Possible extensions of mapping rule format in Capacity Scheduler

version 0.1

Introduction

Capacity Scheduler supports so-called *mapping rules*. If enabled, a list of mapping rules determine where an application will be placed when submitted to YARN.

Conceptually, this is the same as *placement rules* in Fair Scheduler. Both are responsible for determining a target queue depending on the user, application, the user's primary or secondary group, etc.

During the implementation of YARN-9698 (Tools to help migration from Fair Scheduler to Capacity Scheduler), we realized that migrating placement rules to mapping rules is far from straightforward. In fact, even a simple chain of rules cannot be migrated in Capacity Scheduler because of how it works.

As the need for additional features is growing, so is the complexity of the format which describes these rules.

The purpose of this document

We look into various aspects of how the mapping rules could be possibly described in a certain format.

Since we can already define rules in Capacity Scheduler, we can go to three possible directions:

- extend the current format which relies on a key-value pair
- use a new, proprietary format
- use an already existing, structured data format

The goal of this document is **not** to exactly describe the proposed changes - it contains suggestions and considerations and finally, a recommendation from our part. In short, it demonstrates what a given approach would look like.

The exact changes will be described in the upstream JIRA ticket itself or in a separate document.

Differences between the two schedulers regarding mapping rules

In order to understand what changes we have to make, it's crucial to see the differences in the two schedulers.

Behavior / feature	**Fair Scheduler**	**Capacity Scheduler**
`default` rule	Yes	No
`reject` rule	Yes	No
Rule matches, but the	Falls through to the next rule	Returns "default"

queue does not exist and cannot be created		
Take queue from the application submission context	`specified` rule	No (unless `override-mappings` is enabled, but that disables all mappings)
Rule is unreachable	Exception is thrown during startup	Nothing
Specify an arbitrary default queue	Yes	No
`create` flag for a rule	Yes	No
adjustable queue for `default` rule	Yes	No

The table above outlines what we have to add to the existing format. Note that this design document only deals with the **format**. There are also significant behavioural differences between the two, but that will be addressed in a different document.

Explanation of missing features

`default` rule

This basically returns the default queue, which is usually `root.default`. However, in Fair Scheduler, this can be modified.

Specifying an arbitrary default queue

We might not want to place something into `root.default`. That queue can still exist with a purpose, but we'd like to use `root.users.default` instead.

`reject` rule

This rejects the application submission immediately.

`specified` rule

This takes the target queue from the user. For example, the desired queue of a MapReduce job can be defined by using the `-Dmapreduce.job.queuename` command line argument.

Proceeding to the next rule if the desired queue does not exist

This is the Fair Scheduler way. In Capacity Scheduler, we place the application into `root.default`. However, for backward compatibility, we have to indicate what kind of behaviour we want.

Unreachable rule

It's very easy to construct a series of rules where some of them can never be executed. Consider the rule `u:%user:%user`. This rule matches every user and will always return something. If you have subsequent mapping rules after it, those will never be evaluated. This is currently how Capacity Scheduler works, but Fair Scheduler detects unreachable rules and throws an exception during intialization.

`create` flag on a rule

This is a central setting in Fair Scheduler. It defines whether the target queue should be created or not if it doesn't exist. If `create` is true, the queue is always created hence the rule is considered terminal. For nested rules, `create` is interpreted for both the inner and outer rule. In this case, both flags have to be true in order to have the rule terminal.

Approach #1: Using and extending the current format

The current format, explained

Using formal grammar, it roughly looks like this:

```
mappingRules → mappingRule (“,” mappingRule)+

mappingRule → type “:” selector “:” targetQueue

type → "u" | "g"

ID → [a-z][A-Z][0-9]+

selector → "%user" | ID

targetQueue →   "%primary_group"
                “%primary_group.%user"
                “%secondary_group"
                “%primary_group.%user"
```

```
                    "%secondary_group.%user"
                    “%user”
                    queue ".%primary_group.%user"
                    queue "."%secondary_group.%user"

queue → ID ("." ID)+
```

Less formally, it’s three columns are separated by a colon. The first column (type) determines whether it’s a mapping rule for users or for groups. The selector either matches a specific user or all users. The third column determines the target queue based on various placeholders. Multiple mapping rules are separated by comma.

Not all of these combinations are valid. If we define a group rule (“g”) then it can only apply to a specific user, not to all users (`%user`).

