merative™

Cúram 8.1.2

Batch Streaming Developers Guide

Note

Before using this information and the product it supports, read the information in

Edition

This edition applies to Cúram 8.1, 8.1.1, and 8.1.2.

© Merative US L.P. 2012, 2024

Merative and the Merative Logo are trademarks of Merative US L.P. in the United States and other countries.

Contents

1 Developing streamed batch programs

Use this information to design and develop streamed batch programs. Processing load is divided into streams of independent processing, for processing on separate computers as required. A chunker process identifies the processing units and clusters them into chunks of a predefined size. The stream processes then process one chunk at a time.

1.1 Overview

The purpose of this guide is to describe how to design and develop streamed batch programs using the infrastructure provided by the application.

A streamed batch program is one where the processing load is divided into streams of independent processing, where these streams can be processed on separate machines as required. At a technical level the work of the batch program is divided into processing units which can be processed independently of each other, for example the payments for a person or the reassessment of a case.

At execution time, there is "chunker" process which identifies these processing units and clusters these into chunks of a predefined size. The stream processes then process one chunk at a time, once all the chunks have been processed the chunker summarizes the processing carried out by the streams and reports on the batch processing carried out.

This guide is intended for any reader who wants to develop a streamed batch program.

Prerequisites

The *Cúram Batch Performance Mechanisms Guide* gives a good background on the general mechanisms for managing the performance of batch programs including streaming.

1.2 Why Develop Streamed Batch Programs

The decision to develop a streamed batch program can be triggered by a known requirement or project imperative. However, there are a number of reasons why it should be considered in the absence of such a driver, these are set out in this section.

Built-in features

The batch streaming infrastructure implements a number of features which are typically required for batch programs. Using the batch streaming infrastructure means that these features do not need to be directly implemented in your batch program.

The following are the main features of the batch streaming infrastructure:

- **Commit point processing**
  The transaction is committed after each chunk is processed. This stops the transaction size getting too large which can cause performance and/or database locking issues.
- **Skipping processing for records with errors**
  Any record which causes an error when being processed is automatically skipped. The transaction for the chunk is rolled back, the record in error is added to a skip list and the processing of the chunk restarts.
- **Generic batch logging/reporting**
  Logging of progress of the batch process should assist in locating any issues that might be found. The generic elements of the reporting of the work done by the batch program - chunks processed, any chunks unprocessed and any records skipped - are built-in. Additional reporting can be added, see .
- **Re-startability**
  In the situation where the chunker and/or stream processes crash or are killed. They can be restarted and processing will continue from the point where processing was stopped. This typically means that any chunks being processed at the time the stream(s) exited will be re-started, however there is potential under limited circumstances that one or more chunks could remain unprocessed in the event of a restart.

Batch Volumes

Typically the decisive reason to stream a batch program is because the anticipated volume of records cannot be processed inside the available time using a single thread of execution. However, it is not easy to accurately predict the actual volumes that may be encountered in a production system, so the best option (where possible) is to use streaming for all batch programs - even if the "default" run configuration just uses the chunker with an in-process stream.

Some styles of processing can never be streamed and must be run as a serial process inside a single transaction. However, even where this is seemingly the case it is worth closely examining the options to see if the use of parameters to batch programs or developing multiple batch programs will allow the required elements to be run serially while still running in parallel within those serial elements. For example a batch program is required to reassess every active case in the system. All cases of type A should be reassessed before cases of type B. Case type is used as a parameter which allows the batch program to be streamed. This is achieved by running the batch program with case type A as a parameter and then again with case type B as a parameter.

1.3 Designing Streamed Batch Programs

Identify the Processing Unit

Identifying the processing unit is the key to the design of streamed batch programs. The aim here is to identify the smallest possible unit of work that can be executed without risk of overlap between processing units. While there may be a readily identifiable grouping (by case or participant), more work will be needed where some serial processing may be required.

It is important to note that this sub-division of the processing is a fundamental activity where the work of a batch program needs to be done in parallel and this is not merely a feature of the batch streaming infrastructure, but of the problem being addressed.

Understanding Processing Unit Dependencies

It is also important to understand the different sorts of dependencies than can exist between potential processing units. There are serial dependencies where dependent processing needs to be carried out in set order (before or after) the related processing. This serial dependency may form a single logical transaction of work - where both sets of processing should either be completed or not. There are "fan out" dependencies where during the processing of one element a set of other elements requiring processing may be identified.

