Multi-node, partitioned and stateful tasks

MidPoint 3.8 and later

This feature is available only in midPoint 3.8 and later.

Starting with midPoint 3.8, the tasks module provides the following functionality:

1. A task can distribute the work to multiple nodes at once (not only to multiple threads as it was up to midPoint 3.7.x).
2. A task can be resumed at the place where it was suspended (not always from the beginning as it was up to midPoint 3.7.x).

This is implemented using bucket-based work state management along with configurable task partitioning.

In this article we describe the overall picture and the details of work segmentation definition. Workers management and task partitioning are the topics of separate ones.

Bucket-based work state management

Work buckets

The work is divided into *buckets* - abstract chunks of work.

Usually the work consists of iteration over a set of objects: either stored in midPoint repository (e.g. for recomputation task) or stored on a resource (e.g. import, reconciliation or live synchronization). So, the most natural way of segmentation of the work into buckets is by defining a bucket as a set of objects for which a particular item - let us call it *discriminator* - has a value in a given interval. The interval can be numeric, alphanumeric, or of anything comparable (e.g. timestamps). In the future, OIDs can be used for segmentation as well.

There are other possibilities as well. For example, one could segment users according to employee type, organization membership, and so on. Work buckets can be defined using arbitrary search filter(s) over the set of objects.

Basic bucket state

A major distinction for a work bucket is: *is it complete or not*? The work bucket is declared `complete` if there's no work that can be done on it. It does not mean that all the objects were successfully processed, though. Some of them might incur failures; however, this is considered a normal situation and such objects are treated as processed. (Re-processing of such objects can be implemented in the future, if needed.) The other state is `ready` meaning that there is some part of the bucket (maybe all of it) that needs to be processed.

Buckets are kept in the task `workState` data structure. This allows us to track the progress done (at coarse-grained level), restarting the work on last known point if necessary.

Multi-node work distribution

Buckets allow us not only to track the progress, but to easily distribute the work among multiple worker tasks, with the intention of their distribution among cluster nodes.

For such multi-node scenario there is a *coordinator* task and *worker* tasks. Coordinator holds the authoritative list of buckets to be processed. Each worker tries to grab one or more buckets to work on. Such buckets are then copied from the coordinator's `workState` into the worker's one. To know they were allocated their state in coordinator's state is marked as `delegated`. (This is the third possible state besides `ready` and `complete`). After the bucket is processed, it is removed from worker's `workState` and marked in coordinator's `workState` as `complete`.

Minor bucket state a.k.a. bucket progress (future plans)

This is only an idea of a future work.

Sometimes we want to be able to track the progress in more details to avoid needless re-processing objects from the start of the current bucket to the place where the processing stopped. (This might be crucial for situations where the whole processing consists of a single bucket.)

Most typical way how to track in-bucket progress is to:

1. sort processed objects by some *progress-tracking property* (OID, icfs:name, icfs:uid, or basically anything);
2. remember last processed object's progress-tracking property value.

Note that object ordering is not required to manage major bucket state. Nor must the property used for bucket segmentation (discriminator) be the same as minor progress-tracking property - although they will be probably the same for the majority of cases.

Configuring work segmentation into buckets

The following structure is used (embedded in task's `workManagement` item):

Item	Description
taskKind	Kind of task with respect to the work state management: • `standalone`: the task is self-contained. It maintains the list of buckets and process all of them itself. • `coordinator`: the task maintains the list of buckets, but the work is done by its worker subtasks. • `worker`: this is the subtask that performs the work. Besides these values, there is also `partitionedMaster` value that denotes a compound task that is partitioned into subtasks, each carrying out a specific step (stage) of the processing.
buckets	How buckets are created, delegated, completed, how they are translated into objects for processing.
workers	How workers are created and managed. This is applicable only to tasks of `coordinator` kind.
partitions	How subtasks for individual partitions are created and managed. This is applicable only to tasks of `partitionedMaster` kind.

Bucket segmentation definition

An example

The segmentation is defined like this:

An example of segmentation configuration

```
<task ...>
    ...
    <workManagement>
        <buckets>
            <numericSegmentation>
                <discriminator>attributes/ri:uid</discriminator>
                <numberOfBuckets>100</numberOfBuckets>
                <from>0</from>
                <to>100000</to>
             </numericSegmentation>
        </buckets>
    </workManagement>
</task>
```

This is to be read such that that the discriminator (`ri:uid` attribute) is expected to have a numeric value from 0 to 99999 (inclusive) and we want do divide this range into 100 buckets. So the first one will contain values from 0 to 999, second one from 1000 to 1999, then 2000-2999, etc. And the last one (100th) will contain values from 99000 to 99999, inclusive.