Valid mapping rule examples

`u:%user:%user` - every user will be put into his/her own queue. The queue is referenced by its short name and must exist. If the queue doesn’t exist, then `root.default` will be used.

`u:%user:%primary_group` - every user will be put into a queue which is named after their primary group. The queue is referenced by its short name and must exist. If the queue doesn’t exist, then `root.default` will be used.

`u:%user:root.users.%primary_group.%user` - similar to the above, except that we use a full path for `root.users.%primary_group`. If `root.users` is a managed parent queue, then the leaf denoted by `%user` will be created automatically. Otherwise the queue has to exist. If it doesn’t, then `root.default` will be used.

`u:bob:root.users.bob` - if the submitter is “bob”, then the application will be placed into `root.users.bob`.

`g:users:root.users.jobs` - if the submitter user’s group is “users”, then the application will be placed into `root.users.jobs`.

Invalid mapping rule examples

`g:%user:root.users.jobs` - a user has to be specified in the second column.

`u:alice:%primary_group.%user` - the `%user` token cannot be used in this context (it’s already known that it only matches “alice”).

`g:bob:%secondary_group` - only the `%user` token can be used for group mappings.

A possible extension

In this section, we examine how we could possibly extend the current format. As you will see, mapping rules will become harder to read.
Let's see an example of current format which we'd like to extend:

```
u:%user:root.users.%primary_group.%user
```

This rule reads: for each user, we determine the user's primary group and return a queue string `root.users.[user's primary group].[userName]`. So, for example, user bob belongs to the "admins" group, the resulting queue will be `root.users.admins.bob`.

We'd like to specify some extra settings:

- as in Fair Scheduler, we'd like to set `create = true` for both the inner and outer rule. That is, we'd like Capacity Scheduler to create both queues if necessary (note: this might not be possible to do right now, but let's ignore this limitation for the sake of this example).
- if the queue creation fails for whatever reason (eg. `root.users` is not a managed parent), we'd like to use `root.tmp` as a fallback queue, not `root.default`
- we might not want to use a default queue. Instead we'd like the placement engine to fall through to the next rule.

One possible approach is the following to introduce a `create` flag between brackets and also add a fourth column:

```
u:%user:root.users.%primary_group[create].%user[create]:fallthrough
```

If we want to change `"root.default"` to something else:

```
u:%user:root.users.%primary_group[create].%user[create]:default="root.tmp"
```

Note: using both `fallthrough` and `default` doesn't make sense - if create fails or the queue doesn't exist, we proceed to the next one.

It's obvious that the new settings must be included in the mapping string, one way or another. There are multiple solutions to this, but one can notice that the rule became longer and harder to read.

We also miss a couple of other rules:

- Take the queue from the application submission context. This is called `<specified>` in Fair Scheduler. Examples:
    - `u:%user:%specified` - take the queue from the submitter

    - `u:%user:%specified[create]` - take the queue from the submitter and create it if necessary
    - `u:%user:%specified[create]:default="root.tmp"` - take the queue from the submitter and create it if necessary. If the creation fails, we return `root.tmp` and not `root.default`.
- Reject the placement
    - `u:%user:%reject`
- Use the default queue (or specify it to be something other than `root.default`):
    - `u:%user:%default` - this returns `root.default`
    - `u:%user:%default[create]:default="root.tmp"` - use `root.tmp` as default and try to create it.[1]

Now let's see a chain of rules:

1. Try to use `root.users.%primary_group.%user`, create the queues if necessary. Proceed to the next rule if the creation fails.
2. Take the queue from the submitter. Proceed to the next rule if the queue does not exist.
3. Use the default queue. If there's no default queue for whatever reason, proceed to the next rule.
4. Reject the submission.

This would look like:

```
u:%user:root.users.%primary_group[create].%user[create]:fallthrough
,u:%user:%specified:fallthough,u:%user:%default:fallthrough,u:%user
:%reject
```

Thoughts about approach #1

The current mapping rule format is already a bit clumsy. It was usable when it was introduced to Capacity Scheduler. However, recently we've added more logic to it (primary group, secondary group, nested primary and secondary group with user, etc) and in turn, it has become more complex. Extending it would lead to a property value which is difficult to understand. The parsing logic has to be extended manually as well, including the unit tests.

Our recommendation is that we should deprecate this format and not use it any further.

