

```
private NumberSequenceReference numberSequenceReference_SQ()
{;
    return NumberSeqReference::findReference(extendedTypeNum(SQ));
}
```

Later in the upgrade script, you can use that method to get the actual number sequence

```
num = NumberSeq::newGetNum (this.numberSequenceReference_SQ(), false);
salesQuotationTable.QuotationId = num.num();
```

If the number sequence you are using is for an an extended data type that is new to Microsoft Dynamics AX 2012, or if there is the possibility that it has not been previously instantiated within the current scope , you must first instantiate the number sequence prior to using it. Although this is normally an administrator task, which allows the administrator to override some default settings, this can be done programatically from the upgrade script by calling:

```
NumberSequenceTable::autoCreate (extendedTypeNum(SQ), [scope]);
```

At the top of an upgrade script add to the list of attributes:

UpgradeScriptTransactionAttribute(false)

This ensures that upgrade / batch do not automatically wrap the script in a transaction.

While debugging, if you decide to add or change this attribute you need to:

1) Generate IL
2) Rerun upgrade from the step of prioritize dependencies

Failure to do these steps will mean the code you are executing in upgrade will not capture your changes.

Performance Guidelines

Performance is a critical piece of the upgrade process and requires that you think about each line in your script. Most companies will perform this task over a weekend, so the entire upgrade process must be able to be completed within 48 hours. The actual update will typically be performed between Friday night and Monday morning. In addition, prior to running the upgrade process on a live system, the upgrade process is tested several times on a test system.

In addition to the following considerations, please read Performance Improvement Options to determine which apply to your upgrade scripts:

- Monitor and minimize the number of client/server calls.
- Use record set functions whenever possible.
- Break down your scripts into smaller pieces. For example, do not upgrade two independent tables in the same script even if there is a pattern in the way the scripts work. This is because:
    - Each script, by default, runs in one transaction (=one rollback segment). If the segment becomes too large, the database server will start swapping memory to disk, and the script will slowly come to a halt.
    - Each script can be executed in parallel with other scripts.
- Partial commits can only be used out of the box in one situation; this is when the table to upgrade is large and contains a discriminator that can be used to split the script into several scripts. For example, update all "Open" in one script and all "Closed" in another. The scripts should be set up to be dependant on each other to avoid locking problems. (see point below regarding database lock contention)
- Take care when you sequence the scripts. For example, do not update data first and then delete it afterwards.
- Be careful when calling normal business logic in your script. Normal business logic is not usually optimized for upgrade performance. For example, the same parameter record may be fetched for each record you need to upgrade. The parameter record is cached, but just calling the Find method takes an unacceptable amount of time. For example, the kernel overhead for each function call in Microsoft Dynamics AX is 5 ms. Usually 10-15 ms will elapse before the Find method returns (when the record is cached). If there are a million rows, two hours will be spent getting information you already have. The solution is to cache whatever is possible in local variables.
- Run benchmarking on your script using large datasets to verify your performance is acceptable.

- If database lock contention prevents the data upgrade process from scaling up with multiple batch clients running in parallel, consider disabling the transaction in the framework and ensuring idempotency by one of the following:
    - Using an existing field/condition that can check if the table/record has been updated
    - Adding new fields to track upgrade status
    - Using the primary key as ordering columns and recording the last row that was updated
- Use index tunint. Create indexes to speed up the upgrade and possibly remove them after the upgrade. Setting up a configuration key to SysDeletedObjects<version> can help you ensure that the index is deleted after the upgrade is finished.
- If there is no business logic in the script, rewrite the script to issue a direct query to bulk update the data. To write Direct SQL queries, see Appendix 2: Guidelines for Writing Direct SQL in Upgrade Scripts.

Performance Improvement Options

This section provides information to improce performance of upgrade scripts.

Using the Set-based Operators Delete_From, Update_RecordSet and Insert_SecordSet

If the script performs inserts, updates, or deletes within a loop, you should consider changing the logic to use one of the set-based statements. If possible, use these set options to perform a single set-based operation.

When using set-based operations:

- With Insert_RecordSet you cannot use a literal or function call in the field list. This operation does not handle configuration keys so special care is required.
- With Update_RecordSet you cannot perform inner or left outer joins.
- Set based statements do not support memo fields.

Please refer to Speeding Up SQL Operations and Maintain Fast SQL Operations on MSDN for the list and syntax of set based operations available in Microsoft Dynamics AX 2012.

Example:

Before performance improvement:

```
while select inventTable
  where inventTable.ItemType == ItemType::Service
{
this.tableProgress(tablenum(InventTable));
delete_from inventSum where inventSum.ItemId == inventTable.ItemId;
}
```

After performance improvement:

```
delete_from inventSum
  exists join inventTable
  where inventTable.ItemId    == inventSum.ItemId
  && inventTable.ItemType  == ItemType::Service
```

Calling skipDataMethods and skipDatabaseLog Before Calling Update_RecordSet or Delete_From

If your script runs delete_from or update_from on a large table where the delete() or update() methods of the target table have been overwritten, the bulk database operation will fall back to record-by-record processing. To prevent this, call the skipDataMethods(true) method to cause the update() and delete() methods to be skipped. Also, you can call the skipDatabaseLog(true) method to improve performance.

Example:

```
taxExchRateAdjustment.skipDataMethods(true);
taxExchRateAdjustment.skipDatabaseLog(true);

update_recordset taxExchRateAdjustment
  setting GovernmentExchRate = taxExchRateAdjustment.UseGovtBankRate
  where taxExchRateAdjustment.UseGovtBankRate == NoYes::Yes;
```

Using RecordInsertList Class to Batch Multiple Inserts

If the business scenario cannot be written as insert_recordset, consider using the RecordInsertList class to batch multiple inserts to reduce network calls. This operation is not as fast as insert_recordset, but is faster than individual inserts in a loop.

Example:

```
rilAssetTransMerge = new RecordInsertList(tablenum(assetTransMerge));
while select assetTrans
{
        if (!AssetTransMerge::exist(AssetBookType::ValueModel,assetTrans.RecId))
        {
    assetTransMerge.AssetId = assetTrans.AssetId;
    assetTransMerge.AssetGroup = assetTrans.AssetGroup;
    ...
    rilAssetTransMerge.add(assetTransMerge);
    }
}
rilAssetTransMerge.insertDatabase();
```

Optimizing X++ logic

To optimize X++ logic, apply the following rules:

- Minimize the amount of time spent in the X++ interpreter
- For database related code, ensure SQL is fully utilized by including where conditions, for example, to check for null values, using joins across tables
- Use set-based updates and inserts instead of record-based updates and inserts

For more information on Financial Dimensions, download the whitepaper called Implementing the Account and Financial Dimensions Framework.

The following examples illustrate common mistakes in code:

```
while select forupdate projForecastCost
where ! projForecastCost.TransId
```

```
{
    if (! projForecastCost.TransId)
    {
        numberSeq = NumberSeq::newGetNum(ProjParameters::numRefProjTransIdBase());
        .....
    }
}
```

The where !projForecastCost.TransId is already checked by SQL. There is no need to check the value again. The entire statement if (! projForecastCost.TransId) should be removed.

```
void someFunc()
{
   while select custTable
   {
       if (custNum != 0)
       {
          dosomething()
       }
   }
}
```

Again, this is not good coding practice. SQL can perfom this operation for you.

