Using DataMove with InterSystems IRIS

Version 2020.3
2021-02-04

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

Using DataMove with InterSystems IRIS
InterSystems IRIS Data Platform Version 2020.3 2021-02-04

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)
Tel: +1-617-621-0700
Tel: +44 (0) 844 854 2917
Email: support@InterSystems.com

Table of Contents

List of Tables

Using DataMove with InterSystems IRIS

This article describes how to use the DataMove process to move existing data associated with an InterSystems IRIS namespace to a new location.

Important: InterSystems highly recommends that you test DataMove with your specific namespaces and databases in a test environment before using it on a live system.

1 Introduction to DataMove

The DataMove process allows you to move existing data associated with an InterSystems IRIS namespace to a different database, by:

- Creating new mappings for the namespace.
- Analyzing the mapping changes to calculate which globals and global subscripts need to be moved.
- Copying the data to the new database or databases.
- Activating the mapping changes.

For example, DataMove allows you to move a global, or portion of a global, from a namespace's default globals database to a different database. You can use the process to move a namespace's globals to a separate database from its routines, to split data across multiple databases, or otherwise move data to new locations based upon updated design decisions over the evolution of an application.

For more information on mappings, see "Add Global, Routine, and Package Mapping to a Namespace" in the *System Administration Guide*.

2 DataMove Limitations

The DataMove process is subject to the following limitations:

- DataMove operates on the mappings in a single namespace at a time.
- DataMove is designed to move data between databases on the same instance. Extra operational care is required for connected systems, such as those in mirrors and distributed cache (ECP-based) clusters.
- The last step of the DataMove process activates the new mappings. The Activation phase should be run while the application is offline, or when the data that was moved is not being accessed or under the control of a lock.
- InterSystems has currently tested moving no more than 20 GB of data at a time.
- DataMove should not be used on applications that use extended global references. Data integrity cannot be ensured if the application mixes the use of mappings and extended references.

For more information on extended global references, see "Namespaces" in *Using ObjectScript*.

3 The DataMove Workflow

The DataMove workflow comprises the following phases:

1. Changes to the namespace mappings are saved in a temporary storage area.
2. A set of data moves is generated from the mappings.
3. The data moves are validated against the specified globals and databases, and sufficient free disk space in the destination databases is confirmed. If any issues are found, the user can correct them and resume the workflow.
4. A background job copies the existing data to the new location and updates a State property as the copy progresses.
5. When the data has been copied, the new mappings are activated in the namespace.
6. After the namespace changes have been successfully activated, you can delete the old source data that has been copied to the new locations.

DataMove maintains a log file DataMoveName.log of all operations in the /mgr directory. You can examine the log file to see the progress of the DataMove or to check for any problems.

4 The DataMove APIs

This section details the API calls required for each phase of the DataMove workflow.

Make note of the following guidelines:

- You must provide any needed scripting or user interface, according to your specific requirements.
- To make use of the needed macros, your code should include the files %syDataMove, %syConfig, and %syJrninc.
- Your code should check the status returned by each method before proceeding to the next API call.
- If any destination databases do not exist, they will be created by the DataMove workflow.
- The workflow should be executed in the %SYS namespace.

4.1 Edit Namespace Mappings

Config.Map.InitializeEdits(Namespace As %String) As %Status

Initializes the temporary storage area for a new set of mapping edits.

Argument:

- *Namespace* is the namespace on which you want to perform the DataMove.

This method must be called before any edits are made and is valid only for the specified namespace.

CAUTION: Calling this method deletes any existing edits in the temporary storage area, which is used for all mapping changes in all namespaces, including those made by the DataMove API and those made using the Global Mappings page of the Management Portal. (See Global Mappings in the "Configuring InterSystems IRIS" chapter of the *System Administration Guide*.) To prevent concurrent mapping changes from causing failures, you must ensure that there are no mapping changes for any namespace in progress when you call **Config.Map.InitializeEdits()**, and that no other mapping changes are initiated until the DataMove process is complete.

Config.MapGlobals.Create(Namespace As %String, Name As %String, ByRef Properties As %String, ByRef CPFFile As %String = "", Flags As %Integer) As %Status

Creates a new global mapping for this namespace in the temporary storage area. You can call this method one or more times, depending on the number of mappings you plan to include in this DataMove.

Arguments:

- *Namespace* is the namespace on which you want to perform the DataMove.
- *Name* is the name of a global to be mapped to a specific database.
- *Properties* is an array of properties needed for this mapping, in particular, the name of the database used for this mapping.
- *CPFFile* is the path and name of the Configuration Parameter File. Leave blank to use the active CPF.
- *Flags* represents any flags needed for this method call and should be set to `$$$CPFSaveMappings`, which indicates that changes are to be saved in the temporary storage area.