Such fan outs of processing either need to be handled in the context of the processing that identified them, which may cause issues with the overlaps between streams and/or with the volume of work in processing a single record (with the fan out) becoming excessive; or the additional processing required needs to noted to be processed at a later stage (this could be implemented using Multiple Batch Programs, see Multiple Batch Programs on page 12.

There are many approaches to this sub-division of the processing, but some of the more common are listed below:

- **Indirect units**
  An indirect unit is where the unit of processing is not made up of the artifacts being processed directly, but rather using a different (related) grouping. For example when generating payments, which processes Instruction Line Items (ILIs) the processing unit is a participant - this is necessary because of the business requirement to issue all payments (ILIs) due to a single participant in a single payment instrument, grouping by ILI would lead to overlaps as the roll-up of a participant payment pulled in other ILIs.
- **Composite keys**
  Composite keys can be used where there is no "natural" key in the data which uniquely identifies the unit of processing, for example where the combination of a participant and case needs to be used. However, because the database design of the application makes it unlikely that any composition wouldn't have a unique ID in it's own right (Case Participant Role in the example above), this is not directly supported by the batch streaming infrastructure, but some further information is available in Using Composite Keys to Identify Processing Units on page 21.
- **Sub-division of processing space**
  Sub-division of processing space is useful where rather than dividing the entire processing space into a single logical set of processing units, it is necessary to first divide the processing space into large sub-units and then produce a set of processing units within each of these. For example where there are a number of types of financial transactions to be processed, say payments, bills and account transfers, it may not be possible to divide all of the financial transactions as a group into suitable processing units. By first sub-dividing by transaction type, each sub-division can then be broken down into processing units. This is not directly supported by the batch streaming infrastructure, but can be achieved using Multiple Batch Programs, see Multiple Batch Programs on page 12.

Multiple Batch Programs

Where a sub-division of processing space or large amounts of extra processing are required, it may be beneficial to divide the processing between multiple batch programs rather than trying to complete all the processing in a single batch program.

Refer to Identifying any Required Extra Processing on page 13. In particular the development of multiple batch programs may significantly simplify the design and development effort for each program as well as allowing the processing and choice of processing unit be optimized for each distinct task. There are two distinct options for developing multiple batch programs:

- **Independent Batch Programs**
  The advantage of this approach is that given each process has a distinct chunker and stream implementation more specialized approaches can be taken to improve performance etc. It's also important to note that, given suitable hardware and scheduling software, independent batch programs can be run at the same time - this can offer substantially better utilization of the batch window provided the processing for each program is independent. Alternatively, it is possible to create a single batch program that runs several distinct chunker processes in sequence without creating multiple batch programs, see Running Multiple Chunker Processes in a Single Instance on page 20.
- **Single Batch Program using parameters**
  This is not directly supported by the batch streaming infrastructure, but can be achieved by adding a (mandatory) parameter to the chunker identifying the subset of data to be processed and scheduling a run of the chunker for each sub-division. It's important to note that, barring a specialized implementation (see Running Multiple Instances of the Same Batch Program on page 20) multiple instances of a single batch program can not be run at the same time - the batch streaming infrastructure uses a program key to allow the chunker to be restarted in the event of a crash. Two instances cannot be run concurrently with the same key.

Providing Meaningful Information using the Batch Summary

By default the batch streaming infrastructure reports on the number of chunks skipped as well the total execution time. As the report is generic, it can only report in terms of chunks and the number of records in each chunk. By including summary information when implementing a batch program, additional information on the number of business artifacts processed, including categorization can be included.

This information is included in an email/report aimed at the operator running the batch programs. The batch summary report should convey only the key details of the processing. In general the report should contain no more than ten categorizations. For example, included below is the output of the GenerateInstruments batch program - the seven counts in the center are produced as summary information.

```
Report from GenerateInstruments batch job run on 2011-09-06 at
10:44:53.


Total number of Instruction Line Item record(s) processed: 6

Total number of additional Surcharge Instruction Line Item
records(s) created: 0

Total number of Interest record(s) created: 0
```

```
Total number of Payment Instruction record(s) created: 1

Total number of Payment Instrument record(s) created: 1

Total number of Payment Received Instruction record(s) created: 0

Total number of Liability Instrument record(s) created: 0


Job started at 10:44:44 and took 00:00:09 to complete.
```

Figure 1: GenerateInstruments batch program report

Identifying any Required Extra Processing

Extra processing that is required is processed by the chunker after the chunking has been completed. This allows processing to start on other streams. However the extra processing happens before the chunker starts its' own stream (if configured to do so), in a single database transaction. It is important therefore to take the total volume of processing expected when designing any extra processing into account.