Rewrite the above function as:

```
void someFunc()
{
   while select custTable where custNum != 0
   {
          dosomething()
   }
}
```

Below is another example of wasting CPU cycles in the X++ interpreter:

```
private ledgerSRUCode somefunc(AccountNum _accountNum)
{
.....
    if (auxAccountNum  >= '1910'  &&
        auxAccountNum  <= '1979')
    {
           ledgerSRUCode = '200';
    }

    if ((auxAccountNum >= '1810'  && auxAccountNum  <= '1819') ||
        (auxAccountNum  >= '1880'  && auxAccountNum  <= '1889'))
    {
           ledgerSRUCode = '202';
    }
     ...... and so on

     return ledgerSRUCode;
}
```

This function only gets the ledgerSRU. So, when this is done, you should exit the function and not execute the if statements. Also, if you are aware of the most likely results, test for these most likely options early in your code.

Below is a corrected version:

```
private ledgerSRUCode someFunc(AccountNum _accountNum)
{
.....

    if (auxAccountNum  >= '1910'  &&
            auxAccountNum  <= '1979')
    {
            return '200';
    }

    if ((auxAccountNum >= '1810'  && auxAccountNum  <= '1819') ||
        (auxAccountNum  >= '1880'  && auxAccountNum  <= '1889'))
    {
                return '202';
    }
     ...... and so on

}
```

Appendix 1: Guidelines for Writing Direct SQL in Upgrade Scripts

Using Set-Based Updates in X++

Whenever possible, set-based updates should be used in place of row-based updates. Set-based updates have a partial implementation in X++ as insert_recordset, update_recordset, and delete_from. You can implement set-based operations in X++ when:

- An update involves data or references to a single table only. In other words, the data to be updated in a table is not derived from another column. For example:

```
while select forupdate some_table where some_table.some_column == some_value
    {
        some_table.some_column = new_value;
        some_table.doUpdate();
    }
```

Can be rewritten in X++ as:

```
Some_table st;
Update_recordset st
Setting some_column == new_value
Where st.some_column = some_value;
```

If the update method is overridden, the update_recordset will change into a row-by-row update, executing the update code for each row. You can prevent this by using the skipDataMethod operator. Refer to Calling skipDataMethods and skipDatabaseLog Before Calling Update_RecordSet or Delete_From for more details.

- An update_recordset or delete_from that includes in its selection criteria a check for existence or absence of data in the same or different table. In X++ these can be implemented directly using the EXISTS Join or NOT EXISTS Join.

  For example:

```
while select SalesBasketId from salesBasket
    where salesBasket.CustAccount == guestAccount
   {
      delete_from salesBasketLine
       where salesBasketLine.SalesBasketId ==
           salesBasket.SalesBasketId;
   }
```

Can be rewritten as:

```
delete_from salesBasketLine
exists join salesBasket
where salesBasket.SalesBasketId == salesBasketLine.SalesBasketId
   && salesBasket.CustAccount  == guestAccount;
```

Executing Direct SQL from X++

How to Execute Direct SQL for X++

- If Direct SQL code is executed using X++, it requires checking for Code Access Security.as follows:

  In the variable definition section, add:

```
SqlStatementExecutePermission   permission;
;
```

In the code section, add:

```
stmtString = < SQL Statement >;
   stmt = con.createStatement();
   permission = new SqlStatementExecutePermission( stmtString );
   permission.assert();
stmt.executeUpdate(stmtString);
   // the permissions needs to be reverted back to original condition.
CodeAccessPermission::revertAssert();
```

- Direct SQL stored procedures are executed using X++ as shown in the following example:

```
str sql;
str dataAreaId;
Connection conn;
SqlStatementExecutePermission permission;
;
dataAreaId = curExt();
sql =  = 'execute <StoredProcName> \' + dataAreaId + '\' \'' + numSeq + '\'';
permission = new SqlStatementExecutePermission(sql);
conn =  new Connection();
permission = new SqlStatementExecutePermission(sql);
permission.assert();
conn.createStatement().executeUpdate(sql);
// the permissions needs to be reverted back to original condition.
CodeAccessPermission::revertAssert();
```

Best Practices Warning when Executing Direct SQL

Executing Direct SQL is a deviation from Best Practices recommendations, so, whenever Direct SQL is executed, the X++ compiler will flag it as a best practice error. To suppress this warning, before the stmt.executeUpdate(stmtString) statement you will need to place the following comment indicating that this is a known deviation from best practices:

```
//BP Deviation Documented
```

The code will be changed to:

```
stmtString = < SQL Statement >;
   stmt = con.createStatement();
   permission = new SqlStatementExecutePermission( stmtString );
   permission.assert();
   // BP Deviation Documented
```

```
    stmt.executeUpdate(stmtString);
    CodeAccessPermission::revertAssert();
```

Using Utility Functions to Execute Direct SQL

Two new methods, statementExeUpdate() and statementExeQuery(), have been added to the ReleaseUpdateDB class. They can be used to run any Direct SQL statements in ReleaseUpdateDB based classes. For security reasons, these functions do not have CAS assert() or revertAssert() methods, these should be called by the caller. See the code example in Stored Procedure and function Guidelines for ReleaseUpdateDB::statementExeUpdate and ReleaseUpdateDB::statementExeQuery use.

Documenting Direct SQL

For debugging and maintenance purposes, always put the resulting direct SQL statement as a comment before the code that performs the string construction.

Using Table Names in Direct SQL

Use ReleaseUpdateDB::backendFieldName and ReleaseUpdateDB::backendTableName to look up the actual table name in the database. These methods use the correct look up procedure:

```
new DictTable(TableNum(<sometable>)).name(DbBackend::Sql)
new DictField(TableNum(<sometable>),FieldNum(<someTable>,<somefield>)).name(DbBackend::Sql)
```

