UPGRADE

Microsoft Dynamics AX

How to Write Data Upgrade Scripts for Microsoft Dynamics AX 2009

White Paper

[This document describes how to use the Microsoft Dynamics™ AX Data Upgrade Framework and to write data upgrade scripts for customer data upgrade data models (Microsoft Dynamics AX tables).]

Date: May 14, 2008

http://www.microsoft.com/dynamics/ax

Microsoft Dynamics

Contents

Introduction

This document describes how to use the Microsoft Dynamics™ AX Data Upgrade Framework and 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 Microsoft Dynamics AX application developers.

This document is based on Leveraging the Microsoft Dynamics AX2009 Data Upgrade Framework, a Microsoft Dynamics AX2009 Technical Information document, and on the Microsoft Dynamics AX 2009 Data Upgrade Framework. It has been updated for the new data upgrade framework and Best Practices for performance.

Out of the box – Dynamics Ax 2009 supports upgrading data from Dynamics Axapta 3.0 and Dynamics Ax 4.0 to Dynamics Ax 2009.

When is a Data Upgrade Script Needed?

There are changes that can be made in the data model without the need for an upgrade script, and there are changes that need an upgrade script.

The following changes can be made without an upgrade script:

1. Change the name of a field
2. Change the name of a table
3. Add a field to a table with a default value for every field
4. Add/change relations
5. Add/change non-unique indexes
6. Add/change delete actions
7. Add/change/delete temporary table

The following changes require an upgrade script:

1. Delete a table and save data
2. Delete a field and save data
3. Add/change unique indexes
4. Change a non-unique index into a unique index
5. Restructure where data is stored. For example, moving data from one field to another
6. Correct old data inconsistencies
7. Populate new tables with existing data
8. Populate new fields with existing data or a default value that is different from the default value for the data type

The Upgrade Checklist

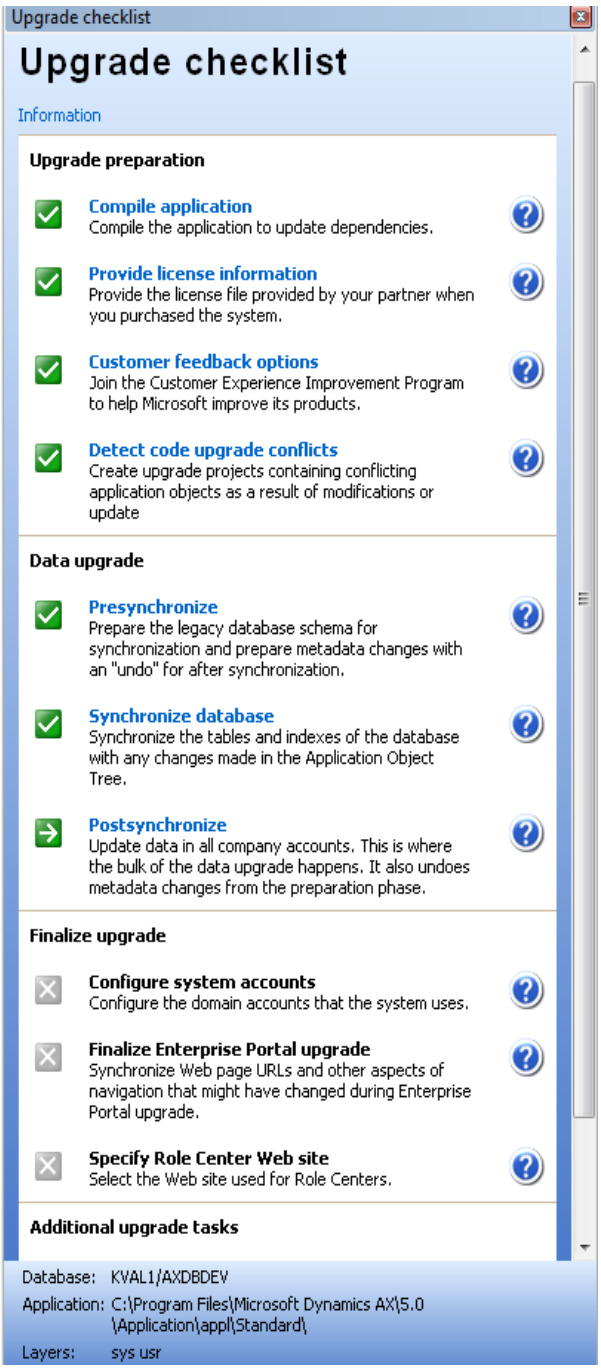


Figure 1. The Upgrade Checklist

The Upgrade Checklist is a navigation pane that guides you through the data upgrade steps. 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 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 Data Upgrade Framework

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. The class diagram of the upgrade script classes is shown in Figure 2.

