JOURNAL BATCH POSTING

Microsoft Dynamics® AX 2009

Journal Batch Posting

White Paper

This white paper gives a brief overview of the updated batch journal posting engine, including tips for maximizing posting performance.

Date: September 29, 2008

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

Table of Contents

Introduction

Microsoft Dynamics AX 2009 introduces a heavily updated batch framework, with a brand-new client-less execution mode that has a modern execution engine for parallelizing tasks that can even have dependencies.

In order to maximize posting throughput, the journal batch posting framework was modified to leverage these capabilities that are newly introduced in Microsoft Dynamics AX 2009. This white paper covers the general architecture of the updated batch posting framework and how it can be used in real-world scenarios.

Understanding Journal Batch Posting

Journal batch posting functionality exists in Microsoft Dynamics AX for the purpose of batching together the posting of a set of journals, which provides a potentially reduced wait time and ease of use. All journals besides the fixed asset depreciation journals are posted through **General Ledger > Periodic > Post journals**; depreciation journals are posted through **General Ledger > Periodic > Post depreciation book journals**.

A journal, in general, encapsulates transactions, such as invoices, payments, and so on, which are known as *lines*. Each one of these lines belongs to a group of lines known as a *voucher*, which must balance across itself; that is, the debit and credits across that voucher must add to zero. Every journal has *n* number of lines and *n* or less vouchers.

When a journal balances across all of its vouchers and has been approved (if in an approval process), it may be *posted*, which then commits the transactions across the system. Based on application settings (in particular, the journal "line limit"), during the posting process a journal may be *split* between its voucher boundaries in order to create several smaller journals that still balance. For example, if the user has configured journals to have a line limit of 1,000 and a journal of size 5,000 lines is sent to be posted, the journal will be split into roughly five 1,000 line journals (pending where voucher boundaries fall), after which each of the five journals is posted separately.

Design

The high-level design goal of the updated batch posting engine was to refactor the existing engine to leverage the new batch framework for maximum performance.

Throughput

A major focus of the design for the updated batch posting engine was to make journal posting highly parallel so as to maintain a high load across an Application Object Server (AOS) farm. If there are 10 AOSs that are tagged to run batch tasks, all 10 AOSs should be running tasks in parallel as much as possible. Care must be taken when scheduling these tasks because you can actually experience reduced performance if too many tasks are scheduled due to the overhead in the batch system that results from maintaining the state

of the tasks. This design addressed this performance issue through a concept of "bundling," which is discussed in Appendix I.

The core batch posting engine follows a model where the batch job queues a task for each journal that needs to be posted. Each task then determines if a journal can be posted directly, or if it needs to be split first. If the journal can be posted directly, the task will call out to the appropriate posting procedures; if the journal cannot be posted directly, the journal is split with each split journal being posted by additional, new tasks. This model is shown in Figure 1.

The new batch posting model is in contrast to the one that existed in Microsoft Dynamics AX 4.x (shown in Figure 2), where parallelism was achieved by essentially throwing as many clients as possible at a set of journals, with no communication between clients, thereby relying on state flags (maintained on the table *LedgerJournalTable*) and database "update conflicts". This led to poor scaling and even occasional errors if the journal ranges overlapped, because of a flawed locking algorithm.