Adding Literals in Direct SQL

It is important for security, amongst other advantages, to pass parameters into the Direct SQL statement. For example, when creating Direct SQL code there are several scenarios where you will need to add literal values to the SQL statement. The most common examples are data area identification and empty date strings. These scenarios are handled by the following examples:

```
    /*   UPDATE PROJTRANSPOSTINg
          SET EMPLITEMID = PET.EMPLID,
              CATEGORYID = PET.CATEGORYID,
              PROJTYPE = PT.TYPE,
              QTY = PET.QTy
         FROM PROJTRANSPOSTING PTP, PROJEMPLTRANS PET, PROJTABLE Pt
         WHERE PTP.TRANSID = PET.TRANSId
           AND PTP.PROJTRANSTYPE = 2
           AND PET.PROJID = PT.PROJId
           AND PTP.DATAAREAID = N'xyz' AND PET.DATAAREAID = N'xyz' AND PT.DATAAREAID = N'xyz' */

      sqlStmt  = strfmt('UPDATE %1', #T(ProjTransPosting));
      sqlStmt += strfmt('  SET %1 = %2, %3 = %4, %5 = %6, %7 = %8',
          #F(ProjTransPosting, EmplItemId),  #AF(ProjEmplTrans, EmplId),
          #F(ProjTransPosting, CategoryId),  #AF(ProjEmplTrans, CategoryId),
          #F(ProjTransPosting, ProjType),    #AF(ProjTable, Type),
          #F(ProjTransPosting, Qty),         #AF(ProjEmplTrans, Qty));
      sqlStmt += strfmt(' FROM %1 %2, %3 %4, %5 %6',
          #T(ProjTransPosting),  #A(ProjTransPosting),
          #T(ProjEmplTrans),     #A(ProjEmplTrans),
          #T(ProjTable),         #A(ProjTable));
      sqlStmt += strfmt(' WHERE %1 = %2 AND %3 = %4 AND %5 = %6 AND %7 = %8 AND %9 = %10 AND %11 = %12',
          #AF(ProjTransPosting, TransId),        #AF(ProjEmplTrans, TransId),
          #AF(ProjTransPosting, ProjTransType),  int2str(enum2int(ProjTransType::Hour)),
          #AF(ProjEmplTrans, ProjId),            #AF(ProjTable, ProjId),
          #AF(ProjTransPosting, DataAreaId),     sqlSystem.sqlLiteral(projTransPosting.DataAreaId),
            #AF(ProjEmplTrans, DataAreaId),        sqlSystem.sqlLiteral(projEmplTrans.DataAreaId),
            #AF(ProjTable, DataAreaId),          sqlSystem.sqlLiteral(projTable.DataAreaId));

/*
UPDATE SALESLINE
SET SHIPPINGDATEREQUESTED =
(   SELECT MAX(DATEEXPECTED) FROM INVENTTRANS
    WHERE INVENTTRANS.DATAAREAID = INVENTTRANS.DATAAREAID
    AND SALESLINE.INVENTTRANSID = INVENTTRANS.INVENTTRANSID
    AND INVENTTRANS.DATEEXPECTED <> '1900-01-01')
WHERE SHIPPINGDATEREQUESTED = '1900-01-01'
AND DATAAREAID = SALESLINE.DATAAREAID
AND EXISTS
( SELECT DATEEXPECTED
FROM INVENTTRANS
WHERE INVENTTRANS.DATAAREAID = N'ext'
AND SALESLINE.INVENTTRANSID = INVENTTRANS.INVENTTRANSID
AND INVENTTRANS.DATEEXPECTED <> '1900-01-01')
*/
        sqlStmt  = 'UPDATE '         + dictTable_SalesLine.name(DbBackend::Sql);
        sqlStmt += ' SET '           + dictTable_SalesLine.fieldName(fieldnum(SalesLine,ShippingDateRequested),DbBackend::Sql);
        sqlStmt += ' = ( SELECT MAX(' + dictTable_InventTrans.fieldName(fieldnum(InventTrans,DateExpected),DbBackend::Sql);
        sqlStmt += ') FROM '         + dictTable_InventTrans.name(DbBackend::Sql);
        sqlStmt += ' WHERE '         + dictTable_InventTrans.name(DbBackend::Sql);
        sqlStmt += '.'               + dictTable_InventTrans.fieldName(fieldnum(InventTrans,DataAreaId),DbBackend::Sql);
        sqlStmt += ' = '             + sqlSystem.sqlLiteral(inventTrans.DataAreaId);
        sqlStmt += ' AND '           + dictTable_SalesLine.name(DbBackend::Sql);
        sqlStmt += '.'               + dictTable_SalesLine.fieldName(fieldnum(SalesLine,InventTransId),DbBackend::Sql);
        sqlStmt += ' = '             + dictTable_InventTrans.name(DbBackend::Sql);
        sqlStmt += '.'               + dictTable_InventTrans.fieldName(fieldnum(InventTrans,InventTransId),DbBackend::Sql);
        sqlStmt += ' AND '           + dictTable_InventTrans.name(DbBackend::Sql);
        sqlStmt += '.'               + dictTable_InventTrans.fieldName(fieldnum(InventTrans,DateExpected),DbBackend::Sql);
        sqlStmt += ' <> '                + sqlSystem.sqlLiteral('1900-01-01') + ')';
        sqlStmt += ' WHERE '         + dictTable_SalesLine.fieldName(fieldnum(SalesLine,ShippingDateRequested),DbBackend::Sql);
        sqlStmt += ' = '             + sqlSystem.sqlLiteral('1900-01-01');
        sqlStmt += ' AND '           + dictTable_SalesLine.fieldName(fieldnum(SalesLine,DataAreaId),DbBackend::Sql);
        sqlStmt += ' = '             + sqlSystem.sqlLiteral(salesLine.DataAreaId);
        sqlStmt += ' AND EXISTS';
        sqlStmt += ' (SELECT '       + dictTable_InventTrans.fieldName(fieldnum(InventTrans,DateExpected),DbBackend::Sql);
        sqlStmt += ' FROM '          + dictTable_InventTrans.name(DbBackend::Sql);
        sqlStmt += ' WHERE '         + dictTable_InventTrans.name(DbBackend::Sql);
        sqlStmt += '.'               + dictTable_InventTrans.fieldName(fieldnum(InventTrans,DataAreaId),DbBackend::Sql);
        sqlStmt += ' = '             + sqlSystem.sqlLiteral(inventTrans.DataAreaId);
        sqlStmt += ' AND '           + dictTable_SalesLine.name(DbBackend::Sql);
        sqlStmt += '.'               + dictTable_SalesLine.fieldName(fieldnum(SalesLine,InventTransId),DbBackend::Sql);
        sqlStmt += ' = '             + dictTable_InventTrans.name(DbBackend::Sql);
        sqlStmt += '.'               + dictTable_InventTrans.fieldName(fieldnum(InventTrans,InventTransId),DbBackend::Sql);
        sqlStmt += ' AND '           + dictTable_InventTrans.name(DbBackend::Sql);
        sqlStmt += '.'               + dictTable_InventTrans.fieldName(fieldnum(InventTrans,DateExpected),DbBackend::Sql);
        sqlStmt += ' <> '                        + sqlSystem.sqlLiteral('1900-01-01') + ')';
```

Specifying DataAreaId in Where-Clauses

The DataAreaId to be used in a where-clause may not be equal to the current company code returned by curExt(). Therefore, curExt() should not be used to build the query string.

Because of the virtual company feature, it cannot be guaranteed that two tables in any join statement will fetch its data using the same DataAreaId. In this instance a Where clause should not use the following predicate: A. DATAAREAID = B.DATAAREAID.

The DataAreaId field should always be compared to a literal or a placeholder.

The following statement may not always work correctly:

```
DELETE FROM INVENTSUM
WHERE DATAAREAID=N'dmo' AND
EXISTS (SELECT 'x' FROM INVENTTABLE B
WHERE B.DATAAREAID=INVENTSUM.DATAAREAID
AND B.ITEMID=INVENTSUM.ITEMID AND B.ITEMTYPE=2)
```

The statement should always be written as follows:

```
DELETE FROM INVENTSUM
WHERE DATAAREAID=N'dmo' AND
EXISTS (SELECT 'x' FROM INVENTTABLE B
WHERE B.DATAAREAID=N'dmo'
AND B.ITEMID=INVENTSUM.ITEMID AND B.ITEMTYPE=2)
```

In the event that the InventTable is shared among several companies in the ‘dmo’ company, then the statement should be as follows, where the virtual company is assumed to be named ‘vir’:

```
DELETE FROM INVENTSUM
WHERE DATAAREAID=N'dmo' AND
EXISTS (SELECT 'x' FROM INVENTTABLE B
WHERE B.DATAAREAID=N'vir'
AND B.ITEMID=INVENTSUM.ITEMID AND B.ITEMTYPE=2)
```

To get the correct DataAreaId, declare a table buffer of the specific table type and use the value of the DataAreaId field in the table buffer.