Definition options

Current implementation supports the following segmentation definitions:

Segmentation definition	Parameters	Description
all definitions	discriminator	Item whose values will used to segment objects into buckets (if applicable). Usually required.
	matchingRule	Matching rule to be applied when creating filters (if applicable). Optional.
	numberOfBuckets	Number of buckets to be created (if applicable). Optional.
numericSegmentation	from	Start of the processing space (inclusive). If omitted, 0 is assumed.
	to	End of the processing space (exclusive). If not present, both `bucketSize` and `numberOfBuckets` must be defined and the end of processing space is determined as their product. In the future we might implement dynamic determination of this value e.g. by counting objects to be processed.

	bucketSize	Size of one bucket. If not present it is computed as the total processing space divided by number of buckets (i.e. `to` and `numberOfBuckets` must be present).
stringSegmentation	boundaryCharacters	Characters that make up the prefix or interval. Currently, the string segmentation is done by creating all possible boundaries (by combining `boundaryCharacters`) and then using these boundaries either as interval boundaries (if `comparisonMethod` is `interval`) or as prefixes (if `comparisonMethod` is `prefix`). This is a multivalued property: the first value contains characters that occupy the first place in the boundary. The second value contains characters destined for the second place, etc. An example: if `boundaryCharacters` = ("qx", "0123456789", "0123456789", "0123456789") then the following boundaries are generated: q000, q001, q002, ..., q999, x000, x001, ..., x999. This might be suitable e.g. for accounts that start either with "q" or with "x" and then continue with numbers, like q732812. Another example: if `boundaryCharacters` = ("abcdefghijklmnopqrstuvwxyz", "0123456789abcdefghijklmnopqrstuvwxyz") then the following boundaries are generated: a0, a1, a2, ..., a9, aa, ab, ..., az, b0, b1, ..., b9, ba, ..., bz, ..., z0, z1, ..., z9, za, ..., zz. This might be suitable e.g. for alphanumeric account names that always start with alphabetic character. Beware: current implementation requires that the characters are specified in the order that complies with the matching rule used. Otherwise, empty intervals might be generated, like when using "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ" there will be an interval of e.g. "values greater than `z` but lower than `A`" (empty one) or "values greater than `Z`" (covers items covered by earlier intervals of a-b, b-c, ...).
	depth	If a value `N` greater than `1` is specified here, `boundaryCharacters` values are repeated `N` times. This means that if values of V_1, V_2, ..., V_k are specified, the resulting sequence is V_1, V_2, ..., V_k, V_1, V_2, ..., V_k etc, with `N` repetitions - so `N * k` values in total.
	comparisonMethod	Either `interval` (the default), resulting in interval queries like `item >= 'a' and item < 'b'`. Or `prefix`, resulting in prefix queries like item `starts with 'a'`. Beware, when using `prefix` method, all the discriminator values are covered by `boundaryCharacters` you specify. Otherwise some items will not be processed at all.
oidSegmentation	The same as stringSegmentation but providing defaults of `discriminator` = `#` and `boundaryCharacters` = `0-9a-f` (repeated `depth` times, if needed).	
explicitSegmentation	content	Explicit content of work buckets to be used. This is useful e.g. when dealing with filter-based buckets. But any other bucket content (e.g. numeric intervals, string intervals, string prefixes) might be used here as well.

More examples

The `oidSegmentation` is the easiest one to be used when dealing with repository objects. The following creates 16^2 = 256 segments.

Buckets defined on first two characters of the OID

```
<workManagement>
    <buckets>
        <oidSegmentation>
            <depth>2</depth>
        </oidSegmentation>
    </buckets>
</workManagement>
```

The following configuration provides string interval buckets:

- less than a
- greater or equal a, less than b
- greater or equal b, less than c
- ...
- greater or equal y, less than z
- greater or equal z

(comparison is done on normalized form of the `name` attribute)

Buckets defined on the first character of the name

```
<workManagement>
    <buckets>
        <stringSegmentation>
            <discriminator>name</discriminator>
            <matchingRule>polyStringNorm</matchingRule>
            <boundaryCharacters>abcdefghijklmnopqrstuvwxyz</boundaryCharacters>
            <comparisonMethod>interval</comparisonMethod>
        </stringSegmentation>
    </buckets>
</workManagement>
```

The following configuration provides three buckets. The first comprises `identifier` values less than 123. The second comprises values from 123 (inclusive) to 200 (exclusive). And the last one contains values greater than or equal to 200.

Three work buckets defined as numeric intervals

```
<workManagement>
    <buckets>
        <explicitSegmentation>
            <discriminator>attributes/ri:identifier</discriminator>
            <content xsi:type="NumericIntervalWorkBucketContentType">
               <to>123</to>
            </content>
            <content xsi:type="NumericIntervalWorkBucketContentType">
               <from>123</from>
               <to>200</to>
            </content>
            <content xsi:type="NumericIntervalWorkBucketContentType">
               <from>200</from>
            </content>
        </explicitSegmentation>
    </buckets>
</workManagement>
```

The following configuration provides four buckets. The first three correspond to users with `employeeType` of `teacher`, `student` and `administrative`. The last one corresponds to user with no `employeeType` set.

Work buckets defined on employeeType values

```
<workManagement>
    <buckets>
        <explicitSegmentation>
            <content xsi:type="FilterWorkBucketContentType">
                <q:filter>
                    <q:equal>
                        <q:path>employeeType</q:path>
                        <q:value>teacher</q:value>
                    </q:equal>
                </q:filter>
            </content>
            <content xsi:type="FilterWorkBucketContentType">
                <q:filter>
                    <q:equal>
                        <q:path>employeeType</q:path>
                        <q:value>student</q:value>
                    </q:equal>
                </q:filter>
            </content>
            <content xsi:type="FilterWorkBucketContentType">
                <q:filter>
                    <q:equal>
                        <q:path>employeeType</q:path>
                        <q:value>administrative</q:value>
                    </q:equal>
                </q:filter>
            </content>
            <content xsi:type="FilterWorkBucketContentType">
                <q:filter>
                    <q:equal>
                        <q:path>employeeType</q:path>
                    </q:equal>
                </q:filter>
            </content>
        </explicitSegmentation>
    </buckets>
</workManagement>
```