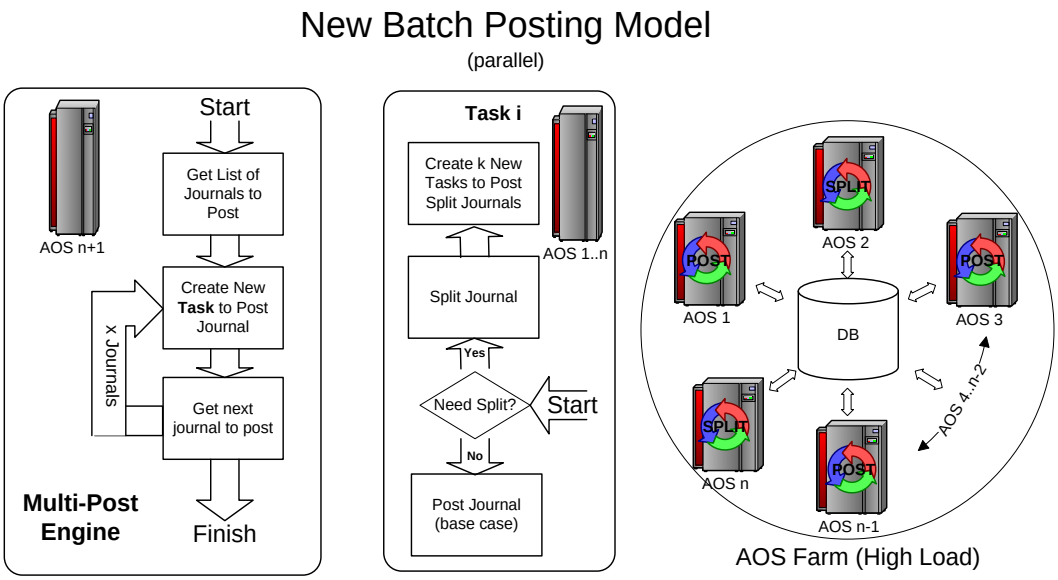


Figure 1: New batch posting model

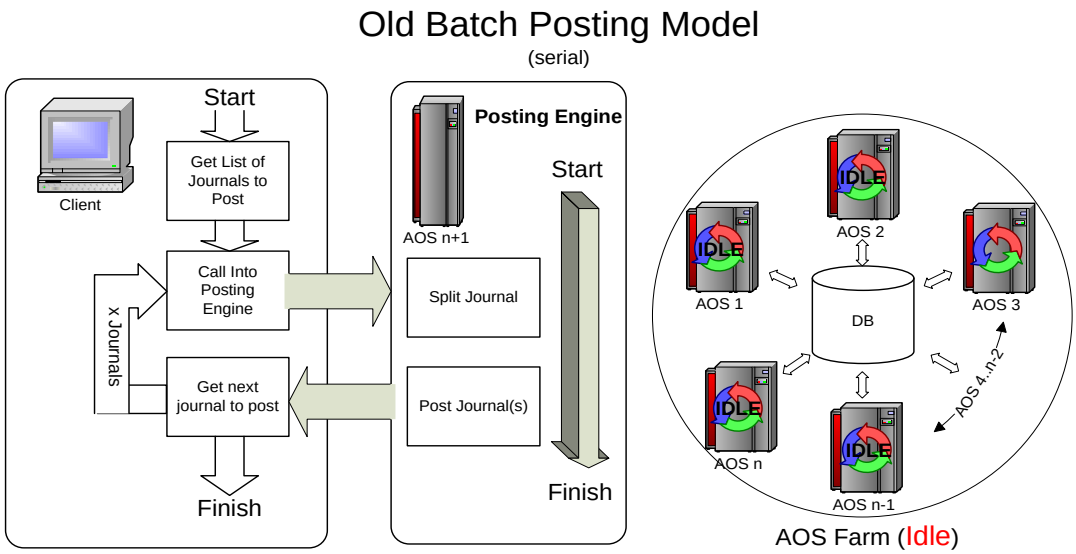


Figure 2: Microsoft Dynamics AX 4.x batch posting model

Batch Task Bundling

As noted in the previous section, the batch system will incur large amounts of overhead if too many (on the order of hundreds of thousands or millions) batch tasks are scheduled. To work around this, a shim framework known as the "Batch Task Bundler" was introduced to simplify the management of the amount of batch tasks that are scheduled at any one time by any one batch job.

This framework provided an approximate 25 percent performance improvement in high task volume scenarios.

For an overview of the bundling framework, see Appendix II.

Recovery Model

The batch framework allows a user to restart failed jobs (due to power outages, and so on). Furthermore, since the transactional level in the posting classes is such that there will never be a partially posted journal—either a journal posts or it does not post—a user can restart a posting batch job as many times as necessary until it completes successfully.

Implementation

We introduced three new classes: *LedgerPostBatch*, *LedgerJournalPostBatch,* and *AssetDepBookJournalPostBatch*. *LedgerPostBatch* extends *RunBaseBatch* and provides the structure for *LedgerJournalPostBatch* and *AssetDepBookJournalPostBatch* through the *validateJournal*, *mustSplitJournal* and *getJournalsToPost* methods, which must be overridden by each child class.

LedgerJournalPostBatch and *AssetDepBookJournalPostBatch* are capable of posting individual journals (specified in *new* through parameters) both inside and outside of a batch

process. For example, you could create code that instantiates a *LedgerJournalPostBatch* object and calls *run* on that instance directly, or you could create code that makes an instance and then add it to a batch job's task list.

The posting classes *LedgerJournalPost* and *AssetDepBookJournalPost* have been modified so that their posting logic is exposed through a public static method called *postJournal*. This exposure simplifies the calling process because an *Args* does not have to be constructed and passed in on a *main()* call. Also, the *LedgerJournalDistribute* (class) and *AssetDepBookJournalTable* (table) objects have been modified so that client messages such as *SysInfoAction** can be suppressed. *LedgerJournalDistribute* adds a class variable called *suppressClientMessages,* which can be initialized through the *new()* call and set by way of a *parm* method; the table simply takes *suppressClientMessages* as a parameter in its *distributeJournalLines* method. Note that these new core classes all have XML documentation. For a diagram of these new core classes, see Figure 3.

As in the Microsoft Dynamics AX 4.x batch posting engine, the main entry point for batch posting is the *run* method on *LedgerJournalMultiPost* (ledger journals) and *AssetDepBookMultiPost* (asset depreciation book journals). It is here in the *run* method that the first level of *LedgerPostBatch* (and derivative) tasks is scheduled to be posted.