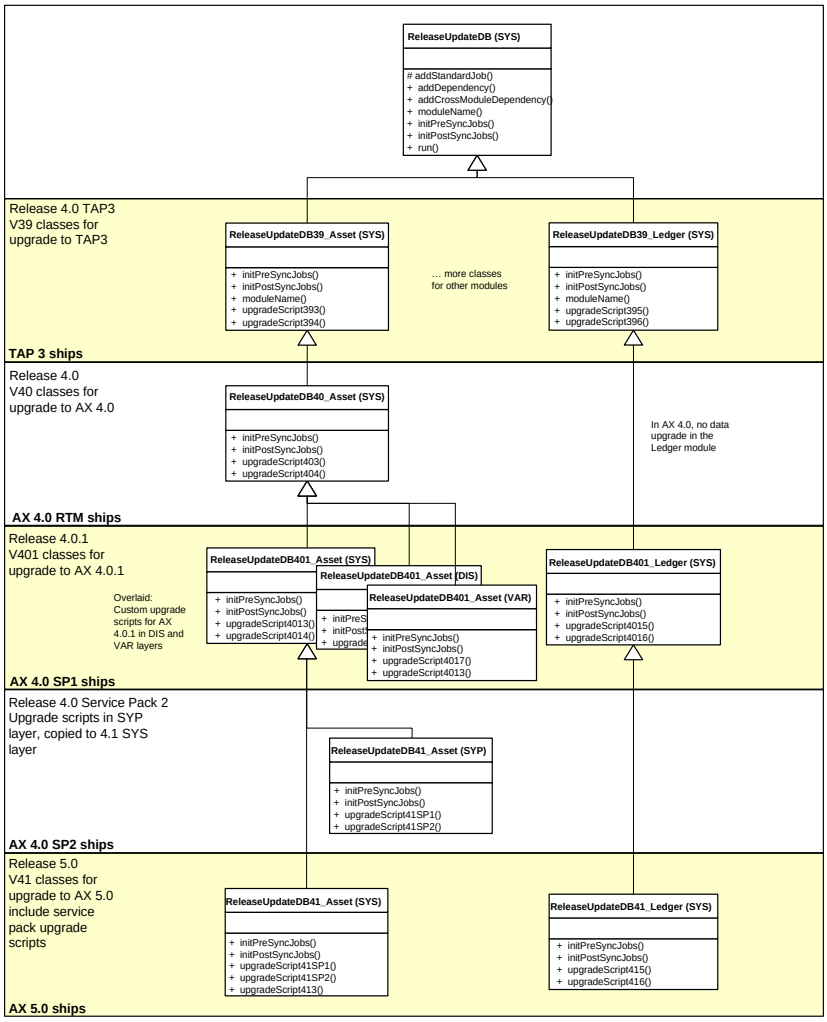


Figure 2. Data Upgrade Script Classes

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's application module and the version you are developing.

TAP3 (39)	**401**	**41**
ReleaseUpdateDB39_Administration	ReleaseUpdateDB401_Administration	ReleaseUpdateDB41_Administration
ReleaseUpdateDB39_Asset	ReleaseUpdateDB401_Bank	ReleaseUpdateDB41_Asset
ReleaseUpdateDB39_Bank	ReleaseUpdateDB401_COS	ReleaseUpdateDB41_Bank
ReleaseUpdateDB39_Basic	ReleaseUpdateDB401_Cust	ReleaseUpdateDB41_Basic
ReleaseUpdateDB39_COS	ReleaseUpdateDB401_Ledger	ReleaseUpdateDB41_COS
ReleaseUpdateDB39_Cust	ReleaseUpdateDB401_Proj	ReleaseUpdateDB41_Cust
ReleaseUpdateDB39_HRM	ReleaseUpdateDB401_Vend	ReleaseUpdateDB41_HRM
ReleaseUpdateDB39_Invent		ReleaseUpdateDB41_Invent
ReleaseUpdateDB39_Jmg		ReleaseUpdateDB41_Jmg
ReleaseUpdateDB39_KM		ReleaseUpdateDB41_KM
ReleaseUpdateDB39_Ledger		ReleaseUpdateDB41_Ledger
ReleaseUpdateDB39_PBA		ReleaseUpdateDB41_Prod
ReleaseUpdateDB39_Prod		ReleaseUpdateDB41_Proj
ReleaseUpdateDB39_Proj		ReleaseUpdateDB41_Req
ReleaseUpdateDB39_Req		ReleaseUpdateDB41_SMA
ReleaseUpdateDB39_SMA		ReleaseUpdateDB41_smm
ReleaseUpdateDB39_smm		ReleaseUpdateDB41_Trv
		ReleaseUpdateDB41_Vend