To get the correct formatting with the ‘-s and the preceding N, parse the DataAreaId to the SqlSystem.sqlLiterals method and use the return value.

The following example assumes that DataAreaId is left justified, which is a valid assumption as it is a system field where the justification cannot be changed by the customers or partners. The example is only used for demonstrating the use of DataAreaId. The table names and fields should be retrieved from the dict classes and the statement should be built using name(DbBackend::Sql).

The following shows the use of DataAreaId and sqlLiteral:

```
static void UseDataAreaId(Args _args)
{
    InventSum   inventSum;
    InventTable inventTable;
    str         sqlStr;
    SqlSystem   sqlSystem = new SqlSystem();
    ;
    sqlStr = strfmt(@"DELETE FROM INVENTSUM
                WHERE DATAAREAID=%1 AND
                EXISTS (SELECT 'x' FROM INVENTTABLE B
                WHERE B.DATAAREAID=%2
                AND B.ITEMID=INVENTSUM.ITEMID AND B.ITEMTYPE=2)",
               sqlSystem.sqlLiteral(inventSum.dataAreaId),
               sqlSystem.sqlLiteral(inventTable.dataAreaId));
}
```

Determining Whether a Table or Field Exists in the Database

You can test whether a table exists in the database by using the isTmp() method on the table buffer as shown in the following example:

```
static void TestTable(Args _args)
{
    SalesTable salesTable;
    ;
    if (!salesTable.isTmp()) // remember the NOT operator
    {
        // table exists in the database.
        //  isTmp will return true if the table is
        // specifically marked as temporary or if it is
        // disabled by the configuration key.
    }
}
```

You can test whether a field exists in the database by testing its configuration key as follows:

```
static void TestField(Args _args)
{
    DictField dictField;
    ;
    dictField = new DictField(tableNum(SalesTable),
                              fieldNum(SalesTable, PriceGroupId));
    if (isConfigurationKeyEnabled(dictField.configurationKeyId()))
    {
        // Field exists in the database
    }
}
```

There is no need to test every field. If you know the field is always in the database because the table is in the database, then there is no need to test each field individually. You only need to test fields that have a different configuration key to the table.

Defining String Lengths

When writing Direct SQL or stored procedures, it cannot be assumed that a given string field has the currently defined length as it may have been changed by the user before the execution of the upgrade job.

A variable designed to hold an ItemId cannot be defined as NVARCHAR(20) even though the current maximum length for the ItemId data type is 20. It may have been changed to a higher value, and, consequently, the variable cannot hold the entire value for all items. The length of the variable should therefore be defined taking the length of the type at execution time into consideration.

The current maximum length of a field should be retrieved through the ReleaseUpdateDB::fieldStringSize method.

Applying LTrim for String Comparisons in the WHERE Clause

In X++, left and right justification is managed by the kernel using string comparisons in the WHERE clause. Microsoft Dynamics AX 4.0 is left justified when installed, so there is no need to handle compares within Microsoft shipped upgrade scripts. If customers use mixed-mode, then, in Direct SQL, the code needs to check the justification of the two sides of the comparison and apply LTRIM on the right justified side if the two sides have different justification properties.

The new static method fields2WhereClause() is created in ReleaseUpdateDB class. It returns a string to be used in a Direct SQL WHERE clause.

Implementing Complex Inserts and Updates in Direct SQL

Complex updates cannot be implemented directly in X++. When these conditions are encountered, the update operations must be rewritten in Direct SQL.

If the method being examined involves one or a small number of update operations, the SQL can be constructed as a string and executed as described in Executing Direct SQL from X++ in this document.

For more complex methods that operate on multiple tables, it is advisable that the method be rewritten as a stored procedure. The stored procedure can be executed via X++ as described in Stored Procedure and Function Guidelines in this document.

Creating Stored Procedures and Functions

If stored procedures are needed in order to implement direct Transact-SQL logic, it may be created during execution time, executed, and then dropped after the upgrade script has run.

The AOS account has the privilege to create a stored procedure but it does not have execute permission on all stored procedures or functions. In order for your upgrade script to have the permission to execute the stored procedure or function you created, you need to prefix the object with the schema that the AOS account owns, and always use the two part name:

    [schema name].[object name]

in the create, execute, and drop statements.

To get the correct schema name, use the utility function:

    ReleaseUpdateDB::getSchemaName().

Example:

```
void createDimHistory_PurchInvoice_DSQL()
{
   InventReportDimHistory  dimHistory;
   VendInvoiceTrans vendInvoiceTrans;
   InventTrans           inventTrans;
   SqlSystem  sqlSystem                  = new SqlSystem();
   SqlStatementExecutePermission sqlStatementExecutePermission;
   str str_ExecSproc;
   str str_SQLEXEC = 'EXEC [%1].%2 %3';

   void runOraCode()
   {
      while select vendInvoiceTrans
      exists join inventTrans
            where inventTrans.InventTransId == vendInvoiceTrans.InventTransId
            &&    inventTrans.InvoiceId     == vendInvoiceTrans.InvoiceId
      notexists join dimHistory
            where dimHistory.InventTransId      == vendInvoiceTrans.InventTransId
            &&    dimHistory.TransRefId         == vendInvoiceTrans.InvoiceId
            &&    dimHistory.TransactionLogType == InventReportDimHistoryLogType::PurchInvoice
      {
         InventReportDimHistory::addFromVendInvoiceTrans(vendInvoiceTrans);
      }
   }
   ;
```

```
    if (dimHistory.isTmp() || inventTrans.isTmp() || vendInvoiceTrans.isTmp())
        return;

    select firstonly RecId from vendInvoiceTrans;

    if (!vendInvoiceTrans.RecId)
        return;

    switch(SqlSystem::databaseBackendId())
    {
        case DatabaseId::Oracle:
            runOraCode();
            break;
        case DatabaseId::MS_Sql_Server:
            str_ExecSproc = strfmt(str_SQLEXEC,ReleaseUpdateDB::getSchemaName()
                                        ,#CREATEDIMHISTORY_PURCHINVOICE
                                        ,sqlSystem.sqlLiteral(vendInvoiceTrans.DataAreaId));
            sqlStatementExecutePermission = new                     SqlStatementExecutePermission(str_ExecSproc);
            sqlStatementExecutePermission.assert();
            ReleaseUpdateDB::statementExeUpdate(str_ExecSproc);
            CodeAccessPermission::revertAssert();
}
```

When writing stored procedures that replace X++ methods or functions in the upgrade class, use the following guidelines:

- The stored procedure name should be the same as the method or function that it is replacing.
- The stored procedure should include the original X++ statements as comments to provide context during testing and troubleshooting.
- Transactional control statements (BEGIN TRANSACTION, COMMIT) should not be coded in the stored procedure. Transaction management is implemented in X++.
- The stored procedure must accept a required parameter of DATAAREAID as data type NVARCHAR(3).
- If the stored procedure will be populating a table with a formatted business sequence column (described in Assigning Business Sequences on Insert section of this document), the procedure must accept the following parameters:
    - @NUMBERSEQUENCE NVARCHAR(20). This will be used as a key to the NUMBERSEQUENCE table to retrieve the next key value and format requirements.
    - @RJUSTIFY CHAR(1). If "Y", this indicates the column is to be right justified.