The implementation behind the rule evaluation is in the `UserGroupMappingPlacementRule` class. Maintaining the code inside this class has

[1] This can be interpreted in two ways. We can try to create `root.default` and return `root.tmp` if it fails or the other way around. The ambiguity comes from the fact that in Fair Scheduler, creating a queue dynamically always succeeds. In Capacity Scheduler, this is a bit more complicated.

been proven difficult and understanding it is a challenge on its own. However, discussing what to do with the code is not the scope of this document.

Approach #2: New proprietary format

Since we already have a lot of settings, using a multi-line, structured format seems to be the most logical solution.

Instead of having a long line of rules, we define each rule separately with the necessary line feeds which helps understand what a given rule does.

It could look like this:

```
rule:
      type= <user or group>
      matches= <specific user, multiple users or everyone>
      queue= <string specification which describes the target queue>
      fallthrough = <boolean - what if no valid queue can be
returned?>
      defaultQueue = <if fallthrough is false, what queue should be
used as default>
```

Let's see what each field means:

- `type`: Defines whether the rule applies to users or groups.
- `matches`: A string, which decides what users or groups should the rule be applied to.
- `queue`: This element tells the placement engine what the target queue is. Since we also have more complicated definitions like `%primary_group.%user` and we want to enable auto-creation of queues, we have a lot of options for this particular field.
- `fallthrough`: If set to "true", then we jump to the next rule if the current rule cannot return a valid queue. This is how Fair Scheduler works. If set to "false", then we return the default queue.
- `defaultQueue`: Used if fallthrough is "false". This overrides `root.default`.

Let's see some examples of how original mapping rules can be expressed in the new format:

Orginal format	New format

`u:%user:%user`	`rule:`
	`  type=user`
	`  matches=*`
	`  queue=%user`
	`  fallthrough=false`

`u:%user:%primary_group.%user`	`rule:`
	`  type=user`
	`  matches=*`
	`  queue={`
	`  parent=%primary_group`
	`  leaf=%user`
	`  }`
	`  fallthrough=false`

`u:alice:root.users.alice`	`rule:`
	`  type=user`
	`  matches=alice`
	`  queue=root.users.alice`
	`  fallthrough=false`

Proposed new rules can be expressed in the following way[2]:

Extended original format	**New format**

`u:%user:root.users.%user[create]`	`rule:`
	`  type=user`
	`  matches=*`
	`  queue={`
	`  parent=root.users`
	`  leaf=%user[create]`
	`  }`
	`  fallthrough=false`

`u:%user:%primary_group[create].%user[create]:defaultQueue=root.tmp`	`rule:`
	`  type=user`
	`  matches=*`
	`  queue={`
	`  parent=%primary_group[create]`
	`  leaf=%user[create]`
	`  }`
	`  fallthrough=false`
	`  defaultQueue=root.tmp`

`u:%user:%specified`	`rule:`
	`  type=user`
	`  matches=*`
	`  queue=%specified`
	`  fallthrough=false`

`u:%user:%reject`	`rule:`
	`  type=user`
	`  matches=*`
	`  queue=%reject`

[2] The rules on the left are not legal mapping rules - just possible extensions that were explained earlier.

Variations for nested rules

There are a variety of ways to express nested rules. Since we already use braces `{ }` and decomposed the queue path to parent and leaf parts, there's no reason to include the create flag in these.

In Fair Scheduler, we have an outer rule and an inner rule. Similarly, we can do the following:

```
rule=nested {
      parent="root.users"
      outer=%primary_group / %secondary_group / %default
      outer_create=true
      inner=%user
      inner_create=true
}
```

This gives us the possibility to define an arbitrary inner rule. But as it turns out, the inner rule is always `%user` in both schedulers. If we don't need this flexbility, then we can simplify

```
rule=nested_user {
      parent="root.users"
      outer=%primary_group / %secondary_group / %default
      outer_create=true
      inner_create=true
}
```

Also, we might even skip `%default` as an option for "outer". If "outer" is undefined but we have a parent, that can be thought as a `%default` outer rule.

We could go on, but the basic idea should be clear by now.

Thoughts on approach #2

Having a clear, well-structured and readable description of the rules is a big step forward. We don't have useless elements and characters that could potentially hinder readability.

However, this solution comes at a cost:

- We have to manually implement the parser. Since the rule is defined over multiple lines, it's more complicated.
- Even worse, nested rules are expressed best with a nested structure. Processing this with an ad hoc parser is prone to errors. We have to track the state of the parser and we need to check every character in the given states to make sure that element is syntactically correct.