Setting the *Name* argument to `"A"` specifies that this mapping affects the entire global `^A`. Setting this argument to `"A(5):A(10)"` specifies that this mapping affects the range of the global with subscripts `^A(5)` up to, but not including, `^A(10)`.

Setting the *Properties* argument to an array `properties` where `properties("Database")` is set to `"USER2"` specifies that the global (or range of a global) is to be mapped to the database `USER2`.

The **Config.MapGlobals.Modify()** and **Config.MapGlobals.Delete()** methods can be used to modify or delete mappings. See the Class Reference for more information.

For more information on defining global ranges, see "Global Mappings" in the *System Administration Guide*.

4.2 Generate DataMove

DataMove.Data.CreateFromMapEdits(Name As %String, ByRef Properties As %String, ByRef Warnings As %String, ByRef Errors As %String) As Status

Creates a new DataMove object, based on map edits in the temporary storage area.

Arguments:

- *Name* is the name of the DataMove object to be created.
- *Properties* is an array of optional properties to be used to create the DataMove object.
- *Warnings* is an array returned with conflicts that do not prevent the DataMove from being performed.
- *Errors* is an array returned with conflicts that do prevent the DataMove from being performed.

If `properties` is an array passed in as the *Properties* argument:

- `properties("Description")` optionally provides a description of the DataMove to be performed.
- `properties("Flags")` optionally provides any flags describing the DataMove operation, such as:
  - `$$$BitNoSrcJournal`, allow copying of non-journaled databases.
  - `$$$BitNoWorkerJobs`, do not use concurrent worker jobs for the DataMove.
  - `$$$BitBatchMode`, run data copy in batch mode.
  - `$$$BitCheckActivate`, call the user-supplied routine **$$CheckActivate^ZDATAMOVE()** to check the application status before activating the mapping changes.

- `properties("LogFile")` optionally specifies a log file name, if other than the default.

The `warnings` array contains a list of mappings where the data being moved is also mapped from another namespace. This array is subscripted by the name of the global or the global range.

The `errors` array contains a list of mappings where a conflict prevents the data from being moved. This array is subscripted by the name of the global or the global range.

For more information on database journaling, see "Journaling" in the *Data Integrity Guide*.

For more information on worker jobs, see *Using the Work Queue Manager*.

For more information on running a process in batch mode, see "Process Management" in *Specialized System Tools and Utilities*.

4.3 Validate DataMove

DataMove.Data.Dispatch(Name As %String , "Validate") As %Status

Validates the DataMove object.

Argument:

- *Name* is the name of the DataMove object.

Validating the DataMove object involves looking all of the specified mappings, checking the source and destination databases, and making sure the destination globals do not already exist. Any errors are reported in the status.

DataMove.Data.Dispatch(Name As %String , "ValidateSizes") As %Status

Makes sure sufficient space exists for the data specified by the DataMove object to be copied.

Argument:

- *Name* is the name of the DataMove object.

Validating sizes for a DataMove object involves determining the amount of data to be copied and ensuring enough space exists in the destination database. Any errors are reported in the status.

4.4 Start DataMove

DataMove.Data.DispatchJob(Name As %String, "Copy") As %Status

Starts the DataMove copy, which handles the actual moving of data.

Argument:

- *Name* is the name of the DataMove object.

This method starts the initial DataMove copy as a background job, which, if `$$$BitNoWorkerJobs` is not set, may create other worker jobs. Once the initial copy is completed, the job will continue in a loop, applying journal files for each range to the destination database.

You can verify that the initial copy has completed by calling **DataMove.Data.Exists()** to instantiate the `DataMove.Data` object, and then verifying that the State property is greater than `$$$CopyDone`.

Also, the **DataMove.Data.Display()** method displays a brief status of the DataMove object, including the State of the DataMove.

4.5 Finish DataMove and Activate Mapping Changes

DataMove.Data.Activate(Name As %String, Display as %Boolean,Timeout As %Integer = 120) As %Status

Finishes the DataMove and activates the namespace mapping changes.

Arguments:

- *Name* is the name of the DataMove object.
- *Display* should be set to `true` if you want to see progress messages.
- *Timeout* is number of seconds to wait for the Copy job to finish running an apply journals operation before proceeding.

This method stops the DataMove background job, finishes processing any journal files, writes the mapping changes to the CPF, and activates the mapping changes. It momentarily sets switch 10 to prevent other processes from interfering with the execution.

If `$$$BitCheckActivate` is set, the method will call a user-supplied routine to check the application status before continuing. If **$$CheckActivate^ZDATAMOVE()** does not return `1`, the method will quit.

For more information on switch 10, see "Using Switches" in *Specialized System Tools and Utilities*.