Implementing Set-Based Updates with Joins

Update operations that involve true joins (in contrast to exists joins) cannot be directly implemented in X++ and represent one case where a Transact-SQL rewrite is needed. The following code is an example of an update that derives data from another table:

```
while select forupdate salesLine
    where salesLine.ShippingDateRequested == dateNull()
    join firstonly  maxof(DateExpected) from inventTrans
    group by InventTransId
    where inventTrans.InventTransId == salesLine.InventTransId &&
          inventTrans.DateExpected  != dateNull()
     {
     salesLine2 =
       SalesLine::findInventTransId(inventTrans.InventTransId,true);
       salesLine2.ShippingDateRequested = inventTrans.DateExpected;
      if (salesLine2)
         salesLine2.doUpdate();
      }
```

The corresponding Transact-SQL update is written as follows:

```
UPDATE SALESLINE
SET SHIPPINGDATEREQUESTED =
(          SELECT   MAX(B1.DATEEXPECTED) FROM INVENTTRANS B1
           WHERE   A.DATAAREAID = B1.DATAAREAID
           AND                  A.DATAAREAID = @dataareaid
           AND                  A.INVENTTRANSID = B1.INVENTTRANSID
           AND                  B1.DATEEXPECTED  <> '1900-01-01'
           AND                  A.SHIPPINGDATEREQUESTED = '1900-01-01')
FROM SALESLINE A, INVENTTRANS B0
WHERE A.SHIPPINGDATEREQUESTED = '1900-01-01'
AND A.DATAAREAID = @dataareaid
AND A.INVENTTRANSID = B0.INVENTTRANSID
AND B0.DATEEXPECTED <> '1900-01-01'
```

Using Direct SQL for Set-Based Updates

The following code is an example of performing a set-based update using the updateSalesAndTransLineDlvAddress:

```
   while select salesTable
   {
      update_recordset salesLine
         setting deliveryAddress         = salesTable.DeliveryAddress,
                 deliveryName           = salesTable.DeliveryName,
                 deliveryStreet         = salesTable.DeliveryStreet,
                 deliveryZipCode         = salesTable.DeliveryZipCode,
                 deliveryCity           = salesTable.DeliveryCity,
                 deliveryCounty          = salesTable.DeliveryCounty,
                 deliveryState           = salesTable.DeliveryState,
                 deliveryCountryRegionId = salesTable.DeliveryCountryRegionId
         where salesLine.SalesId         == salesTable.SalesId
            && salesLine.DeliveryAddress == '';

       //The journal lines must be updated for intrastat to function
       update_recordset custInvoiceTrans
         setting DlvCountryRegionId  = salesTable.DeliveryCountryRegionId,
                 DlvCounty           = salesTable.DeliveryCounty,
                 DlvState            = salesTable.DeliveryState
            where custInvoiceTrans.SalesId         == salesTable.SalesId
            && custInvoiceTrans.DlvCountryRegionId == '';

      update_recordset custPackingSlipTrans
```

```
        setting DlvCountryRegionId  = salesTable.DeliveryCountryRegionId,
                DlvCounty           = salesTable.DeliveryCounty,
                DlvState            = salesTable.DeliveryState
           where custPackingSlipTrans.SalesId          == salesTable.SalesId
           && custPackingSlipTrans.DlvCountryRegionId   == '';
    }
```

In this example, the code loops through every SalesTable Entry and:

- Updates SalesLine with the relevant address information for the salesid.
- Updates CustInvoicetrans with the address information for salesid.
- Updates custPackingSlipTrans with the address information for salesid.

Direct SQL needs to be rewritten in this case because of the need to:

- Perform one mass update where possible.
- Reduce looping on a large transactional table such as salesline.

The following is the Transact-SQL code that you should generate from X++:

```
UPDATE      SALESLINE
SET   DELIVERYADDRESS         = T.DELIVERYADDRESS,
      DELIVERYNAME            = T.DELIVERYNAME,
      DELIVERYSTREET          = T.DELIVERYSTREET,
      DELIVERYZIPCODE         = T.DELIVERYZIPCODE,
      DELIVERYCITY            = T.DELIVERYCITY,
      DELIVERYCOUNTY          = T.DELIVERYCOUNTY,
      DELIVERYSTATE           = T.DELIVERYSTATE,
      DELIVERYCOUNTRYREGIONID = T.DELIVERYCOUNTRYREGIONID
FROM  SALESLINE L,
      SALESTABLE T
WHERE       T.DATAAREAID      = @DATAAREAID
AND   L.DATAAREAID      = T.DATAAREAID
AND   L.SALESID         = T.SALESID
AND   L.DELIVERYADDRESS = ''

UPDATE      CUSTINVOICETRANS
SET   DLVCOUNTRYREGIONID      = T.DELIVERYCOUNTRYREGIONID,
      DLVCOUNTY               = T.DELIVERYCOUNTY,
      DLVSTATE          = T.DELIVERYSTATE
FROM  CUSTINVOICETRANS C,
      SALESTABLE T
WHERE       T.DATAAREAID      = @DATAAREAID
AND   C.DATAAREAID      = T.DATAAREAID
AND   C.SALESID         = T.SALESID
AND   C.DLVCOUNTRYREGIONID = ''


UPDATE      CUSTPACKINGSLIPTRANS
SET   DLVCOUNTRYREGIONID  = T.DELIVERYCOUNTRYREGIONID,
      DLVCOUNTY           = T.DELIVERYCOUNTY,
      DLVSTATE           = T.DELIVERYSTATE
FROM  CUSTPACKINGSLIPTRANS C,
      SALESTABLE T
```

```
WHERE           T.DATAAREAID     = @DATAAREAID
AND    C.DATAAREAID     = T.DATAAREAID
AND    C.SALESID           = T.SALESID
AND    C.DLVCOUNTRYREGIONID   = ''
```

The performance improvement achieved in this example is significant. On a database, Baseline ran for 24 minutes. With SET BASED CHANGE, it ran in 16 seconds.

This type of update, which does not require sequencing conditional to each record, can be written in X++ as a sequence of Direct SQL statements.

Using a Set-Based Insert Operation

There are a number of cases in the upgrade process where tables that are new in Microsoft Dynamics AX 4.0 must be populated from one or more tables. If the volume of data to be processed in these tables is large, and if INSERT_RECOREDSET does not achieve the desired performance, then using a set-based insert operation is required.

Example Transact-SQL set-based inserts are written as:

```
INSERT INTO SOME_NEW_TABLE (column-list)
        SELECT column-list FROM SOME_OLD_TABLE WHERE criteria
```

System Sequence Considerations

A complicating factor when we use a Direct SQL set-based insert into a table in the Microsoft Dynamics AX database is that tables have one or more sequentially assigned numbers which are derived from the SYSTEMSEQUENCESand NUMBERSEQUENCETABLE tables.