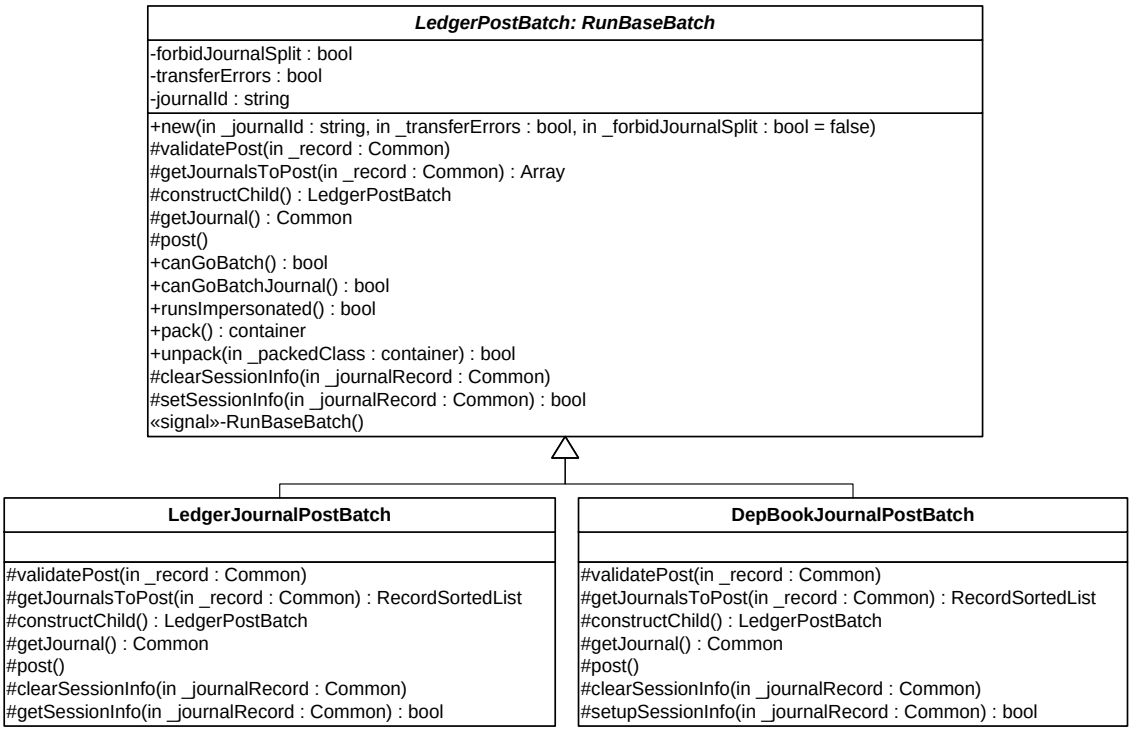


Figure 3: New core classes UML diagram

Application Usage

Setup

To properly leverage the new batch posting engine, you must be running Microsoft Dynamics AX 2009 and have the following configuration:

Anoint one or more AOSs as a batch server by using the **Administration > Setup > Server configuration** form (see Figure 4). Note that you can also configure batch groups as necessary through **Administration > Setup > Batch groups**, although this step is not required.

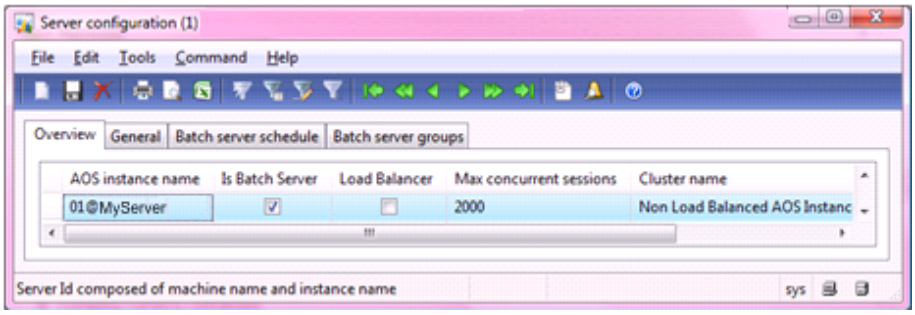


Figure 4: Server configuration form

Basic Usage

Batch posting requires two steps. First, journals are created through users, Application Integration Framework (AIF), or some other process. The journals may be configured to use "line limits" (also known as "splitting"). Second, the journals are posted by using the **General Ledger > Periodic > Post journals** form or by using multi-select posting from the journal forms.

Batch Posting

Batch posting for financial journals can be performed in either of two ways. The classical way is to navigate to the **General Ledger > Periodic > Post journals** form directly, and then select a list of journals to post (see Figure 5). A new way that is introduced in Microsoft Dynamics AX 2009 is to use multi-selection on the journal form by holding the **CTRL** key while selecting multiple rows. When a user clicks the **Post** button during a multi-selection, the user will be prompted with a batching dialog instead of the usual "direct" posting action.