In general if the five year maximum volume of data is likely to take more than a couple of minutes to process, then a separate batch program (see) is a better option than using extra processing. Where extra processing is typically used is where there is some additional processing cannot be associated with any one chunk.

1.4 Implementing Streamed Batch Programs

The implementation of a streamed batch program may seem complex at first glance, however if the individual elements of the implementation are considered separately it can be broken down into more manageable tasks. This section should guide you through these tasks.

Before you start

Before starting into the detail, there is one key note in relation to transaction management.

> **warning Use of Transaction Management calls is not supported**
>
> Because the Batch Streaming infrastructure uses the database as an effective communication mechanism between the stream(s) and the chunker it needs to retain control over the database transactions. To this end no transaction management calls should be made within any of the code involved within the streamed batch program. In particular none of the following methods should be called:
>
> - `curam.util.transaction.TransactionInfo.begin()`
> - `curam.util.transaction.TransactionInfo.commit()`
> - `curam.util.transaction.TransactionInfo.rollback()`

Modeling and Class structure

For the correct classes to be generated, along with the supporting meta-data for the batch launcher, a class must be defined in the UML model with one method of stereotype <<batch>> for each batch executable.

For more information on the <<batch>> stereotype, please consult the *Cúram Modeling Reference Guide*. When writing a streamed batch program two batch executables are required for the chunker and the stream. For example, consider two classes each with a method called process.

In other to use the batch streaming infrastructure the following is also required:

1. a chunker implementation which implements the BatchMain interface.
2. a stream implementation which implements the BatchStream interface

To minimize the volume of classes required, it is recommended to add the methods from the interfaces, which are required for the implementation, to the modeled classes created above. This gives rise to two modeled classes as shown in the figure below.
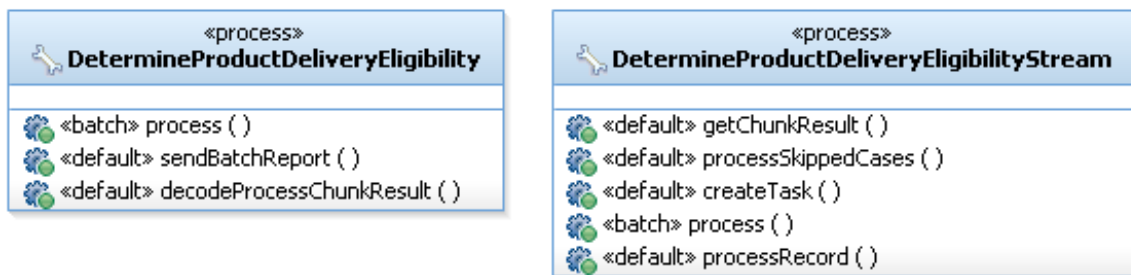

Figure 2: UML Model for Example Streamed Batch Program

The factory mechanism used for generated Cúram classes prevents other classes seeing the interfaces implemented by the impl classes. To get around this, it is necessary to create a wrapper class to implement the required interfaces. This is shown in the examples below.

```
public class DetermineProductDeliveryEligibilityWrapper
  implements BatchMain {

  private curam.core.intf.DetermineProductDeliveryEligibility
    determineProdDeliveryEligibilityObj;

  public DetermineProductDeliveryEligibilityWrapper(
    curam.core.intf.DetermineProductDeliveryEligibility
    determineProductDeliveryEligibility) {

    determineProdDeliveryEligibilityObj =
      determineProductDeliveryEligibility;

  }

  public void sendBatchReport(
    String instanceID, BatchProcessDtls batchProcessDtls,
    BatchProcessChunkDtlsList processedBatchProcessChunkDtlsList,
```

```
    BatchProcessChunkDtlsList
 unprocessedBatchProcessChunkDtlsList)
    throws AppException, InformationalException {


 determineProdDeliveryEligibilityObj.sendBatchReport(instanceID,
      batchProcessDtls, processedBatchProcessChunkDtlsList,
      unprocessedBatchProcessChunkDtlsList);

  }

  public BatchProcessingResult doExtraProcessing(
    BatchProcessStreamKey batchProcessStreamKey,
    Blob batchProcessParameters)
    throws AppException, InformationalException {

    return null;

  }

}
```