A two-step process of initially populating a temporary table that uses a DBMS-specific sequence mechanism (IDENTITY for Transact-SQL, ROW NUMBER for Oracle) and then copying the temporary table’s rows to the final permanent table is required.

The two sections that follow provide Transact-SQL examples of populating both a system sequence (RECID) and business sequence.

RECID in Microsoft Dynamics AX 2012

The RECID allocation algorithm has undergone significant changes in Microsoft Dynamics AX 2012. A RECID can be allocated in two different ways:

- Kernel automatically allocates the RECID during insert and INSERT_RECORDSET
- User manually chooses to allocate the RECID

In the case of upgrade, we are concerned about #2. This section will document the allocation APIs, the usage and some patterns. The document does not dwell in the allocation algorithm itself.

Manually allocating RECID

There are cases where you want to allocate the RECID manually in your script. The following are some of the scenarios:

- You are trying to do a bulk insert manually. There are cases where row by row insert is not sufficient and you want to do a bulk insert. Import/Export code is an example of this usage pattern. In such a case, you need to allocate the RECID manually.

- An upgrade script uses direct SQL to insert data. In this usage pattern, you need to allocate RECID manually.
- Upgrade script was optimized to use RecordInsertList instead of row by row insert. But, cross references need to be set up on another table (for example REFRECID). In such a case, allocate the RECID upfront for the record so that cross references can be patched up.

In all the above scenarios, the allocation is done the same way, using the RECID allocation APIs. There are three APIs that you need to know about:

RECID suspension - suspendRecids

RECID reservation - reserveValues

RECID releasing suspension - removeRecidSuspension

The APIs are members of the SystemSequence class.

The following is a code snippet of how to use the allocation APIs.

```
static void Job2(Args _args)
{
    SystemSequence s;
    AAMyTable t;
    int64 startValue;
    int i;
    ;
    s = new SystemSequence();                              Create a new instance of the
    s.suspendRecIds( tablenum (AAMyTable) );              Suspend the RECID allocation by the
    startValue = s.reserveValues( 10, tablenum( AAMyTable ) );   Reserve the RECID by passing in the
                                                                 number of id's to reserve. The return
                                                                 value is the starting value of the
    for ( i = 0; i <10; i++ )                                    range you reserved. The API
    {                                                            gaurantees that the allocated id's
                                                                 are contiguous.
        t.IntFld = i;
        t.RecId = startValue + i;                          Assign the RECID to the RECID column
        t.insert();
    }

    s.removeRecIdSuspension( tablenum ( AAMyTable ) );    Remove the suspension
}
```

Tips on using the RECID allocation API:

- Once you suspend the RECID allocation for that table, the kernel will not dispense any more RECIDs for that table on that session.
- The ReserveValues API will guarantee contiguity of the RECID range that is being reserved.
- If you try to insert an id that has not been reserved, then kernel will raise an exception.
- If you are trying to assign a RECID without suspending, kernel will raise an exception.
- If you do not remove the suspension after using the reservation API's, the suspension remains until the end of your session.

Assigning RECID on INSERT

RECID is a continuously ascending key value for each table in the Microsoft Dynamics AX schema. It is derived from table SYSTEMSEQUENCES which keeps the next available key value (NEXTVAL) for each table by that table's Table ID.

The SystemSequences table may be empty if the table is new and no records have been inserted. Please refer to the ReleaseUpdateDB39_Cust.createDimHistorySprocs(), which provides an example of the solution for that problem: it checks if a RECID existed and if not, inserting and deleting a record to get the RECID's started.

In Microsoft Dynamics AX 4.0, RECID is a 64-bit integer column; this data type is implemented in SQL Server as BIGINT.

The abbreviated example below illustrates using SYSTEMSEQUENCES and a temporary table using IDENTITY for sequential numbers:

```
CREATE      PROCEDURE initFromSMMQuotationTable
            @DATAAREAID NVARCHAR(3
AS
DECLARE     @NEXTVAL     BIGINT,
            @ROWCOUNT    BIGINT


SELECT ......,
      RECID          = IDENTITY(BIGINT,1,1) AS QUOTATIONID
INTO   #TEMP
FROM   DEL_SMMQUOTATIONTABLE
WHERE        QUOTATIONSTATUS = 0 -- SMMQUOTATIONSTATUS::INPROCESS

SELECT @NEXTVAL=NEXTVAL
FROM SYSTEMSEQUENCES (UPDLOCK, HOLDLOCK)
WHERE       ID = -1
AND   TABID = 1967

INSERT INTO SALESQUOTATIONTABLE
(column-list)
SELECT          ......,
     RECID  = QUOTATIONID+@NEXTVAL
FROM #TEMP

SELECT @ROWCOUNT = COUNT(*) FROM #TEMP


UPDATE SYSTEMSEQUENCES
SET    NEXTVAL=NEXTVAL + @ROWCOUNT
WHERE       ID = -1
AND   TABID = 1967
GO
```

```
Assign an IDENTITY column
with a starting value of 0
```

```
Retrieve the next value
for RECID for this table
```

```
When we insert into the
permanent table, we add the
temporary table's IDENTITY
column to the next value
```

```
We update SYSTEMSEQUENCES to
reflect the number of rows
that we have added to this
```

Looking Up Table ID and Field IDs

If you are gettingTABID in the stored procedure, you should perform the fetch from the SQL Dictionary.

Assigning Business Sequences on Insert

Business sequences are a more complex problem to solve with Direct SQL; not only is the number sequentially assigned from a table (NUMBERSEQUENCETABLE), but you also have to consider the following factors:

- The specific number sequence to be used for a specific column.
- Whether the column is to be left or right justified.
- The customer's specific formatting requirements (FORMAT) for the column.

The first two factors are accessible in X++ and, as described in the stored procedure guidelines above, must be passed as parameters to any stored procedure which must populate a formatted business sequence number.

Once the specific numbersequence to be used is known, the formatting requirement must be retrieved from the FORMAT column of the NUMBERSEQUENCETABLE table. The following lists additional details for this scenario.

- The stored procedure is passed an indicator that specifies if right justification is to take place. A value of "Y" means right-justify the column. The default is to left-justify the column.
- Because formatted sequence columns are of different maximum lengths, you must look up the length of the column that is to be formatted and record the length in your procedure. The instructions that follow will describe how you pass the column's length, along with the formatting requirements, to a user-defined SQL function that will format the column correctly.