Regardless of which batch posting method is used, when the user is prompted with the batch dialog (see Figure 5 and Figure 7), batch processing must be opted into by clicking the **Batch** tab, and then by selecting the **Batch processing** check box.

Important: Failure to opt in to use batch processing will cause the batch posting to revert to a slower single-threaded mode that runs on the user's client.

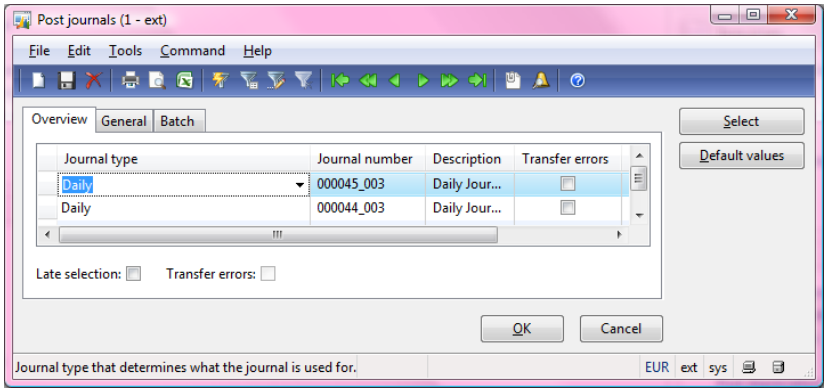


Figure 5: Classical batch posting of journals by using the Post journals form

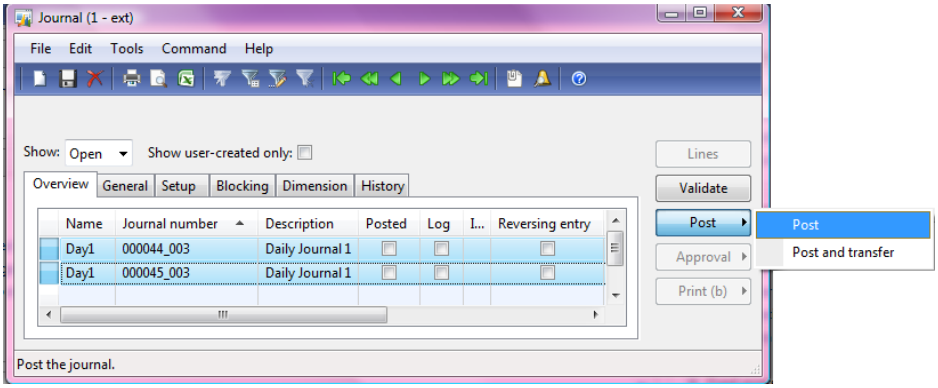


Figure 6: Batch posting by using multi-select in the Journal form

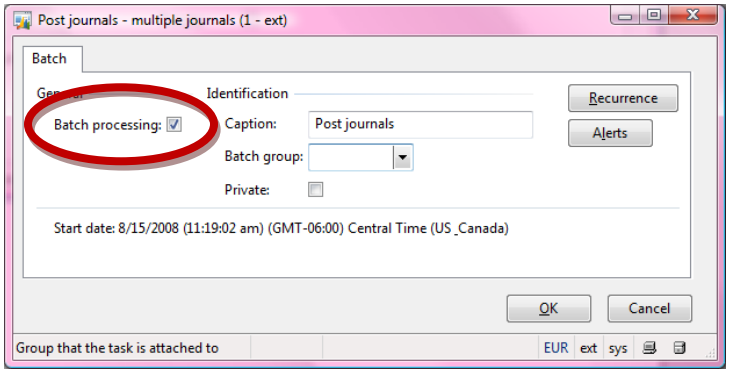


Figure 7: Ensure that the Batch processing check box is selected

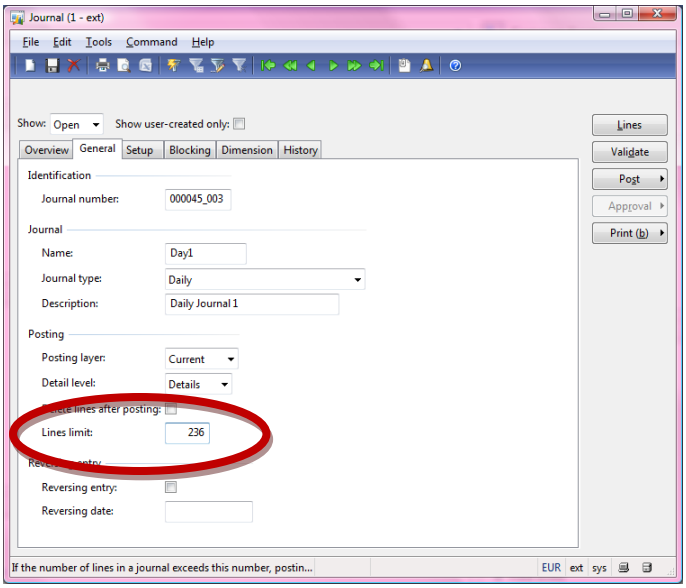
Line Limits

Journal line limits allow the user to specify an approximate maximum size for a journal during posting. If a line limit is specified, the system will split the journal along voucher boundaries, creating new journals that are greater than or equal to the line limit that is specified. Note that the effective split journal size may be vastly different than the line limit that is specified if you are using large vouchers or voucher sizes that do not evenly divide the line limit that is specified. For example, if you specify a line limit of 400, but the journal to be posted has a single 5,000 line voucher, no splitting will be performed because a

voucher cannot be split. In many cases, these large vouchers could be broken into several smaller vouchers during user entry, thereby potentially increasing posting throughput. If you find the system to be throughput limited, investigate the possibility of having journals entered across multiple vouchers instead of inside of one large voucher.

Setting Line Limits

Line limits are set on the journal forms (for example, **General Ledger > Journals > General journal**) under the **General** tab.



Line limit option on *LedgerJournalTable*

Optimizing Line Limits

The optimum line size will vary, but a good general rule to follow is to have the net number of journals slightly greater than the number of maximum batch threads, so as to maximize throughput without overwhelming the batch system. The maximum number of batch threads can be found on the **Batch server schedule** tab on the **Administration > Setup > Server configuration** form (the default is 8 per AOS). Be sure to take into account the Start and End times and to sum across the AOSs that belong to the specified batch group, if a group is specified at all.

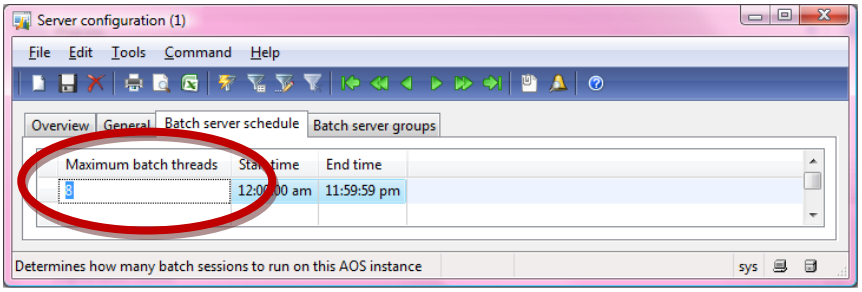


Figure 8: View of maximum batch threads

Line Limit Example