Name ReleaseUpdateDB39 means upgrade to Microsoft Dynamics AX TAP3. Pre-synchronization, Post-synchronization and Additional features upgrade methods coexist in these classes.

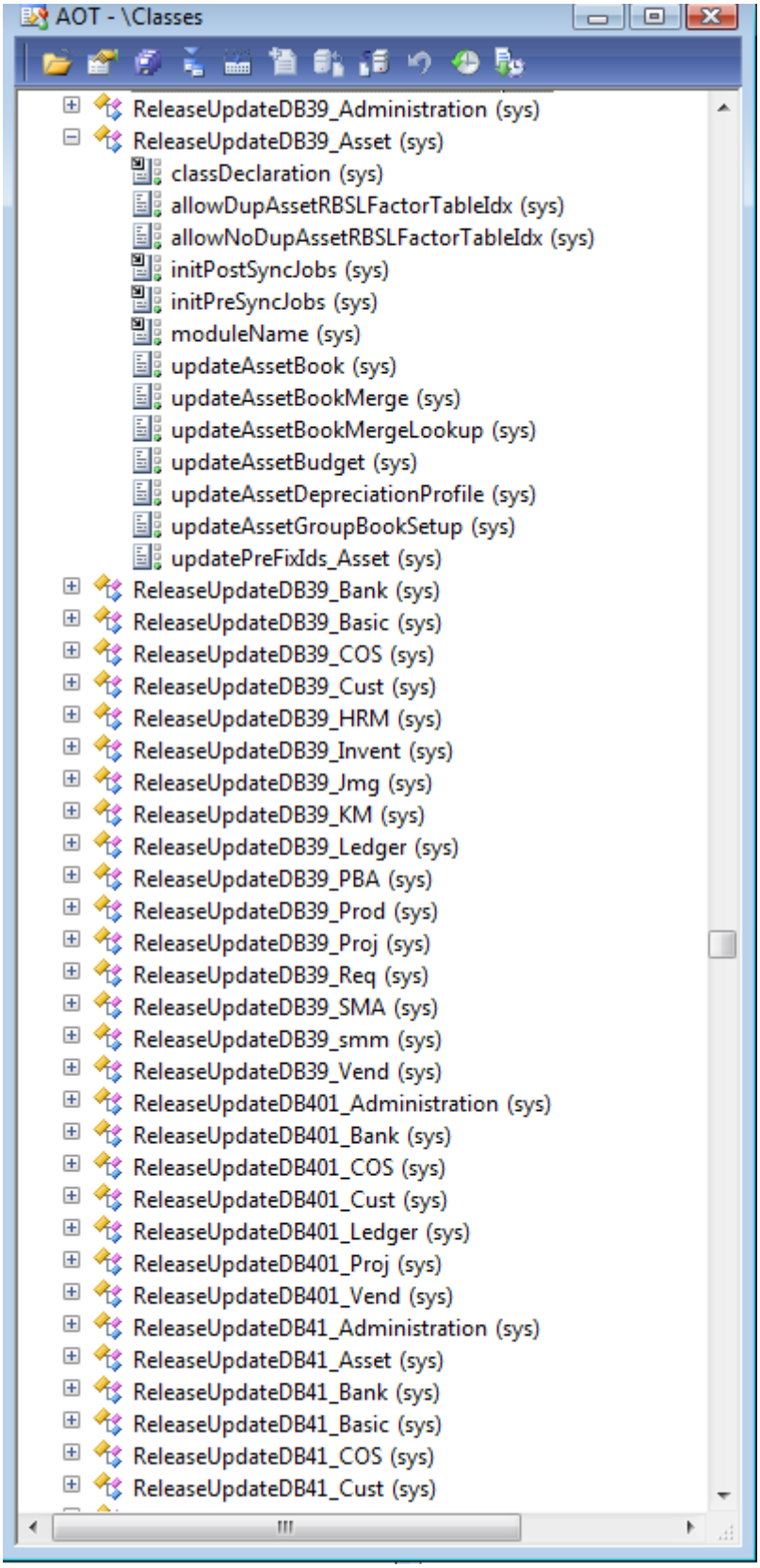
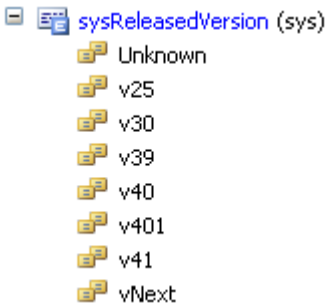