Figure 3: Chunker Wrapper implementation

In this particular example the doExtraProcessing operation isn't implemented and so the wrapper just returns and the modeled class doesn't contain this method.

```
public class DetermineProductDeliveryEligibilityStreamWrapper
  implements BatchStream {

  private
 curam.core.intf.DetermineProductDeliveryEligibilityStream
    determineProdDeliveryEligibilityStreamObj;

  public DetermineProductDeliveryEligibilityStreamWrapper(
    curam.core.intf.DetermineProductDeliveryEligibilityStream
    determineProdDeliveryEligibilityStream) {

    determineProdDeliveryEligibilityStreamObj =
      determineProdDeliveryEligibilityStream;

  }

  public String getChunkResult(int skippedCasesCount)
    throws AppException, InformationalException {

    return
 determineProdDeliveryEligibilityStreamObj.getChunkResult(
      skippedCasesCount);

  }

  public BatchProcessingSkippedRecord processRecord(
    BatchProcessingID batchProcessingID, Object parameters)
    throws AppException, InformationalException {

    return
 determineProdDeliveryEligibilityStreamObj.processRecord(
      batchProcessingID,
```

```
        (DetermineProductDeliveryEligibilityKey) parameters);

  }

  public void processSkippedCases(
     BatchProcessingSkippedRecordList
  batchProcessingSkippedRecordList)
     throws AppException, InformationalException {


  determineProdDeliveryEligibilityStreamObj.processSkippedCases(
        batchProcessingSkippedRecordList);

  }

}
```

Figure 4: Chunker Wrapper implementation

Chunker Entry-point

The `process` method described in the example above needs to complete a few steps in order to initialize and start the chunker.

1. Call `batchStreamHelper.setStartTime()` to start the run timer for the program.
2. Set the `instanceID`, in general this should be based on a hard coded entry in the `Batch Process Name` code table. However, making this value dynamic can allow more than one instance of the batch program to be run concurrently, see Running Multiple Instances of the Same Batch Program on page 20.
3. Extract IDs of processing units, see Extracting Processing Units IDs for the Chunker on page 16 for more details.
4. Set the batch main parameters, see Configuring the Chunker on page 17 for more details.
5. Call `batchStreamHelper.runChunkMain` to start the chunker processing. This method will exit when all the processing for the chunker has completed. Typically this method just returns at this point, allowing the chunker process to exit. However, by restarting from step one above another chunker instance could be run in the same process after the first chunker has completed, see Running Multiple Chunker Processes in a Single Instance on page 20.

Extracting Processing Units IDs for the Chunker

The list of IDs passed into the chunker must be an instance of the struct `BatchProcessingIDList`. In general, it is possible to construct a modeled entity operation that returns an instance of this struct or populate an instance of this struct using the results of one or more entity operations in code.

However, if necessary, complex business logic could be constructed to populate this list with an appropriate set of IDs. But, in this instance, it is important to consider that this processing will all take place in the chunkers thread of execution and it *may* be more efficient to forgo some of the optimization at this point and allow the streams to filter out some instances where no work is required as this effort will be parallelized across all the streams.

While the infrastructure assumes that the IDs passed in this struct are single keys, it is possible to use composite keys if necessary, see Using Composite Keys to Identify Processing Units on page 21.

Configuring the Chunker

There are a set of configuration options passed into the chunker when it starts, that control various parameters of the operation of the streamed batch program.

1. The `ChunkMainParameters.chunkSize` parameter controls the number of records in each chuck. Because this value typically has to be tuned for productive use, so that the transaction time for each chunk remains low, it is typically exposed as an `EnvVar` with a sensible default value.

2. The `ChunkMainParameters.dontRunStream` parameter controls whether or not a stream is run in the chunker process while waiting for the other streams to complete. Because this value typically has to be tuned for productive use, it may be the case that the machine hosting the chunker is required for other processing while the streams run elsewhere, it is typically exposed as an `EnvVar` with a default value to run the stream (`false`).

3. The `ChunkMainParameters.startChunkKey` parameter specifies the key value for the first chunk to be picked up by the streams. Where extra processing has been implemented this value is typically offset by one to allow for the `ChunkResult` used for the extra processing, see Extra Processing on page 17 for further details.

4. The `ChunkMainParameters.unProcessedChunkReadWait` parameter controls the wait time when re-scanning to detected unprocessed chunks once all the chunks have been handed out to streams. Because this value typically has to be tuned for productive use, so that the value is sensible relative to the transaction time for each chunk, it is typically exposed as an `EnvVar` with a sensible default value.