- We might generate a parser from a formal grammar. A tool like ANTLR can come in handy. But without adequate experience, it's also full of pitfalls.
- Lot of testing is necessary to ensure that the rule parser works as intended.

Overall, creating our own format is risky. For simple text, it's doable, but in our case, we better off rely on something that already exists.

Approach #3: JSON

Everything that was suggested as part of the "proprietary format" can be expressed in JSON as well.

JSON is a very popular human-readable data interchange format. It is more readable than XML because values are not enclosed between tags, instead, it is more like a nested key-value description format.

There are several reasons why JSON should be the preferred choice. But first, let's see how the previously defined format would look like in JSON:

```
{
"rules": [
      {
            "type": "user",
            "matches": "user3",
            "queue": {
                  "parent": "root.users.%primary_group[create]",
                  "leaf": "%user[create]"
            },
            "fallthrough": true,
            "defaultQueue": "root.tmp"
      },
      {
            "type": "user",
            "matches": "*",
            "queue": {
                  "parent": "root.users",
                  "leaf": "%user[create]"
            },
            "fallthrough": true,
            "defaultQueue": "root.tmp"
      },
      {
            "type": "user",
```

```
            "matches": "*",
            "queue": "%specified",
            "fallthrough": true,
            "defaultQueue": "root.tmp"
      },
      {
            "type": "user",
            "matches": "*",
            "queue": "%default",
            "defaultQueue": "root.something"
      },
      {
            "type": "user",
            "matches": "*",
            "queue": "%reject",
      }
      ]
}
```

As you can see, the structure is pretty much the same. The only difference is that rules are now listed in curly brackets and keys (and most values) are placed in quotation marks. However, this is still an acceptable compromise.

Variations

When it comes to describing more complex mapping rules, we have the same options. We can separate the parent from leaf and use separate fields to describe the “create” setting:

```
"queue": {
      "parent": "root.users”,
      “outer_rule”: “%primary_group",
      “outer_create”: true,
      "leaf": "%user",
      “inner_create”: true
},
```

In fact, the key “queue” might be misleading, because we actually describe a policy, not a queue itself. So using “policy” or “rule” is more sensible.

Another thing to keep in mind is that we don’t necessarily need to use placeholder strings if the rule is sufficiently broken up into pieces. For example, the rules “specified”, “default” and “reject” cannot be used as a token which is prepended or appended to an already existing string. These rules don’t make sense:

- “root.users.%specified” (in theory, this *could* be used, but would be very confusing)
- “root.%reject”
- “root.hive.query.%default”

This also means that we can simply drop the “%” character, there’s no need for it.

With this in mind, we can modify our format:

```
"policy": {
      "parent": "root.users",
      "outer": "primary_group",
      "outer_create": true,
      "inner": "user",
      "inner_create": true
}

"policy": {
      "parent": "root.users",
      "outer": "primary_group",
      "outer_create": true,
      "inner_create": true
}

“policy”: “nested”,
“nestedRule”: {   // describe the rule/policy in a separate object
      "parent": "root.users",
      "outer": "primary_group",
      "outer_create": true,
      "inner_create": true
}

"policy": "specified",
"create": true

"policy": "default",
"defaultQueue": "root.something"
"create": true

"policy": "reject"
```

Thoughts on approach #3

Based on our analysis, JSON seems to be the solution we should aim for. JSON libraries are available in every project. Obviously, there's no need to write a parser, but what's more, we can create schemas and use automatic object mapping which saves a lot of time for us.

Comparison of possible mapping rule formats

Mapping description format	**Pros**	**Cons**
Existing (property key=val)	The main structure already exists. Parser already exists. Probably this affects QM the least.	Parser needs to be extended & tested. Hard to read. All rules are enumerated in a single line, separated by comma. Becomes even harder to read if we intend to extend it.
Proprietary format	Very compact, easy to read.	New parser is necessary. Ad-hoc parser is error-prone, we need a lot of unit tests. Generated parser needs a grammar - lot of pitfalls without adequate experience. Also needs a lot of testing. No automatic object mapping.
JSON	Easier to read than the existing key-value approach. Parser, error handling (no need to code & test anything). Schema, object mapping.	Slightly more verbose than the proprietary format.
XML	Easier to read than the existing key-value approach. Parser, error handling (no need to code & test anything). Schema, object mapping.	Heavier than JSON due to the amount of tags.