Note: **DataMove.Data.Activate()** checks the State property of the DataMove object to make sure that the initial copy is complete before proceeding. It also checks the JRNApplied property to make sure that at least one journal pass has been completed. If **DataMove.Data.Activate()** does not run to completion, see the "Troubleshooting" section of this document.

4.6 Delete Source Globals and Clean Up DataMove Class

DataMove.Data.Dispatch(Name As %String , "DeleteSrcGlobals") As %Status

Deletes the globals that have been copied in the DataMove.

Argument:

- *Name* is the name of the DataMove object.

This method deletes all globals in the source database that have been copied to a destination database.

If `$$$BitNoWorkerJobs` is not set, this method creates a worker job for each global to be deleted.

DataMove.Data.Dispatch(Name As %String , "Finish") As %Status

Completes the DataMove process.

Argument:

- *Name* is the name of the DataMove object.

This method writes a success or failure message to the log file, closes the log file, and sets the State property of the DataMove object to `$$$Done`.

DataMove.Data.Dispatch(Name As %String , "Delete") As %Status

Cleans up the DataMove process.

Argument:

- *Name* is the name of the DataMove object.

This method deletes the DataMove object and cleans up any temporary storage.

4.7 Other API Calls

DataMove.Data.Exists(Name As %String, ByRef Obj As %ObjectHandle, ByRef Status As %Status) As %Boolean

Checks that the DataMove exists and returns the DataMove object.

Arguments:

- *Name* is the name of the DataMove object.
- *Obj* is the returned handle to the DataMove object.
- *Status* is the returned status.

This method returns a DataMove object and lets you access its properties. Useful properties include State (the current state of the DataMove) and JRNApplied (the number of journal passes applied).

Returns `1` if the DataMove exists and `0` otherwise.

For a complete list of DataMove states, see "DataMove States".

DataMove.Data.Display(Name As %String) As %Status

Displays a summary of the DataMove object.

Argument:

- *Name* is the name of the DataMove object.

This method displays the current State of the DataMove and the moves created from the defined mappings. If the current state is `$$$JRNApply` or `$$$JRNApplyDone`, the number of journal passes completed is displayed after the State, in parentheses.

For a complete list of DataMove states, see "DataMove States".

DataMove.Data.StopCopy(Name As %String) As %Status

Stops the DataMove copy job.

Argument:

- *Name* is the name of the DataMove object.

This method stops the DataMove copy background job, allowing you to gracefully stop the copy after it is in process. You can then do a rollback with **DataMove.Data.Dispatch(Name, "Rollback")**.

DataMove.Data.Dispatch(Name As %String , "Rollback") As %Status

Rolls back the DataMove.

Argument:

- *Name* is the name of the DataMove object.

This method deletes any globals that have been copied to destination databases by the DataMove copy job. This method can be used to abort the DataMove or recover from an error in the copy process.

DataMove.Data.StopCopy() must be run before calling this method.

After doing the rollback, you can start over or delete the DataMove with **DataMove.Data.Dispatch(Name, "Delete")**.

5 List of DataMove States

The DataMove object keeps track of its progress through the workflow by means of the State property. You can inspect this property to monitor its progress, or to troubleshoot any issues that arise, by calling the methods **DataMove.Data.Exists()** or **DataMove.Data.Display()**.

Table 1: List of DataMove States

State	Integer Value
$$$Done	1
$$$NotStarted	2
$$$Started	3
$$$Size	4
$$$SizeDone	5
$$$Copy	6
$$$CopyDone	7
$$$JrnApply	8
$$$JrnApplyDone	9
$$$CPFUpdate	11
$$$CPFUpdateDone	12
$$$NSPActivate	13
$$$NSPActivateDone	14
$$$Delete	15
$$$DeleteDone	16
$$$Cleanup	17
$$$CleanupDone	18

6 DataMove Example

This ObjectScript class moves global range `^A(5)` up to `^A(10)` in namespace USER to database USER3 and the remainder of the `^A` global to database USER2, using a DataMove object named `DMTEST`.