Figure 3. Upgrade Classes in the Applications Object Tree

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.



SYS versions are defined in the Base Enum SysReleaseVersion

Each ReleaseUpdateDB* class (except for the base ReleaseUpdateDB class) 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 TAP3 (39) and 4.0.1 (401), but does not have an upgrade script for release version 4.0 (40). Therefore, the class ReleaseUpdateDB401_Ledger inherits directly from ReleaseUpdateDB39_Ledger. While for the Asset module, there are upgrade scripts for versions 39, 40, 401, 41 (Microsoft AX 2009). Therefore the class ReleaseUpdateDB401_Asset must inherit from ReleaseUpdateDB40_Asset, which in turn inherits from ReleaseUpdateDB39_Asset.

```
Public class ReleaseUpdateDB401_Ledger extends ReleaseUpdateDB39_Ledger
{
}
```

```
Public class ReleaseUpdateDB40_Asset extends ReleaseUpdateDB39_Asset
{
}
```

```
Public class ReleaseUpdateDB401_Asset extends ReleaseUpdateDB40_Asset
{
}
```

```
Public class ReleaseUpdateDB41_Asset extends ReleaseUpdateDB401_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

    // Add upgrade scripts
 }
```

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

Data Upgrade for Service Packs

Service packs are shipped in the SYP layer of each major Microsoft Dynamics AX release. Service pack fixes are rolled forward into the next version of the SYS release. Therefore, they can be viewed as a pre-release of the next major release, and, to perform a data upgrade, the upgrade scripts are added to the upgrade script classes of the next major release in the SYP layer.

For example, in Figure 2, a data upgrade script in the Fixed Asset module for the Service Pack for Microsoft Dynamics AX 4.0.1 is implemented in the ReleaseUpdateDB41_Asset class in the SYP layer. This script will be merged with the data upgrade scripts for SYS release Microsoft Dynamics AX 4.1 into ReleaseUpdateDB41_Asset in the SYS layer. The data upgrade framework handles service pack releases by detecting at individual script level what has been run already in a service pack of the previous SYS release and skips the upgrade script.

```
void initPostSyncJobs()
{
    #initSyncJobsPrefix
    …
    // Add service pack upgrade scripts, for example, upgradeScript413
 }
```

Data Upgrade for Customization

Customizations are performed in layers higher than the SYS (and SYP) layers. If the customization requires a data upgrade, the same layer would be used to update the data upgrade scripts.

Customization of a data upgrade is performed by overlaying the SYS level data upgrade scripts classes in the same layer as the customization. This can be achieved by either overriding a SYS layer upgrade script or by adding a new upgrade script. This is illustrated in Figure 2. There are two overlaid ReleaseUpdateDB401_Asset classes in the DIS and VAR layer.

Note: Service pack releases and customizations (including local features, option pack providers, and partner customizations) have different purposes. Therefore, implementation of the data upgrade scripts for a Service Pack and for customization data upgrade will be different.

When customizing a data upgrade by overlaying data upgrade script classes, the `initPreSyncJobs`, `initPostSyncJobs` and `initAdditionalJobs` methods must be overlaid and the jobs from lower layers must be included in the current layer. For example, the ReleaseUpdateDB401_Asset::InitPreSyncJob in the VAR layer in Figure 2 should resemble the following sample:

```
void initPostSyncJobs()
{
    #initSyncJobsPrefix
```

```
        // Add SYS upgrade scripts, including overlaid upgrade scripts
        …
        // Add new DIS upgrade scripts, not DIS overlaid upgrade scripts
        …
        // Add new VAR upgrade scripts, not VAR overlaid upgrade scripts
    }
```

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:

1. Place the algorithm in the upgrade class for the source version, replacing the original algorithm. For example, if you are upgrading from version 3.0 to 4.0 SP1, put the combined algorithm in the ReleaseUpdateDB39 class.

