Mapping Rule Enhancements

Design doc - V1

Authors: Gergely Pollák, Szilárd Németh, Péter Bacskó, Rudolf Réti

What we want to achieve

Currently the Queue mapping solution in Capacity Scheduler is really hard to maintain and extend, since we want to provide the users a way to migrate from Fair Scheduler to Capacity Scheduler, we need to close the feature gaps. To achieve this we must make CapacityScheduler's mapping rules much more flexible, since we will need to support the feature set of 2 schedulers.

The goal of this proposal is to define how to make Capacity Scheduler mapping rules more flexible, extendable and backward compatible.

Use Cases

- Support migrating Fair Scheduler users
- Add new variables usable in queue names
- Define rejection rules
- Define time base selectors, to allow certain things to run at certain times
- Since the multiple leaf queue change (https://issues.apache.org/jira/browse/YARN-9879) queues should be referenced with their full name wherever it is possible, but for example in the case of rules such as `u:%user:%primary_group.%user` it is impossible. This would be changed implicitly with this proposal

Proposed changes

Merge UserGroup and Application mapping classes

Currently the User/Group and Application mappings are handled by two separate classes (`UserGroupMappingPlacementRule` and `AppNameMappingPlacementRule`), this means these rules cannot be mixed, also these classes have a lot in common. Actually both follow a common pattern:

If the submitted application matches the selector, then place to the defined queue.

The current configuration syntax for user placement is `u:SELECTOR_CONDITION:QUEUE_PATTERN` (eg. `u:bob:root.%user`). In this case the `u:SELECTOR_CONDITION` is the selector, since the "`u`" defines we need to match the user name, and the `SELECTOR_CONDITION` is the actual value we check. For groups we use "`g`" to determine which parameter of the application submission context we examine. In case of application mapping rules we use a different property for declaration, so those only use the `SELECTOR:QUEUE_PATTERN` format, no need to select what we match against.

The `QUEUE_PATTERN` is a queue path, which may contain certain placeholders like `'%user'` or `'%primary_group'`, those will be evaluated and replaced by the respective values. This replacement system is currently somewhat restrictive, but the goal of this document is to extend this feature.

This can be easily generalized by merging the two classes and generalizing the selector and placer classes.The first step in implementing this should be merging these classes. This will speed up development time, since future changes won't have to be implemented in two slightly different classes.

Rules to determine what should happen in case of failure

There are situations where a rule matches a certain application submission, but the defined queue does not exist, for example `%user` mapping, where the parent queue is not a managed parent. Now it's a bit chaotic what happens in such situations in some cases the app gets rejected, other cases it's just placed in the default queue.

The best way to resolve this confusion to let the user specify what should happen if a matcher matches but the resolution is impossible. The following cases should be supported:

- Skip to the next rule
- Reject application
- Place to default queue

The solution should be extendable, if some new use case emerges, it should be easy to add it.

```
/**
 * This method evaluates the rule, and returns the MappingRuleResult, if
 * the rule matches, null otherwise.
 * @param variables The variable context, which contains all the variables
 * @return The rule's result or null if the rule doesn't apply
 */
MappingRuleResult evaluate(VariableContext variables) {
  if (matcher.match(variables)) {
    return action.execute(variables);
  }

  return null;
}

/**
 * Returns the action's fallback
 * @return The fallback of the action
 */
```

```
  MappingRuleResult getFallback() {
    return action.getFallback();
  }
}
```

Generalize Variable Context

Auto queue creation relies on having some replaceable parts of the placement path, like "`%user`" or "`%primary_group`". However this is currently implemented in a very limited way and only on certain code paths.

It would be better if we'd build a variable mapping where we could put all replaceable variables and their values, this map then could be used for replacement throughout the whole placement engine, the same context could be used in matching and queue name building as well.

Most of the variables are predefined, however rules may alter or add new variables eg. the default queue's name.