Extra Processing

Extra processing is typically processing which logically belongs in the batch program, but does not fit in the context of any one chunk. As this is executed in a single transaction by the chunker the potential size of any processing needs to be considered carefully.

This is implemented in the `doExtraProcessing` method that gets the `BatchProcessStreamKey` and `batchProcessParameters` as parameters and is expected to return a `BatchProcessingResult` instance containing the encoded results of it's processing, see Encoding Batch Summary Information on page 18.

Setting the Batch Stream Entry Point

This method, called `process` in the example above, needs to set the instanceID, to a value that matches that set by the chunker, then call `batchStreamHelper.runStream` to start the stream processing.

Typically this method just returns at this point, allowing the stream process to exit, however by restarting from the beginning above another stream instance could be run in the same process after the first stream has completed, see Running Multiple Chunker Processes in a Single Instance on page 20.

Processing a Single Record ID

The core processing of the stream takes place in this method, implemented in `processRecord`. This method gets `BatchProcessingID` and `batchProcessParameters` as parameters

and is expected to return a `BatchProcessingResult` instance containing the encoded results of its' processing.

See for further details.

If an unrecoverable technical error is encountered when processing the record passed into this method an exception should be thrown. This will cause the batch streaming infrastructure to add this ID to a skip list, roll back the transaction and re-start the processing of the current chunk. This method will not be called for an record IDs on the skipped list. Examples of unrecoverable technical errors would include database errors or errors writing to third party systems. The key point to consider is the expectation that the error is transient i.e. that there is a reasonable chance that it will not recur the next time the processing is run. It is also possible that the chunker may retry the skipped records once all the streams have completed, for more information, see

Given that only a count of skipped IDs will be reported and that the records will be re-processed on the next run of the batch program, careful consideration needs to be given to any unrecoverable *business* errors detected. It may make more sense to modify the business state of the governing business object, for example suspending the case, and informing a user so they can take corrective action, via a task sent to the case owner. Examples of unrecoverable business errors would include incomplete or invalid data. Note that errors like this will only be resolved by the intervention of a knowledgeable user.

One class of unrecoverable business error requires special consideration: where there is invalid or incomplete configuration data. In this instance we would expect this to effect many (if not all) of the records to be processed. But by acting to modify the business state of the governing business object and individually informing users, the impact on the business might be very bad - imagine 1,000,000 suspended cases and individual tasks for users! In this instance it is better to treat such issues in the same way as unrecoverable technical errors.

Processing Skipped Cases

This method implemented by the stream in `processSkippedCases` is called once per chunk, passing in the list of skipped records, in a parameter of type `BatchProcessingSkippedRecordList`.

This allows any required processing for the skipped records, for example creating notifications, to take place. Because skipped records can be re-processed, either in the chunker or in a separate run of the batch program, it's important that any actions, like the notification, make it clear that subsequent processing *may* have resolved the issue and this should be verified before taking any corrective action.

Encoding Batch Summary Information

The method implemented in `getChunkResult` is called at the end of each processed chunk to encode the results of the processing for this chunk.

The count of skipped chunks is passed into the method. Other values (totals of subcategories of processing, etc) should be accessed as instance variables in the streamer implementation. As this method results in a single `String` it is up to the developer to choose a suitable encoding mechanism to deal with multiple values. A tab delimiter is typically used. The meaning and ordering of the fields along with the encoding used should match the decoding processing

implemented for the chunker - see [Decoding Batch Summary Information on page 19](#) for further information.

Any instance variables used to values used in this method need to be reset at the end of the encoding process to ensure the count for the next chunk isn't inflated by the values for the previous chunk(s).

Decoding Batch Summary Information

This processing needs to be implemented in the chunker, in the example above this is in a separate decodeProcessChunkResult method, which is reused from the Send Report processing. The meaning and ordering of the fields along with the encoding used needs to match the encoding processing implemented for the stream.

Refer to [Encoding Batch Summary Information on page 18](#) for further information.

Sending a Batch Report

Report processing is implemented in the sendBatchReport method of the chunker. This method gets the InstanceID, BatchProcessDtls, ProcessedBatchProcessChunkDtlsList and UnprocessedBatchProcessChunkDtlsList as parameters. It is up to the developer as to what information is included in the report and how it is distributed (saved as a file, emailed, sent as a notification or task in the application, and so on).