2. 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 4.0 SP1 version to

```
3.      public void updateCustTrans()
4.      {
5.          if (ReleaseUpdateDB::getFromVersion() != sysReleasedVersion::v30)
6.          {
7.                Original script logic for upgrade from 4.0 to 4.0 SP1
8.          }
9.      }
```

Using Configuration Key to Remove Obsolete Objects after Upgrade

Note that after the upgrade is finished, you can disable the configuration keys “Keep update objects” (`SysDeletedObjects40 and SysDeletedObjects41` for Microsoft Dynamics AX 2009). 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

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 18 application modules for upgrade scripts. They are named `ReleaseUpdateDB<version>_<module>`, for example ReleaseUpdateDB39_Bank.

Each of these classes contains scripts for pre-synchronization, post-synchronization and additional upgrades. The scripts are scheduled by the `initPreSyncJobs`, `initPostSyncJobs` and `initAdditionalJobs` methods respectively.

Each class can handle your upgrade script in one of four different ways - Start, Shared, Normal (also called Standard), and Final. Note that it is important to 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)	

1. **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.

2. **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

3. **Presynchronize Normal 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.

4. **Presynchronize Final scripts**

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

5. **Postsynchronize Start scripts**

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

6. **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.

7. **Postsynchronize Normal 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)

8. **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.

1. **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

To create a script you need to create a method on the appropriate class. For example, for Microsoft Dynamics AX 4.0 TAP3 the class is ReleaseUpdateDB39_<module>. You must also inform the framework how to handle the script. This is done 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`. If you would like your job to run in 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.

The following are 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) ]);
```

Upgrade script configuration keys

Developers 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.

```
this.addFinalJob(methodstr(ReleaseUpdateDB39_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(Configur
ationKey1) ])
```

Note that if you are using setModuleConfigKey, it should be called from InitPreSyncJobs, initPostSyncJobs and InitAdditionalJobs method 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, include the following in the appropriate `InitXXXJobs` method:

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

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

1. If you have a dependency **between the scripts inside a module**, use the `addDependency` method.
2. If the script is **dependant on another module's script**, you can use the `addCrossModuleDependency` method to ensure a correct execution sequence between scripts placed in the different classes:

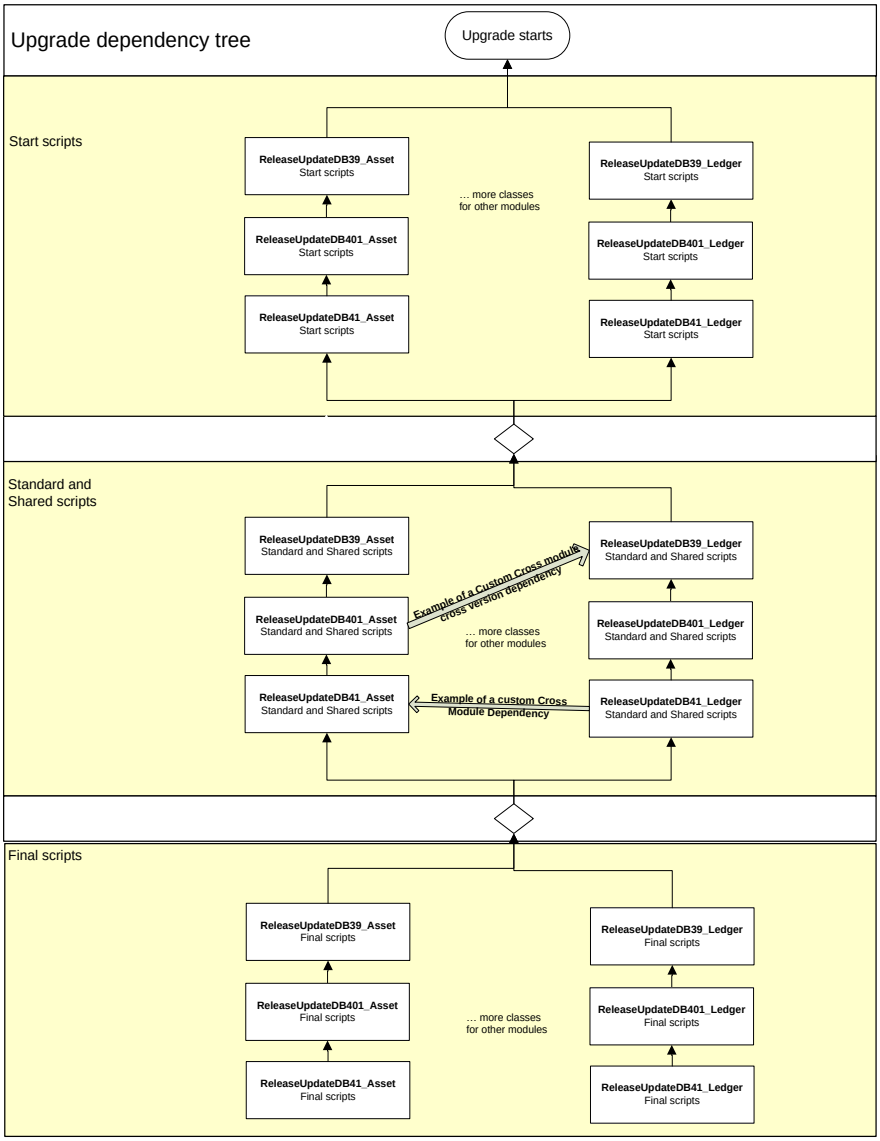
```
this.addCrossModuleDependency(classnum(<ClassName>), methodStr(<ClassName>, <MethodName>),
                        classnum <ClassName>, methodStr(<ClassName>, <MethodName>));
```

1. If the script is **dependant on another module's script from a previous version**, you can use the `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>);
```