VariableContext

Variable context is a simple class which is used to store variables and their values as well as methods for replacing variables in strings. This should support queue based replacement, where the string is split along the dots, and only replaces a part if it exactly matches the variable and a generic string replacement, which will replace every variable in the provided string.

```
/**
 * This class is to store the variables (or tokens) which can be used in mapping rules.
 */
interface VariableContext {
  boolean has(String key);
  String get(String key);
  void put(String key, String value);
  void putImmutable(String key, String value);

  /**
   * This method will replace all variable tokens (%VARIABLE_NAME) in the input
   * string with their respective values.
   * @param input The string which might contain variable tokens
   * @return The string in which the tokens are replaced with the respective values of the
tokens.
```

```
   */
  String replaceVariables(String input);

  /**
   * This method will split the path along dots, and all resulting parts
   * which matches a variable exactly, will get replaced by the respective values of the
variables.
   * @param path The path which might contain variable tokens
   * @return The path in which the tokens are replace with the tokens' respective values
   */
  String replacePathParts(String path);
}
```

Variable Suggestions

Here is a list for the suggested additional variables:

- `%user` - The name of the current user
- `%application` - The name of the submitted application
- `%primary_group` - The primary group of the user
- `%secondary_group` - The secondary group of the user
- `%specified_queue` - The queue specified during application submission
- `%default_queue` - The full path of the globally defined default queue

And here are a few more which demonstrate how can this approach easily extended:

- `%day_of_week` - The current day of the week, English name
- `%day_of_week_numeric` - Number 1-7 (Monday … Sunday), matching Java day indexing
- `%day_of_month_numeric` - Number 1-31 matching the current day of the month
- `%month` - Name of the current month in English
- `%month_numeric` - Number 1-12 the index of the month (January...December)

Generalize Mapping Rules

In the current implementation all rule matching is determined by a huge if-else chain in a method which tries to cover all the possible use cases, as the number of use cases increase so does the size of this construct, so we need a new approach to keep it maintainable and be able to add new features.

All mapping rule have two parts:

- Selector: A selector determines if the rule applies to the current application submission. All selectors must have a matcher, which determines what we check against and in most cases a value, which needs to be checked. Eg. `u:bob:root.bob` as we mentioned

earlier `u:bob` is the selector. The ‘u’ part defines the matcher, which will match against the username provided in the submission, the `‘bob’` is the value the matcher matches against.

- Action: Instruction what to do, might affect the flow of placement not only the application placement. E.g. skip the next rule, or change default for application. Actions also should provide a failure handler, which will be invoked if it cannot be executed.

With the generalization of the mapping rules it makes possible to implement different selectors and actions, which makes adding new features much easier and modular.

Matcher Interface

The Matcher interface will get an `ApplicationSubmissionContext` and a `VariableContext` and based on this information it should return whether it matches the current application. The parameters required by the matcher is determined by its implementation and the parser, which creates the Matcher is responsible for providing those.

The interface is also allowed to add extra variables to the variable context based on its matching results. Only matching matchers may alter the variable context.

```
interface RuleMatcher {
  /**
   * Returns true if the matcher matches the current context.
   * @param variables The variable context, which contains all the variables
   * @return true if this matcher matches to the provided variable set
   */
  boolean match(VariableContext variables);
}
```

Initial Matcher Suggestions

Here is a list for the suggested matchers we should start with:

- Generic variable matcher, matches if the two provided values match. One should be a variable at least. Eg. `(new GenericMatcher(“%user”, “Tohotom”))`
  - UserMatcher, specialized `GenericMatcher` with forced `“%user”` matching
  - PrimaryGroupMatcher, specialized `GenericMatcher` with forced `“%primary_group”` matching
  - SecondaryGroupMatcher, specialized `GenericMatcher` with forced `“%secondary_group”` matching
  - ApplicationNameMatcher, specialized `GenericMatcher` with forced `“%app_name”` matching
- `MatchAll` matcher, matches everything.