Suppose a journal is to be posted on a batch group with two AOSs. One AOS has 8 threads; the other has 4 threads. The journal to be posted has 7,744 lines with an average voucher size of 121. What would be a good line limit to choose for good posting throughput?

Answer: Because there are 7,744 lines and the average voucher size is 121 we have 7,744/121 = 64 vouchers. If a line limit of 121 is chosen, the system will create 63 journals (the original journal is left alone). This is clearly not an optimal number, as there are only 12 total batch threads available. Ignoring the average voucher size for a moment, we can see that an approximately good size would be 7,744 (total lines) / 12 (batch threads) = ~645 lines per batch thread. Taking that number divided by the average voucher size (645/121), we see that 5 average-sized vouchers would fit into the "approximately optimal" size. However, because some vouchers will post more quickly than others, creating 12 journals (one journal per batch thread) will potentially leave threads free. Therefore, choose a line limit of 4*121 = 484 (instead of 645), which will net approximately 16 tasks, helping keep the batch thread pipeline full.

Note that this example is pedagogical. If you want to directly compute a line limit, simply determine the number of threads that you would like to create, and then divide the number of lines by that amount. To verify that the line limit will have the net effect that you want, verify that the average number of lines per voucher is less than the line limit.

Summary

The updated batch posting framework has the potential to greatly improve posting performance if it is used correctly. Understanding the architecture of the framework (see the Design section) and how to optimize around it (see the Application Usage section) will help you meet that end.

Appendix I

Terms and Definitions

Term/Acronym	Definition
AOS batch threads or *batch threads*	The concurrent batch processing units for a batchable AOS. For example, an AOS can have 10 batch threads.
Batch (noun)	A process run and managed by the batch framework; can be programmed to have recurrences, and so on.
Batch task or *task (noun)*	A work item owned by a batch that can have dependencies on other tasks as defined in Batch Framework documentation.
Batchable (adjective)	Indicates that an AOS can run batch processes. For example, that AOS is batchable.
Journal (noun)	An entity that encapsulates transactions.
Journal line limit (noun)	An approximate upper limit on the size of a journal.
Journal lines (noun)	A transaction inside of a journal.
Post (verb)	To commit the transactions specified in a journal.
Voucher (noun)	A subset of lines in a journal that balance to zero.