2. If a script is **dependent on another script from a previous version but located in the same module**, then you don't need a dependency, as the upgrade framework automatically provides implicit dependency in that case.

Here is an example of the dependency tree:



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. This means that the executed code will use a new version of metadata, but the database will still be the old version.

Also, please note that several special tables are synchronized during AOS startup even before the upgrade checklist starts:

```
SysSetupLog
SysSetupCompanyLog
SysRecordTemplateTable
```

```
SysTraceTable
SysTraceTableSQL
SysTraceTableSQLExecPlan
SysTraceTableSQLTabRef
SysUserLog
SysUserInfo
SysInetCSS
```

```
SysInetThemeTable
SysImageTable
SysPersonalization
LanguageTable
Batch
BatchGroup
SysLicenseCodeSort
DocuParameters
SysSecurityFormTable
SysSecurityFormControlTable
SysEvent
KMConnectionType
SalesParmUpdate
SalesParmSubTable
PurchParmUpdate
PurchParmSubTable
SysVersionControlParameters
ReleaseUpdateScripts
ReleaseUpdateScriptDependency
ReleaseUpdateJobStatus
DocuOpenFile
CompanyInfo
```

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

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

Note that changes in Application class are risky and should be made with caution.

Transaction and Idempotency

It is an important requirement that each data upgrade script be idempotent. That is, 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.

Note that 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>);
```

and including 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 simply 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_"
2. Set the configuration property to: "`SysDeletedObjectsXX`" where XX is the next version, for example "41" (for Microsoft Dynamics AX 5.0)
3. Implement the upgrade script that will transform the data into the new data model
4. Test the upgrade script
5. Benchmark the upgrade script
6. *(New for Interim Upgrade):* 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 3.0 field is renamed in 4.0 as DEL_field, it needs to stay in the source until 4.1 if 3.0->4.0 upgrade is supported.

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:

1. Removing a field from a unique index
2. Adding a new unique index
3. 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 one of the following options:

Option 1. Create an upgrade script using the pattern:

```
    set fieldSet = new set(Types::INTEGER);
    ;
    fieldSet.add(fieldNum(<TableName>, <FieldName1>));
    fieldSet.add(fieldNum(<TableName>, <FieldName2>));
    fieldSet.add(fieldNum(<TableName>, <FieldName3>));
    ReleaseUpdateDB::deleteDuplicatesUsingIds(tableNum(<TableName>), 0, fieldSet);
```

Add this script to the shared pre-synchronization jobs. It will execute across companies even for company specific tables.

Note that if the table or fields have changed names or IDs from one version to another, you have to use option 2 instead.

Option 2. When the duplicate records contain values that need more complex logic to clean up, the solution is more involved:

1. 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);
```

2. Create a normal upgrade script. This will move the data according to the new data model.
3. 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);
```

Deleting the Contents of a Table

Situations may occur where you need to delete the contents of a table, 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::deleteDataInTableWithTableId(tableNum(<tableId>));
```

Note this action will delete across companies even for company-specific tables and it is the fastest way to perform the operation.