For demonstration purposes, this example does not include comprehensive error checking. Check the returned status after each API call.

```
Include (%syDataMove, %syConfig, %syJrninc)

/// Sample class to illustrate using DataMove to move globals.
/// Run MapEdit(), Activate(), and CleanUp() in order.
Class DataMove.Sample Extends %RegisteredObject
{

/// Edit a global mapping, generate the DataMove, and start Copy JOB
ClassMethod MapEdit(Datamove As %String = "DMTEST") As %Status
{
```

```
    set namespace="USER" // namespace that contains ^A global

    // Initialize temporary storage area to hold mapping edits
    set status = ##class(Config.Map).InitializeEdits(namespace)

    // Create mapping for global ^A to database USER2 in temporary storage area
    kill properties
    set properties("Database") = "USER2"
    set status = ##class(Config.MapGlobals).Create(namespace, "H", .properties, , $$$CPFMappingEdit)
    if $$$ISERR(status) quit // should check status after every API call

    // Create mapping for global range ^A(5) up to ^A(10) to database USER3 in temporary storage area
    set properties("Database") = "USER3"
    set status = ##class(Config.MapGlobals).Create(namespace, "H(5):(10)", .properties, ,
$$$CPFMappingEdit)

    // Create new DataMove object, based on map edits in the temporary storage area
    kill properties, warnings, errors
    set properties("Description") = "DataMove test"
    // Flags could also include $$$BitCheckActivate if we want to call $$CheckActivate^ZDATAMOVE()
    set properties("Flags")=$$$BitNoWorkerJobs+$$$BitNoSrcJournal+$$$BitBatchMode+$$$BitCheckActivate
    set status = ##class(DataMove.Data).CreateFromMapEdits(Datamove, .properties, .warnings, .errors)
    if $data(warnings) write ! zwrite warnings
    if $data(errors) write ! zwrite errors

    // Validate DataMove object
    set status = ##class(DataMove.Data).Dispatch(Datamove, "Validate")

    // Make sure enough space exists to copy globals
    set status = ##class(DataMove.Data).Dispatch(Datamove, "ValidateSizes")

    // Display the status of the DataMove and the moves created from the defined mappings
    write ! do ##class(DataMove.Data).Display(Datamove)

    // Start the DataMove copy job
    set status = ##class(DataMove.Data).DispatchJob(Datamove,"Copy")

    quit status

}

/// Finish DataMove and activate Namespace change
ClassMethod Activate(Datamove As %String = "DMTEST") As %Status
{
    // Check that DataMove exists and return the DataMove object
    if '##class(DataMove.Data).Exists(Datamove, .dmObject, .status) {
        write !,"No such DataMove "_Datamove
        quit status
    }

    // Check state of DataMove before doing final activation
    if (dmObject.State < $$$CopyDone) || (dmObject.State > $$$NSPActivate) {
        write !,"Wrong State for DataMove "_Datamove
        quit $$$ERROR($$$MGBLStateWrong, "Activate()", dmObject.State)
    }

    // Finish the DataMove and activate the namespace mapping changes
    set status = ##class(DataMove.Data).Activate(Datamove)

    quit status

}

/// Delete copied globals, do DataMove clean up and delete object
ClassMethod CleanUp(Datamove As %String = "DMTEST") As %Status
{
    // Delete the source globals that have been copied in the DataMove
    set status=##class(DataMove.Data).Dispatch(Datamove, "DeleteSrcGlobals")

    // Complete the DataMove process
    set status=##class(DataMove.Data).Dispatch(Datamove, "Finish")

    // Delete the DataMove object and clean up any temporary storage
    set status=##class(DataMove.Data).Dispatch(Datamove, "Delete")

    quit status
}

/// Stop DataMove copy job, rollback changes, delete DataMove
ClassMethod Stop(Datamove As %String = "DMTEST") As %Status
{
    // Stop DataMove copy job
    set status = ##class(DataMove.Data).StopCopy(Datamove)

    // Roll back the DataMove
```

```
    set status = ##class(DataMove.Data).Dispatch(Datamove, "Rollback")

    // Delete the DataMove object and clean up any temporary storage
    set status = ##class(DataMove.Data).Dispatch(Datamove, "Delete")

    quit status
}

}
```

7 Troubleshooting DataMove

The DataMove background job should be restartable by calling **DataMove.Data.DispatchJob(Name, "Copy")**, if it encounters an error or if the system stops for some reason. The DataMove object keeps track of which globals have been copied, whether the initial copy has completed, and how many journal passes have been applied.

If **DataMove.Data.Activate()** detects an error, or if the system crashes while it is running, check the State property of the DataMove object or examine the log file.

- If **DataMove.Data.Activate()** stops before the namespace configuration is updated (`State < $$$CPFUpdateDone`), resume the DataMove workflow from the "Start DataMove" section of this document.
- If **DataMove.Data.Activate()** detects an error before the namespace activation (`State < $$$NSPActivateDone`), it should back out the CPF changes and leave the system running with the original namespace mappings. Resume the DataMove workflow from the "Start DataMove" section of this document.
- If **DataMove.Data.Activate()** stops after the namespace configuration is updated, then the system will likely start with the new CPF and activate the new mappings. After verifying that the new mappings are activated, resume the DataMove workflow from the "Delete Source Globals and Clean Up DataMove Class" section of this document.
- If **DataMove.Data.Activate()** stops while updating the CPF, verify the contents of the CPF and that the namespace mappings are activated. It is possible, but unlikely, to have a partially written (corrupt) CPF.