Appendix II

Batch Bundling Framework

Introduction

The improved, head-less batch framework was added to Microsoft Dynamics AX 2009 during the first couple of milestones of the release. The updated batch framework allows for finer granularity of processes and therefore improved parallelization across multiple AOS machines.

During performance testing, it was discovered that this ability to have very granular processes could potentially add a lot of overhead to the system, because the batch system spent a significant portion of its time monitoring and adjusting the state of these tasks if the number of processes (tasks) became large.

The Financials (FIM) team, working with the global performance team, developed an additional adapter framework that allows developers to maintain granularity of processes (high number of logical tasks), while limiting the number of physical tasks being maintained by the batch system.

FIM found that one of its release criteria processes—batch multi-posting of 100,000 journals (using 100,000 logical tasks)—received a 25 percent performance improvement by leveraging this new "bundling" framework.

Framework Usage Prerequisites

There are three prerequisites of the framework that the *RunBaseBatch*-derivative classes (tasks) that are to be executed must meet:

1) The bundled tasks must be self recoverable.
    a. The process orchestrated by the task must persist state to the database—or be independent of prior state—so that it can be executed any number of times without ill effects.

2) The bundled tasks must be independent.
    a. There can be no top-level inter-task order-of-execution dependencies.

3) The bundled tasks must fully implement the *SysPackable* interface.
    a. The *pack* and *unpack* methods as-well-as the static *create* method must be implemented. Every class field accessed in the task's *run* method must be fully serialized.

Note: The third prerequisite may easily be met by implementing the *SysPackable* interface on the *RunBaseBatch*-derivative classes (tasks) that are to be bundled.

Framework Concepts

Normally, each task is tracked individually by the batch framework, which can add large amounts of overhead to the batch system if there are a large number of tasks. The batch bundling framework that is introduced in Microsoft Dynamics AX 2009 cuts out the majority of this overhead by *bundling* (or grouping) these tasks together, so that the number of items to be tracked by the batch framework is greatly reduced.

A group of tasks are encapsulated by a *bundle*. A *bundle* represents the minimum unit of work that may be executed directly by the batch framework. This abstract container (bundle) stores and executes batch tasks instead of the batch framework. The bundled tasks are still executed in the context of the greater batch process and therefore are unaware of any extra layer between themselves and the batch framework.

Each task that is to be executed is added to a custom bundle—defined by the developer—which defines maximum size, as well as a few other properties. When the bundle reaches its maximum size, it is deemed fully *assembled* and it is set aside so that it can be scheduled later.

A bundle uses the *bundle assembler* to bundle its tasks. The bundle assembler is generic and works with all bundles. When all tasks have been bundled by the *bundle assembler*, they may be scheduled for execution. The bundled tasks are not directly executed by the batch framework, but rather by the bundle itself. This behavior is what leads to the first two of the three prerequisites that are listed in the previous section.