Alternatively, you can create a pre-synchronization normal script using the delete_from construct.

Upgrading a Table with Table ID or Field ID Changed

When a table or field is renamed, no upgrade scripts are needed. However, when the ID of a table or field is changed, in order to preserve the table and its data, you must call the following methods in a pre-synchronization Start script:

`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)

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

Deleting Configuration Keys

Note: 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 str numberSequence_SQ()
{;
    return 'SQ';
}
```

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

```
num = NumberSeq::newGetNumFromCode(this.numberSequence_SQ(), false);
salesQuotationTable.QuotationId = num.num();
```

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:

1. Monitor and minimize the number of client/server calls.
2. Use record set functions whenever possible.
3. 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:
    1. 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.
    2. Each script can be executed in parallel with other scripts.

1. 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)
2. Take care when you sequence the scripts. For example, do not update data first and then delete it afterwards.
3. 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. Usually10-15 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.
4. Run benchmarking on your script using large datasets to verify your performance is acceptable.
5. 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

1. 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.

1. 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

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.

Note when using set-based operations:

1. 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.
2. With Update_RecordSet you cannot perform inner or left outer joins.
3. Set based statements do not support memo fields.

Please refer to Speeding Up SQL Operations and Maintain Fast SQL Operations in the SDK documentation for list and syntax of set based operations available in Microsoft Dynamics AX 4.0.

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:

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

Examples of the wrong way to 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:

1. 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.

1. 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++

1. 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();
```

1. 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. Note that, 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 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));
}
```

Notes:

1. The example above 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.
2. 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).

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 follows:

```
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.

Oracle Only: Applying NLS_LOWER on String Columns in the WHERE Clause

In Dynamics AX on Oracle, all indexes on string fields are defined as functional indexes as SUBSTR(NLS_LOWER(string column),1, length of column). The column fields in search conditions need to have the **NLS_LOWER** "wrapped" around them, in order to achieve functional correctness and performance.

SUBSTR is optional. If it is omitted the Oracle optimizer will still use the index because SUBSTR does not impact that decision.

```
SELECT PRICECALCID, ITEMID, INVENTDIMID, BOMID, ROUTEID, ROWNUM AS NEWPRICECALCID
FROM BOMCALCTABLE WHERE NLS_LOWER(DATAAREAID) = N'dmo'
```

Note that if customers have de-selected the option in the Server Configuration utility to not use SUBSTR and NLS_LOWER, they will not have functional indexes; they will have regular indexes and thus the SUBSTR and NLS_LOWER is not required.

Structuring an Upgrade Script for Managing SQL Server and Oracle

The following is an example of how you can structure your code so that it can run on both SQL and Oracle:

```
void testDeliveryAddress()
{
    SysSQLSystem            sqlSystem;
    SqlStatementExecutePermission   permission;
    Connection connection;
    Statement statement;
    str sqlStmt;
    str dataareaid;

    void runSQLCode()
    {
       // MS SQL specific code
        sqlStmt = 'UPDATE SALESLINE SET DELIVERYADDRESS = T.DELIVERYADDRESS,';
        sqlStmt += ' DELIVERYNAME = T.DELIVERYNAME,';
        sqlStmt += ' DELIVERYSTREET = T.DELIVERYSTREET,';
        sqlStmt += ' DELIVERYZIPCODE = T.DELIVERYZIPCODE,';
        sqlStmt += ' DELIVERYCITY = T.DELIVERYCITY,';
        sqlStmt += ' DELIVERYCOUNTY = T.DELIVERYCOUNTY,';
        sqlStmt += ' DELIVERYSTATE = T.DELIVERYSTATE,';
        sqlStmt += ' DELIVERYCOUNTRYREGIONID = T.DELIVERYCOUNTRYREGIONID';
        sqlStmt += ' FROM     SALESLINE L,';
        sqlStmt += ' SALESTABLE T';
        sqlStmt += ' WHERE     T.DATAAREAID    = ' + sqlSystem.sqlLiterals(dataareaid);
        sqlStmt += ' AND     L.DATAAREAID     = T.DATAAREAID';
        sqlStmt += ' AND    L.SALESID     = T.SALESID';
        sqlStmt += ' AND    L.DELIVERYADDRESS = ' + sqlSystem.sqlLiterals('');
        permission = new SqlStatementExecutePermission(sqlStmt);
        permission.assert();
        statement.ExecuteUpdate(sqlStmt);
    }
    void runOraCode()
    {
        // Oracle specific code
    }
    ;
    connection  = new Connection();
    statement   = connection.createStatement();
    sqlSystem   = new SqlSystem();
    dataareaid = curExt();
    switch (sqlSystem.databaseId())
    {
        case DatabaseId::MS_Sql_Server   :
            runSQLCode();
            break;

        case (DatabaseId::Oracle)        :
            runOraCode();
            break;

        default                          :
            break;
    }

}
```