Action interface

The Action interface is responsible for determining what should happen if an application matches the rule. Possible outcomes for actions are:

- Place the application to a queue (this should handle default, user default etc.)
- Reject application
- Add a variable to the variable context. (This can be used to make dynamic placement targets for later rules, or even tagging)
- Failure, given the current parameters the action cannot be executed, this will automatically invoke the failure handler settings of the action.

The action interface also can determine what should happen if the action cannot be executed the possible remedies:

- Skip to the next rule
- Reject application
- Place to default queue

```
/**
 * Base class for actions, fallback logic handling is implemented here
 */
abstract class RuleAction {
  //The default fallback method is reject, so if the action fails (eg. not a valid queue returned)
  //We will reject the application. However this behaviour can be overridden on a per rule basis
  MappingRuleResult fallback = MappingRuleResult.createRejectResult();

MappingRuleResult getFallback() {
    return fallback;
  }

  /**
   * Sets the fallback method to reject, if the action cannot be executed the
   * application will get rejected
   */
  void setFallbackReject() {
    fallback = MappingRuleResult.createRejectResult();
  }

  /**
   * Sets the fallback method to skip, if the action cannot be executed
   * We move onto the next rule, ignoring this one
```

```
   */
  void setFallbackSkip() {
    fallback = MappingRuleResult.createSkipResult();
  }

  /**
   * Sets the fallback method to place to default, if the action cannot be executed
   * The application will be placed into the default queue, if the default queue
   * does not exist the application will get rejected
   */
  void setFallbackDefaultPlacement() {
    fallback = MappingRuleResult.createDefaultPlacementResult();
  }

  /**
   * This method is the main logic of the action, it shall determine based on the
   * mapping context, what should be the action's result.
   * @param variables The variable context, which contains all the variables
   * @return The result of the action
   */
  abstract MappingRuleResult execute(VariableContext variables);
}
```

Action Suggestions

Here is a list for the suggested actions we should start with:

- `PlaceToQueue` - Very basic action which will place the current application to the queue provided, if the queue name contains variables those will be replaced.
- `RejectAction` - This action will simply reject the application
- `UpdateVariable` - This action will set the provided custom variable to the provided value. E.g. `UpdateVariable("%currentDefault", "temporaryDefault")`

Examples

Changing the default queue for applications submitted by user 'bob', bob should be able to specify his queues

Matcher	Action	Fail	Description
AnyMatcher()	UpdateVariable("%current_default%", "%default_queue%")	Reject application	We initialize the %current_default% variable with the

			value of the default queue
UserMatcher(“bob”)	UpdateVariable(“%current_default”, “root.default_bob”)	Skip to next rule	If the user name matches bob we set the custom variable `%current_default` to `root.default_bob`.
ApplicationMatcher(“DailyProcession”)	PlaceToQueue(“root.daily”)	Reject application	The application should be placed toe the `root.daily` queue, if it does not exists we reject the application
UserMatcher(“bob”)	PlaceToQueue(“%specified”)	Skip to next rule	If bob have specified a queue name, we place the application to that queue, if he did not specify a queue this rule will fail, and move to the next one
AnyMatcher()	PlaceToQueue(“%current_default”)	Reject application	Place any application to the %current_default queue, this means if bob is the current user it has been overwritten to “`root.default_bob`” otherwise it will be placed into the main default queue.

Make/keep placement rules independent of configuration syntax

This proposal does not include how we can create a declarative configuration option for setting up these much more complex placement rules, it is left to the parser, the components intentionally ignore any configuration syntax, to make sure the configuration parsing and rule

object creation are independent. This way the configuration can be improved without changing the mapping rules.

Current rules to new engine examples

Here we declare how the current mapping rule implementation can be transformed into the new solution. Since this document is not about a new syntax, we simply list what objects should be created for the listed mapping rules. This should be done by the parser.