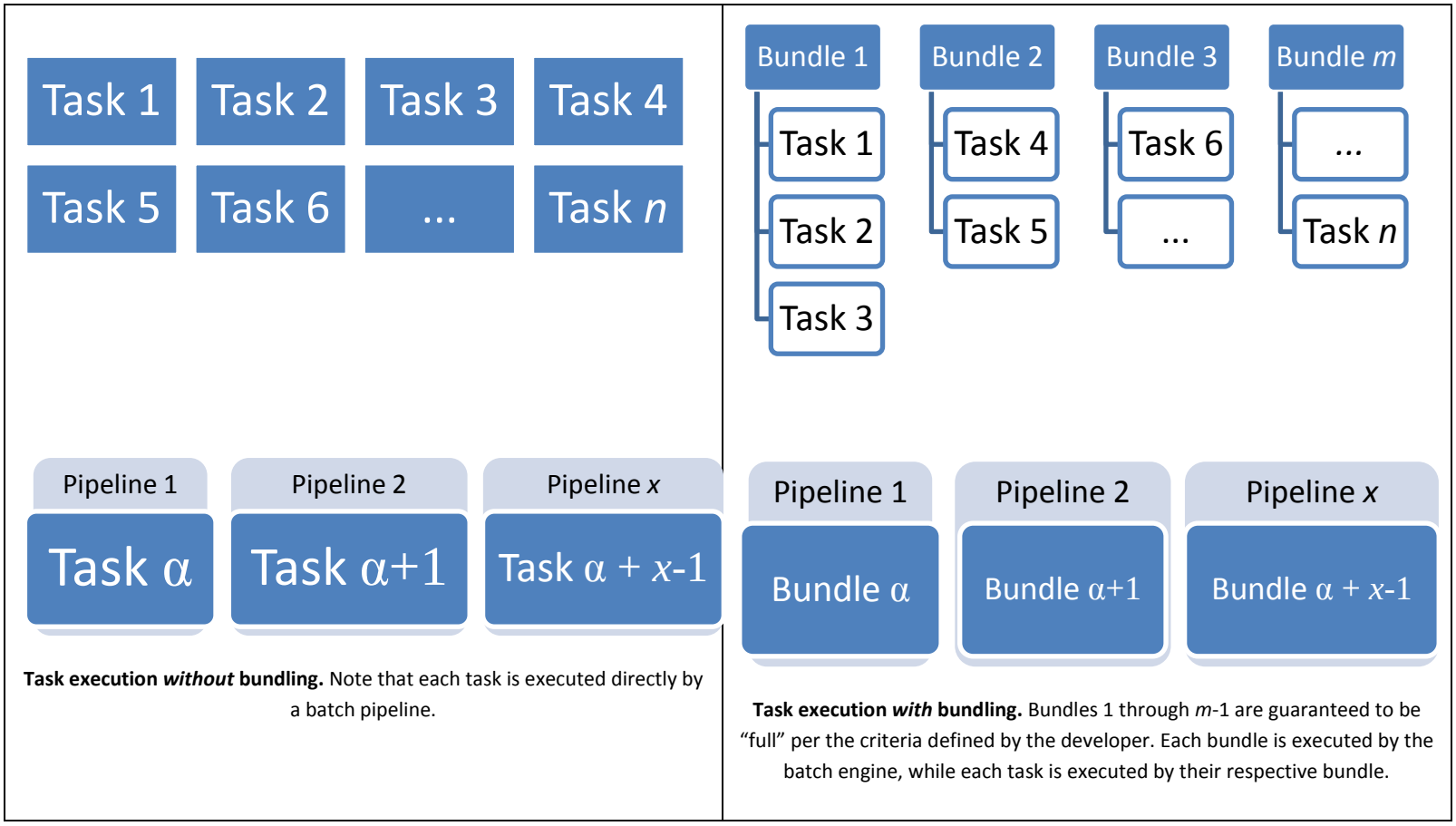


Figure 1—Task bundling concept

Core Framework Classes

Two classes form the core of the framework. The *BatchTaskBundleAssembler* class acts as the *bundle assembler*. The *BatchTaskBundle* abstract class forms the core for each bundle that can be assembled by the assembler.

```
BatchTaskBundleAssembler

BatchTaskBundle
```

Listing 1—Core framework classes

Leveraging the Framework

There are three main steps that you must take to leverage the framework.

1) You must define a concrete bundle that extends *BatchTaskBundle,* which defines metrics for your tasks, such as relative size, and so on.
    a. The following methods must be implemented on BatchTaskBundle-derivative classes: *getSupportedTasksList*, *isFull,* and *isTaskLargerThanMyBaseCapacity*.

2) You must implement *SysPackable*—including *pack*, *unpack* and *create*—on the tasks that are supported by the concrete bundle that was created in step 1.

3) You must modify the area of code creating runtime tasks (from step 2) to use the *BatchTaskBundleAssembler* class instead of the batch engine for scheduling the created tasks.

Use Case

The FIM team leveraged the bundling framework to achieve a 25 percent improvement in batch multi-posting performance.

Batch multi-post uses four main classes:

1) LedgerJournalMultiPost
    a. This is the entry point for batch multi-posting. This class extends *RunBaseMultiParm,* which in turn, extends *RunBaseBatch*.

2) LedgerPostBatch
    a. This is an abstract class that forms the base for the *LedgerJournalPostBatch* and *AssetDepBookJournalPostBatch* classes. This class extends *RunBaseBatch*.

3) LedgerJournalPostBatch
    a. This is a concrete implementation of *LedgerPostBatch* that performs posting for Ledger journals. This class extends *LedgerPostBatch*.

4) AssetDepBookJournalPostBatch
    a. This is a concrete implementation of *LedgerPostBatch* that performs posting for asset depreciation book journals. This class extends *LedgerPostBatch*.

Concrete Bundle

The concrete *LedgerPostBatchTaskBundle* bundle (extending the abstract *BatchTaskBundle* class) was defined specifically for batch journal posting.

The *LedgerPostBatchTaskBundle* bundle defines when it has become "too full" of *LedgerPostBatch*-derivative tasks, as well as a few other quantifiers.

Note that the only time that you need to specify the bundle is during the construction of the *BatchTaskBundleAssembler* class as follows:

```
BatchTaskBundleAssembler::construct(this.parmCurrentBatch(), classnum(LedgerPostBatchTaskBundle))
```

After the bundle has been specified, the assembler will automatically create and use bundles of that type.

Implementation of *SysPackable* on *LedgerPostBatch* and Derivatives

In order to use the bundling framework, *SysPackable* must be implemented on all supported tasks. This includes *pack*, *unpack,* and *create*.

In this use case, *pack* and *unpack* are defined on the *LedgerPostBatch* class, while *create*—which is a static method—is defined on both *LedgerJournalPostBatch* and *AssetDepBookJournalPostBatch*.