However, typically a count of the chunks processed and skipped as well as the totaled summary information is included, along with the total runtime for the batch program. The curam.core.impl.CuramBatch class is typically used for this purpose, assuming a StringBuffer (called emailMessage in the example) has been built up, then the code example below will result in an email being sent and the report saved to the file system.

```
curamBatchObj.emailMessage = emailMessage.toString();

  // constructing the Email Subject based on a message file entry
  curamBatchObj.setEmailSubject(
    curam.message.BPODETERMINEPRODUCTDELIVERYELIGIBILITY
      .INF_DETERMINE_ELIGIBILITY_SUB);

  // set output file identifier -
  // the initial part of the file name, the datetime is added to
 this.
  curamBatchObj.outputFileID =
    curam.message.BPODETERMINEPRODUCTDELIVERYELIGIBILITY
      .INF_DETERMINE_PROD_DEL_ELIG.getMessageText(
        ProgramLocale.getDefaultServerLocale());

  // set the elapsed time
  curamBatchObj.setStartTime(batchProcessDtls.startDateTime);
  curamBatchObj.setEndTime();

  // send email
  curamBatchObj.sendEmail();
```

Figure 5: Sending a Batch Report using CuramBatch

Global Batch StreamConfiguration Options

There are a set of global configuration options which are set via `EnvVars` which effect the behavior of all streamed batch programs.

They are additional to those options referenced in . These are referenced directly by the batch streaming infrastructure and as such do not need to be referenced by any specific implementation. However, as these can affect the behavior of the finished batch process they are documented below:

- **curam.batch.streams.batchprocessreadwaitinterval**
  Sets the interval (in milliseconds) for which a batch stream will wait before retrying when reading the batch process table. The default value for this is 1000.
- **curam.batch.streams.chunkkeyreadwaitinterval**
  Sets the interval (in milliseconds) for which a batch stream will wait before retrying when reading the chunk key table. The default value for this is 1000.
- **curam.batch.streams.scanforunprocessedchunkswaitinterval**
  Sets the interval (in milliseconds) for which the main batch stream (chunker) will wait before trying to scan for unprocessed chunks, once the value in the chunk key table has exceeded the number of chunks. The default value for this is 1000.
- **curam.trace.batchprogress**
  Controls the logging within the batch streaming infrastructure. The default value for this is `false` (logging off).

1.5 Advanced Topics

Running Multiple Chunker Processes in a Single Instance

Given that the `batchStreamHelper.runChunkMain` method returns once the chunker has finished it is possible to write a batch process which kicks off several chunkers in turn. The equivalent for the stream is to do the same after the `batchStreamHelper.runStream` method returns, allowing multiple streams to be started one after the other.

However, extreme care is required to ensure when doing this that each chunker/stream process does not pickup any traces of the previous process(es). To this end it's recommend that the batch program instance be created as a separate class with a distinct class for each chunker/stream instance allowing them to kept totally separate.

Running Multiple Instances of the Same Batch Program

By varying the `instanceID` programmatically, most obviously by setting it based on a parameter passed in, it is possible to run multiple instances of the same chunker implementation at the same. This would also require a similar mechanism to set the `instanceID` of the stream instance(s) as this ID acts as the pairing mechanism between the two.

However, while this is technically possible, care must be taken that this varying of the `instanceID` also varies the set of processing units picked by the chunker. The situation can arise where streams with different instance IDs end up processing the same unit at the same time and contention and/or cross-locking can occur.

Using Composite Keys to Identify Processing Units

While not directly supported it is possible to use a composite key to identify processing units. However, because the batch streaming infrastructure itself encodes all the IDs for a chunk into a string (using a `tab` separator) care must be taken. The encoding used to turn the composite key into a string cannot include `tab` characters or the elements separated by the `tab` are passed separately into the `processRecord` method.

It is also worth noting that the implementation of the `processRecord` method will need to decode the `BatchProcessingID` into the sub-elements of the composite key.

Notices

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the Merative website.

Personal use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of Merative

Commercial use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of Merative.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

Merative reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by Merative, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

MERATIVE MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

Merative or its licensors may have patents or pending patent applications covering subject matter described in this document. The furnishing of this documentation does not grant you any license to these patents.

Information concerning non-Merative products was obtained from the suppliers of those products, their published announcements or other publicly available sources. Merative has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-Merative products. Questions on the capabilities of non-Merative products should be addressed to the suppliers of those products.

Any references in this information to non-Merative websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this Merative product and use of those websites is at your own risk.

Privacy policy

The Merative privacy policy is available at https://www.merative.com/privacy.

Trademarks