Current user/group rule	**New parser creates rule with**			
	Matcher	**Matcher argument**	**Action**	**Action arguments**
u:bob:root.bob	UserMatcher	bob	PlaceToQueue	root.bob
u:frank:root.users.%user	UserMatcher	frank	PlaceToQueue	root.users.%user
g:developers:root.users.%user	PrimaryGroupMatcher	developers	PlaceToQueue	root.users.%user
u:%user:root.default	MatchAll		PlaceToQueue	root.default
u:%user:%secondary_group.%user	MatchAll		PlaceToQueue	%secondary_group.%user

Current application rule	New parser creates rules with			
	Matcher	**Matcher argument**	**Action**	**Action arguments**
procession:root.big	ApplicationNameMatcher	procession	PlaceToQueue	root.big
procession:%application	ApplicationNameMatcher	procession	PlaceToQueue	%application
%application:%application	MatchAll		PlaceToQueue	%application
%application:root.catchall	MatchAll		PlaceToQueue	root.catchall

Example interfaces and basic usage

```
/**
 * This class is to store the variables (or tokens) which can be used in mapping rules.
 */
interface VariableContext {
  boolean has(String key);
  String get(String key);
  void put(String key, String value);
  void putImmutable(String key, String value);

  /**
   * This method will replace all variable tokens (%VARIABLE_NAME) in the input
   * string with their respective values.
   * @param input The string which might containe variable tokens
   * @return The string in which the tokens are replace with the tokens' respective values
   */
  String replaceVariables(String input);

  /**
   * This method will split the path along dots, and all resulting parts
   * which matches a variable exactly, will get replaced by the variables'
   * respective values.
   * @param path The path which might contain variable tokens
   * @return The path in which the tokens are replace with the tokens' respective values
   */
  String replacePathParts(String path);
}

/**
 * This class stores the condition and action of a mapping rule, if the matcher
 * matches the current application the action will be executed
 */
class MappingRule {
  private RuleMatcher matcher;
  private RuleAction  action;

  MappingRule(RuleMatcher matcher, RuleAction  action) {
    this.matcher = matcher;
    this.action = action;
  }
```

```
  /**
   * This method evaluates the rule, and returns the MappingRuleResult, if
   * the rule matches, null otherwise.
   * @param variables The variable context, which contains all the variables
   * @return The rule's result or null if the rule doesn't apply
   */
  MappingRuleResult evaluate(VariableContext variables) {
    if (matcher.match(variables)) {
      return action.execute(variables);
    }

    return null;
  }

  /**
   * Returns the action's fallback
   * @return The fallback of the action
   */
  MappingRuleResult getFallback() {
    return action.getFallback();
  }
}

interface RuleMatcher {
  /**
   * Returns true if the matcher matches the current context.
   * @param variables The variable context, which contains all the variables
   * @return true if this matcher matches to the provided variable set
   */
  boolean match(VariableContext variables);
}

/**
 * This class is a generic matcher, which during the match method call will replace all variables
 * in the left and right hand expressions, and checks if those are equal.
 */
class RuleMatcherEquals implements RuleMatcher {
  private String left;
  private String right;

  RuleMatcherEquals(String left, String right) {
    this.left = left;
```

```
    this.right = right;
  }

  @Override
  public boolean match(VariableContext variables) {
    //Replacing the variables in the left and righ hand side expression then
    //comparing them
    return variables.replaceVariables(left).equals(variables.replaceVariables(right));
  }
}

/**
 * This class is a specialized Equal matcher, which checks for user name equality
 * If the user name of the current submission matches the provided user name
 * it will match.
 */
class RuleMatcherUser extends RuleMatcherEquals {
  RuleMatcherUser(String userExpression) {
    super(userExpression, "%user");
  }
}

/**
 * This class is a specialized Equal matcher, which checks for application name equality
 * If the application name of the current submission matches the provided name
 * it will match.
 */
class RuleMatcherApplication extends RuleMatcherEquals {
  RuleMatcherApplication(String userExpression) {
    super(userExpression, "%application");
  }
}

/**