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:

1. The stored procedure name should be the same as the method or function that it is replacing.
2. The stored procedure should include the original X++ statements as comments to provide context during testing and troubleshooting.
3. Transactional control statements (BEGIN TRANSACTION, COMMIT) should not be coded in the stored procedure. Transaction management is implemented in X++.
4. The stored procedure must accept a required parameter of DATAAREAID as data type NVARCHAR(3).
5. 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:
    1. @NUMBERSEQUENCE NVARCHAR(20). This will be used as a key to the NUMBERSEQUENCE table to retrieve the next key value and format requirements.
    2. @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:

1. Updates SalesLine with the relevant address information for the salesid.
2. Updates CustInvoicetrans with the address information for salesid.
3. Updates custPackingSlipTrans with the address information for salesid.

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

1. Perform one mass update where possible.

2. 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
```

Number 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 SYSTEMSEQUENCES and 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 Dynamics AX 2009

The RECID allocation algorithm has undergone significant changes in Dynamics AX 5.0. RECID's can be allocated in two different ways:

1. Kernel automatically allocates the RECID during insert and INSERT_RECORDSET
2. 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:

1. 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.
2. An upgrade script uses direct SQL to insert data. In this usage pattern, you need to allocate RECID manually.
3. 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;
   ;
```

Create a new instance of the systemSequence class

```
    s = new SystemSequence();
    s.suspendRecIds( tablenum (AAMyTable) );
    startValue = s.reserveValues( 10, tablenum( AAMyTable ) );


    for ( i = 0; i <10; i++ )
    {
        t.IntFld = i;
        t.RecId = startValue + i;
        t.insert();
    }


    s.removeRecIdSuspension( tablenum ( AAMyTable ) );
}
```

Suspend the RECID allocation by the kernel

Reserve the RECID by passing in the number of id's to reserve. The return value is the starting value of the range you reserved. The API gaurantees that the allocated id's are contiguous.

Assign the RECID to the RECID column

Remove the suspension

Tips on using the RECID allocation API:

1. Once you suspend the RECID allocation for that table, the kernel will not dispense any more RECIDs for that table on that session.
2. The ReserveValues API will guarantee contiguity of the RECID range that is being reserved.
3. If you try to insert an id that has not been reserved, then kernel will raise an exception.
4. If you are trying to assign a RECID without suspending, kernel will raise an exception.
5. 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.

Note that 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 incremented by 1

Retrieve the next value for RECID for this table (by TABID)

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

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

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:

1. The specific number sequence to be used for a specific column.
2. Whether the column is to be left or right justified.
3. 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 FORMAT column of the NUMBERSEQUENCETABLE table.

Notes:

1. 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.

2. 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


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


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


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


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

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 formatting function

As in the previous example, we create an IDENTITY column in the temporary table with initial value of 0

Retrieve the next value from NUMBERSEQUENCETABLE using the NUMBERSEQUENCE key supplied

Details on calling this function follow

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 here from table SQLDICTIONARY
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 temp table to final table. Note that temp table RECID
--is sued to supply values to both QUOTATIONID and REID in 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:

1. Adds the value of the IDENTITY column to the NEXTREC value retrieved from NUMBERSEQUENCETABLE.
2. Formats the result according to the FORMAT column retrieved from NUMBERSEQUENCETABLE.
3. 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:

1. The FORMAT column value from NUMBERSEQUENCETABLE.
2. The integer value to be formatted.
3. 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.