The example below illustrates the use of a user-defined function FN_FMT_NUMBERSEQUENCE which accomplishes the formatting and justification requirements of a business sequence column:

```
CREATE          PROCEDURE initFromSMMQuotationTable
                @DATAAREAID NVARCHAR(3),
                @NUMBERSEQUENCE NVARCHAR(20),
                @RJUSTIFY CHAR(1)

AS
DECLARE         @NEXTREC        BIGINT,
                @FORMAT         NVARCHAR(40),
                @ROWCOUNT       BIGINT
                @RJUSTIFY_LENGTH INT
```

You must determine the column's length if it is to be right justified and set a variable so we can pass that to the

```
IF RJUSTIFY = 'Y'
        SET @RJUSTIFY_LENGTH = 40
ELSE
        SET @RJUSTIFY_LENGTH = 0
```

As in the previous example, we create an IDENTITY column in the temporary table with

```
SELECT QUOTATIONID = IDENTITY(BIGINT,1,1),
        ......
INTO   #TEMP
FROM  DEL_SMMQUOTATIONTABLE
WHERE           QUOTATIONSTATUS = 0 -- SMMQUOTATIONSTATUS::INPROCESS
```

Retrieve the next value from NUMBERSEQUENCETABLE using the NUMBERSEQUENCE key supplied

```
SELECT @NEXTREC = NEXTREC, @FORMAT=FORMAT
FROM  NUMBERSEQUENCETABLE (UPDLOCK, HOLDLOCK)
WHERE           DATAAREAID = @DATAAREAID
AND    NUMBERSEQUENCE = @NUMBERSEQUENCE


INSERT INTO SALESQUOTATIONTABLE
(column-list )
SELECT
   DBO.FN_FMT_NUMBERSEQUENCE(@FORMAT,QUOTATIONID,@NEXTREC, @RJUSTIFY_LENGTH) ,
   ......

FROM  #TEMP


SELECT @ROWCOUNT = COUNT(*) FROM #TEMP
```

Details on calling this function follow

```
UPDATE NUMBERSEQUENCETABLE
SET     NEXTREC = NEXTREC+@ROWCOUNT
WHERE           DATAAREAID = @DATAAREAID@NUMBERSEQUENCE
AND     NUMBERSEQUENCE = @NUMBERSEQUENCE
```

```
Update NUMBERSEQUENCE to
reflect the number of
rows added to the table
```

In many cases it will be necessary to assign a sequential number both for RECID and a business sequence column. However, SQL Server only permits one IDENTITY column per table.

The following example demonstrates how to use the single IDENTITY column for both purposes. This example is also useful as a template for creating new procedures to upgrade data into new tables in the Microsoft Dynamics AX 4.0 schema:

```
CREATE          PROCEDURE initFromSMMQuotationTable
                @DATAAREAID NVARCHAR(3),
                @NUMBERSEQUENCE NVARCHAR(20),
                @RJUSTIFY CHAR(1) ='N'
AS
DECLARE         @NEXTREC                BIGINT,
                @NEXTVAL                BIGINT,
                @FORMAT                 NVARCHAR(40),
                @ROWCOUNT               BIGINT
                @RJUSTIFY_LENGTH        INT
-- Set the length of the column that is to be right-justified
-- Confirm length in table definition
IF RJUSTIFY = 'Y'
        SET @RJUSTIFY_LENGTH = 40
ELSE
        SET @RJUSTIFY_LENGTH = 0


-- The SELECT INTO creates a temp table
-- RECID is assigned during the insert and given
-- a sequentially ascending number starting with 0
SELECT QUOTATIONID = ''
        ......
        RECID = IDENTITY(BIGINT,1,1),
INTO    #TEMP
FROM    DEL_SMMQUOTATIONTABLE
WHERE           QUOTATIONSTATUS = 0 -- SMMQUOTATIONSTATUS::INPROCESS


-- Retrieve next key value for RECID
-- Note TABID; you need to determine the
-- value from the SQLDICTIONARY table.
SELECT @NEXTVAL=NEXTVAL
FROM    SYSTEMSEQUENCES (UPDLOCK, HOLDLOCK)
WHERE           ID = -1 AND TABID = 1967

-- Retrieve next key value for business sequence (QUOTATIONID)
-- NUMBERSEQUENCE is supplied in X++ and passed in @NUMBERSEQUENCE
SELECT @NEXTREC = NEXTREC, @FORMAT=FORMAT
FROM    NUMBERSEQUENCETABLE (UPDLOCK, HOLDLOCK)
WHERE           DATAAREAID = @DATAAREAID AND NUMBERSEQUENCE = @NUMBERSEQUENCE
```

```
-- Insert from the temp table to the final table. The temp table RECID
--is used to supply values to both QUOTATIONID and RECID in the final table
INSERT INTO SALESQUOTATIONTABLE
(column-list )
SELECT
   DBO.FN_FMT_NUMBERSEQUENCE(@FORMAT, RECID,@NEXTREC, @RJUSTIFY_LENGTH) ,
   ......,
   RECID+@NEXTVAL
FROM  #TEMP


-- Row count of temp table then used to update both NUMBERSEQUENCETABLE
-- and SYSTEMSEQUENCES tables
SELECT @ROWCOUNT = COUNT(*) FROM #TEMP

UPDATE NUMBERSEQUENCETABLE          SET NEXTREC = NEXTREC+@ROWCOUNT
WHERE           DATAAREAID = @DATAAREAIDAND AND NUMBERSEQUENCE = @NUMBERSEQUENCE

UPDATE SYSTEMSEQUENCES       SET NEXTVAL=NEXTVAL + @ROWCOUNT
WHERE           ID = -1 AND TABID = 1967
```

Calling FN_FMT_NUMBERSEQUENCE

A user defined function FN_FMT_NUMBERSEQUENCE is provided to assist with the formatting requirements of a business sequence column. This function enables the following operations to be performed:

- Adds the value of the IDENTITY column to the NEXTREC value retrieved from NUMBERSEQUENCETABLE.
- Formats the result according to the FORMAT column retrieved from NUMBERSEQUENCETABLE.
- Right justifies the formatted column to the length specified. If the function encounters a value of 0, no justification occurs and the formatted value remains left justified by default.

The parameters that are supplied to FN_FMT_NUMBERSEQUENCE are:

- The FORMAT column value from NUMBERSEQUENCETABLE.
- The integer value to be formatted.
- The value from NEXTREC in NUMBERSEQUENCETABLE. If this is not supplied, it is set to 0 by default.
    - The length of the column to be right justified. If this is not supplied it is set to 0 by default. If 0 is specified or becomes the default, then no justification occurs.

The ReleaseUpdateDB38_Basic::createFnFmtNumberSequence method creates the FN_FMT_NUBMERSEQUENCE function. If your script needs to call the function, you should make the script depend on the ReleaseUpdateDB38_Basic::createFnFmtNumberSequence script and then you can reference the function in your Direct SQL code.

Appendix 2: Debugging batch jobs.

Debugging upgrade batch jobs in Microsoft Dynamics AX 2012:

By default, batch jobs in Microsoft Dynamics AX 2012 run as Intermediate Language(IL) code. For more information, see How to: Debug IL Code in Microsoft Dynamics AX 2012.

When you add or change an upgrade script in the batch, you must do an X++ IL incremental build. Right-click on the root node of the AOT, point to Add-Ins, and then click on X++ IL incremental build.

Debugging upgrade batch jobs in Microsoft Dynamics AX 4.0 0r Microsoft Dynamics AX 2009:

- Edit the method runsImpersonated of your batch job class to always return True. For upgrade scripts, the class is ReleaseUpdateExecute.
- In Microsoft Dynamics AX 2009 this fix may be needed, for more information see Debugging non-interactive X++ code in Dynamics AX 2009 when running on Windows Server 2008.