 * Base class for actions, fallback logic handling is implemented here
 */
abstract class RuleAction {
  //The default fallback method is reject, so if the action fails (eg. not a valid queue returned)
  //We will reject the application. However this behaviour can be overridden on a per rule basis
  MappingRuleResult fallback = MappingRuleResult.createRejectResult();
  MappingRuleResult getFallback() {
    return fallback;
  }
```

```
  /**
   * Sets the fallback method to reject, if the action cannot be executed the
   * application will get rejected
   */
  void setFallbackReject() {
    fallback = MappingRuleResult.createRejectResult();
  }

  /**
   * Sets the fallback method to skip, if the action cannot be executed
   * We move onto the next rule, ignoring this one
   */
  void setFallbackSkip() {
    fallback = MappingRuleResult.createSkipResult();
  }

  /**
   * Sets the fallback method to place to default, if the action cannot be executed
   * The application will be placed into the default queue, if the default queue
   * does not exist the application will get rejected
   */
  void setFallbackDefaultPlacement() {
    fallback = MappingRuleResult.createDefaultPlacementResult();
  }

  /**
   * This method is the main logic of the action, it shall determine based on the
   * mapping context, what should be the action's result.
   * @param variables The variable context, which contains all the variables
   * @return The result of the action
   */
  abstract MappingRuleResult execute(VariableContext variables);
}

/**
 * Example action which will place an application in a queue
 */
class RuleActionPlaceToQueue extends RuleAction {
  //Name of the queue in which we are going to place the applcation
  String queue;
  RuleActionPlaceToQueue(String queue) {
    this.queue = queue;
```

```
  }

  /**
   * The execute method will generate a placement mapping rule result, which
   * will contain the name of the queue provided at construction time. All the
   * variables will get replaced in the queue name, if any.
   * @param variables The variable context, which contains all the variables
   * @return A PlacementResult
   */
  @Override
  MappingRuleResult execute(VariableContext variables) {
    //We replace all the variables in the queue name before creating the mapping result
    //this will allow us to handle placement rules like u:bob:%user
    return MappingRuleResult.createPlacementResult(variables.replacePathParts(queue));
  }
}

/**
 * Example action which will reject the application
 */
class RuleActionReject extends RuleAction {
  @Override
  MappingRuleResult execute(VariableContext variables) {
    return MappingRuleResult.createRejectResult();
  }
}

/**
 * Example action which will update a variable in the variable context
 * the most trivial way to use it is to update the %default queue name
 */
class RuleActionUpdateVariable extends RuleAction {
  private String key;
  private String value;

  RuleActionUpdateVariable(String key, String value) {
    this.key = key;
    this.value = value;
  }

  @Override
  MappingRuleResult execute(VariableContext variables) {
    //On execute we simply update the variable context with the provided value
```

```
    //The value may contain variables which will get replaced with their repsective values
    variables.put(key, variables.replaceVariables(value));
    //The result is always a skip in this case, since we did not make any placement
    //decisions, the actual variable will get used by a later rule, so we move onto the next rule
    return MappingRuleResult.createSkipResult();
  }
}


enum MappingRuleResultType {
  //Represents a result where we simply ignore the current rule and move onto the next one
  SKIP,
  //Represents a result where the application gets rejected
  REJECT,
  //Represents a result where the application gets placed into a queue
  PLACE
}

/**
 * This class represents the outcome of an action
 */
class MappingRuleResult {
  //The name of the queue we should place our application into
  //Only valid if result == PLACE
  String queue;

 //The result of the action
  MappingRuleResultType result;

  //To the reject result has no variable field, so we don't have to create a new instance all the
time
  //this is THE instance which will be used to represent REJECT
  private static final MappingRuleResult RESULT_REJECT
      = new MappingRuleResult(null, MappingRuleResultType.REJECT);

//To the skip result has no variable field, so we don't have to create a new instance all the time