SysPackable must be implemented for the bundles (and their bundled tasks) to be persisted to the database correctly.

For code examples, see Appendix A, Listings A.3–A.5.

Entry Point

The initial use case of the bundle framework was the FIM team's batch multi-post engine, whose entry point is defined by the *LedgerJournalMultiPost* class' *run* method.

The *LedgerJournalMultiPost.run* method creates *LedgerJournalPostBatch* batch runtime tasks, and then schedules them for execution.

Originally these tasks were scheduled directly by the batch framework as shown in Appendix A, Listing A.1. It was re-written to schedule the tasks by using the bundling engine as shown in Appendix A, Listing A.2.

Appendix A

Batch Bundling Framework Code Snippets

```
public void run()
{
    BatchHeader             batchHeader; // Header for batch job that hosts batch tasks
    LedgerJournalPostBatch  ledgerJournalPostBatch; // Batch task that will post the individual journals
    LedgerJournalParmPost   ledgerJournalParmPost; // Table that stores which journals to post
    ;

    if (this.isInBatch())
    {
          batchHeader = BatchHeader::construct(this.parmCurrentBatch().BatchJobId);
    }

       ledgerJournalParmPost = LedgerJournalParmPost::findByParmId(parmId);

    while (ledgerJournalParmPost.RecId != 0)
    {
        try
        {

            ledgerJournalPostBatch = LedgerJournalPostBatch::construct(ledgerJournalParmPost.LedgerJournalId,
                                                                       ledgerJournalParmPost.TransferErrors);

            if (batchTaskBundleAssembler != null)

            if (batchHeader != null)
            {
                // This is being run as batch, so create a new task.
                   batchHeader.addRuntimeTask(ledgerJournalPostBatch, this.parmCurrentBatch().RecId);
            }
            else
            {
                // This is being run outside of a batch, so just post the
                // journal now.
                ledgerJournalPostBatch.run();
            }
        }
          // ...
          // Catch logic...
          // ...

        next ledgerJournalParmPost;
    }


       if (batchHeader != null)
       {
              // This is in batch, so schedule the queued tasks.
              batchHeader.save();
       }
}
```

Listing A.1—Original *LedgerJournalMultiPost\run* method

```
public void run()
{
    BatchTaskBundleAssembler    batchTaskBundleAssembler; // Bulks together tasks
    LedgerJournalPostBatch      ledgerJournalPostBatch; // Batch task that will post the individual journals
    LedgerJournalParmPost       ledgerJournalParmPost; // Table that stores which journals to post
    ;

    if (this.isInBatch())
    {
        // This is running in batch, so construct an object for bulking the
        // tasks.
        batchTaskBundleAssembler = BatchTaskBundleAssembler::construct(this.parmCurrentBatch(),
classnum(LedgerPostBatchTaskBundle));
    }

    ledgerJournalParmPost = LedgerJournalParmPost::findByParmId(parmId);

    while (ledgerJournalParmPost.RecId != 0)
    {
        try
        {

            ledgerJournalPostBatch = LedgerJournalPostBatch::construct(ledgerJournalParmPost.LedgerJournalId,
ledgerJournalParmPost.TransferErrors);

            if (batchTaskBundleAssembler != null)
            {
                // This is being run as batch, so add the task to a batch bundle.
                batchTaskBundleAssembler.addTask(ledgerJournalPostBatch);
            }
            else
            {
                // This is being run outside of a batch, so just post the
                // journal now.
                ledgerJournalPostBatch.run();
            }
        }
          // ...
          // Catch logic...
          // ...

        next ledgerJournalParmPost;
    }

    if (batchTaskBundleAssembler != null)
    {
        // This is in batch, so schedule the queued tasks.
        batchTaskBundleAssembler.scheduleBundledTasks();
    }
}
```

Listing A.2—New *LedgerJournalMultiPost\run* method leveraging the bundling framework

```
public container pack()
{
    return [#CurrentVersion, #CurrentList];
}
```

Listing A.3—*LedgerPostBatch\pack*

```
public boolean unpack(container packedClass)
{
    Version version = RunBase::getVersion(packedClass);
    ;

    switch (version)
    {
        case #CurrentVersion:
        {
            [version, #CurrentList] = packedClass;
            break;
        }
        case #Version1:
        {
            [version, #CurrentListV1] = packedClass;
            break;
        }
        default:
            return false;
    }

    return true;
}
```

Listing A.4—*LedgerPostBatch\unpack*

```
public static LedgerJournalPostBatch create(container _packedObject)
{
    LedgerJournalPostBatch newLedgerJournalPostBatch = LedgerJournalPostBatch::construct('', false);
    ;

    newLedgerJournalPostBatch.unpack(_packedObject);

    return newLedgerJournalPostBatch;
}
```

Listing A.5—*LedgerJournalPostBatch\create*

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

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

Worldwide +1-701-281-6500

www.microsoft.com/dynamics