Appendix 3: Changes in writing data upgrade scripts for Microsoft Dynamics AX 2013 R3

There are two major-version upgrade paths and two minor-version upgrade paths.
Use the major-version upgrade paths when upgrading AX 4.0 or AX 5.0 to AX 2013 R3.
Use the minor-version upgrade paths when upgrading from AX 6.0/6.1 to AX 2013 R3.
Minor-version upgrades run only the target upgrade scripts in place.

When doing major-version upgrades, observe the following guidelines:

- There is no change in the way the scripts are written in the major-version upgrade; the scripts should be written as they would be for an upgrade to AX 6.2.
- All major-version scripts should stay in either ReleaseUpdateDB41_<Module> or ReleaseUpdateDB60_<Module> classes.
- You can update the existing methods or add new methods to the existing/new classes.

When doing minor-version upgrades, observe the following guildelines:

- Class naming convention: ReleaseUpdateDB<Version>_<Module>, where the version is 6.0 – the version you upgrade from. (Note that versions 6.0 and 6.1 are both treated as 6.0). Sample script: ReleaseUpdateDB60_Administrator. updateSysEDTMigration() is a minor-version upgrade script from AX 6.0/6.1 to AX 6.3.
- Script version: The script version is defined by the #version macro in the class declaration. In a minor-version upgrade, it is the version that you upgrade from. For example, #define.version(sysReleasedVersion::v60) means an upgrade from AX 6.0/6.1 when it is applied on a minor-version script.
- Script stage: As for the major-version scripts, there is a set of script stages that should be used when setting the attributes for a minor-version script:
    - ReleaseUpdateScriptStage::PreSyncUpdate: Pre-synchronization minor-upgrade script.
    - ReleaseUpdateScriptStage::PostSyncUpdate: Post-synchronization minor-upgrade script.
    - ReleaseUpdateScriptStage::AdditionalUpdate: Additional minor-upgrade script.
- Other upgrade script attributes are used just as they are in the major-version upgrade scripts.
- Required init* jobs methods: As for a major-version script, there is a new set of required methods that must be added to the script class: initPreSyncUpdateJobs, initPostSyncUpdateJobs, and initAdditionalcUpdateJobs. These methods contain a single line of code that is used by the Upgrade framework: #initSyncJobsPrefix. To save time, you can just copy these methods from any existing class.

When writing minor-version upgrade scripts to upgrade from AX 6.2 to AX 2013 R3, observe the following guidelines:

- Class-naming convention: ReleaseUpdateDB<Version>_<Module>, where the version you are upgrading from is 6.2. Sample script:

  ReleaseUpdateDB62_Administrator. postSyncScript() is a minor-version upgrade script from AX 6.2 to AX 6.3.
  - Script version: The script version is defined by the #version macro in the class declaration. In a minor-version upgrade, it is the version that you upgrade from. For example, #define.version(sysReleasedVersion::v62) means an upgrade from AX 6.2 when it is applied on a minor-version script.
  - Script stage: As for the major-version scripts, there is a set of script stages that should be used when setting the attributes for a minor-version script:
    - ReleaseUpdateScriptStage::PreSyncUpdate: Pre-synchronization minor-upgrade script.
    - ReleaseUpdateScriptStage::PostSyncUpdate: Post-synchronization minor-upgrade script.
    - ReleaseUpdateScriptStage::AdditionalUpdate: Additional minor-upgrade script.
  - Other upgrade script attributes are used just as they are in the major-version upgrade scripts.
  - Required init*Jobs methods: As for a major-version script, there is a new set of required methods that must be added to the script class: initPreSyncUpdateJobs, initPostSyncUpdateJobs, and initAdditionalcUpdateJobs. These methods contain a single line of code that is used by the Upgrade framework: #initSyncJobsPrefix. To save time, you can just copy these methods from any existing class.

Note: When writing minor-version upgrade scripts, keep in mind that the scripts are for upgrades from the latest cumulative update (AX 6.0/6.1+CU5, AX 6.2+CU7).

Note: From the preceding naming convention, an in-place upgrade script from AX 6.0/6.1 to AX 6.3 and a major upgrade script from AX 4.0/5.0 to AX 6.3 can be in the same ReleaseUpdateDB60_* class, but they have different Stage attributes.

Understanding the #version macro in the classes:
- In a major-version upgrade, #version defines the version you upgrade *to*.
- In a minor-version upgrade, #version defines the version you upgrade *from*.

How to test load/run your minor-version scripts:
- During development, you might want to check whether your minor-version scripts load properly. Use the following steps:
  1. Generate incremental CIL if there is any change in the scripts.
  2. Run the following job to clean up any previously loaded scripts so you can reload them (you must update one line of code to reflect the version you upgrade from; see the TODO that follows):

```
static void CleanupScripts(Args _args)
{
    SysSetupLog          sysSetupLog;
    SysSetupCompanyLog   sysSetupCompanyLog;
    ReleaseUpdateScriptsHistory     releaseUpdateScriptsHistory;
    ReleaseUpdatePrioritizedJobs    releaseUpdatePrioritizedJobs;
    ReleaseUpdateMinorScripts       releaseUpdateMinorScripts;

    appl.globalCache().set(staticMethodStr(ReleaseUpdateDB, getFromVersionEx), 0,
sysReleasedVersion::v62);// TODO: v60 or v62 depending on which version you upgrade
from.
```

```
    ReleaseUpdateDB::cleanupJobs();
    ReleaseUpdateCockpit::cleanupBatch();
    delete_from releaseUpdateMinorScripts;
    delete_from sysSetupCompanyLog;
    delete_from releaseUpdateScriptsHistory;
    delete_from releaseUpdatePrioritizedJobs;
    delete_from sysSetupLog where sysSetupLog.Name ==
classStr(SysCheckListItem_LoadUpdateExScripts);
    ReleaseUpdateDB::setMinorVersionUpdateInProgress(false);
    SysChecklist_Update::resetCheckList();
}
```

3. Open the Upgrade In-Place checklist (SysCheckList_Update menu item), and click Presynchronize. Start the data upgrade or upgrade additional features, depending on what type of scripts you want to load (PreSync, PostSync, or Additional).
4. Verify that your scripts load correctly in the cockpit form.
5. You can also click the Run button to schedule and run the scripts; however, make sure the batch is configured properly in the SysServerConfig form, and that the batch server is assigned to the Data Upgrade group in the BatchGroup form. If you reconfigure a batch, you might need to restart the AOS server.

The following table shows how the upgrade framework picks the scripts based on the source version and script version.

Upgrade path	**Script picked by the framework** (based on its #version)
AX 4 to AX 6.*x*	sysReleasedVersion::v41 & sysReleasedVersion::60
AX 5 to AX 6.*x*	sysReleasedVersion::60
AX 6.0 to AX 6.3	sysReleasedVersion::60
AX 6.2 to AX 6.3	sysReleasedVersion::62

Microsoft Dynamics AX

Microsoft Dynamics is a line of integrated, adaptable business management solutions that enables you and your people to make business decisions with greater confidence. Microsoft Dynamics works like and with familiar Microsoft software, automating and streamlining financial, customer relationship and supply chain processes in a way that helps you drive business success.

U.S. and Canada Toll Free 1-888-477-7989

Worldwide +1-701-281-6500

www.microsoft.com/dynamics