  //this is THE instance which will be used to represent SKIP
  private static final MappingRuleResult RESULT_SKIP
      = new MappingRuleResult(null, MappingRuleResultType.SKIP);

 //To the default placement result has no variable field, so we don't have to create a new
instance all the time
  //this is THE instance which will be used to represent default placement
```

```
private static final MappingRuleResult RESULT_DEFAULT_PLACEMENT
    = new MappingRuleResult("%default", MappingRuleResultType.PLACE);

/**
 * Constructor is private to force the user to use the predefined generator methods
 * to create new instances in order to avoid inconsistent states.
 * @param queue Name of the queue in which the application is supposed to be placed,
 *              only valid if result == PLACE, otherwise it should be null
 * @param result The type of the result
 */
private MappingRuleResult(String queue, MappingRuleResultType result) {
  this.queue = queue;
  this.result = result;
}

public String getQueue() {
  return queue;
}

MappingRuleResultType getResult() {
  return result;
}

/**
 * Generator method for place results.
 * @param queue The name of the queue in which we shall place the applicaiton
 * @return The generated MappingRuleResult
 */
static MappingRuleResult createPlacementResult(String queue) {
  return new MappingRuleResult(queue, MappingRuleResultType.PLACE);
}

/**
 * Generator method for reject results.
 * @return The generated MappingRuleResult
 */
static MappingRuleResult createRejectResult() {
  return RESULT_REJECT;
}

/**
 * Generator method for skip results.
 * @return The generated MappingRuleResult
```

```
   */
  static MappingRuleResult createSkipResult() {
    return RESULT_SKIP;
  }

  /**
   * Generator method for default placement results. It is a specialized placement
   * result which will only use the "%default" as a queue name
   * @return The generated MappingRuleResult
   */
  static MappingRuleResult createDefaultPlacementResult() {
    return RESULT_DEFAULT_PLACEMENT;
  }
}

/**
 * Dummy class to represent the actual mapping rule class and to demonstrate
 * how the previously defined classes are being used.
 */
class MappingRulesEvaluator {
  List<MappingRule> rules;

String getPlacementForApp(ApplicationSubmissionContext asc, String user) {
    VariableContext mappingCtx = new MappingContextImpl();

    mappingCtx.put("%user", user);
    mappingCtx.put("%default", "root.default");
    mappingCtx.put("%application", asc.getApplicationName());
    mappingCtx.put("%primary_group", groups.getGroups(user).get(0));
    //...etc filling up the variable context

    for (MappingRule rule : rules) {
      //iterating through all the rules and evaluating them
      MappingRuleResult result = rule.evaluate(mappingCtx);

      //if a rule matches to the current context we process it
      if (result != null) {
        try {
          //if the result is a placement result, we try to place it
          if (result.getResult() == MappingRuleResultType.PLACE) {
            String queue = result.getQueue();
            //checking if the resulting queue is valid
            if (validateQueue(queue)) {
```

```
            //if it is valid we have a placement, returning it
            return queue;
          } else {
            //if the queue is not valid we go to the fail branch, since the action failed
            throw new Exception("Invalid target queue");
          }
        }
      } catch (Exception e) {
        //on error we use the fallback action
        result = rule.getFallback();
      }

      //processing actions (fallback and non-placement)
      switch (result.getResult()) {
        case PLACE:
          //This should only be a default placement, since other placement actions
          //are caught earlier
          String queue = result.getQueue();
          if (validateQueue(queue)) {
            return queue;
          } else {
            //if the default queue isn't valid we reject the application
            return null;
          }
        break;
        case REJECT:
          //returning null is the way to reject
          return null;
        break;
        //SKIP is handled automatically since it means to move to the next rule
        //but since we are in a loop, it will happen if we don't do anything
      }
    }
  }

  //if no placement was determined we reject the application
  return null;
 }
}
```