Rule Caches

PegaRULES Process Commander v 4.2

This document is the property of:
Pegasystems Inc.
101 Main Street
Cambridge, MA 02142-1590

Phone: (617) 374-9600
Fax: (617) 374-9620
www.pega.com

PegaRULES Process Commander
Document: Rule Caches
Software Version 4.2 SP6
Updated: May 24, 2006

Contents

Rule Caches in the Process Commander Application

Overview

Pegasystems' patented Rules technology is the foundation of Process Commander BPM solutions, providing flexibility and agility. Rule resolution is the process used to identify which rules are applied to a specific business decision. Through dynamic rule selection, Process Commander provides flexibility that is otherwise impossible (except with the alternative of applying rules specified in code).

In order to allow this powerful Rule Resolution process to occur with the required performance levels for enterprise-scale systems, PegaRULES has a series of caches. A *cache* contains data that has been copied out of one place and stored temporarily in another for easier access. In Process Commander, some caches are stored in the system memory (JVM), some are stored in the database, and some are stored on disk.

Caches stored in memory include:

- Rule cache
- Rules Assembly cache
- "Conclusion" cache

Caches stored on disk in the file system include:

- LookupList cache
- Static Content cache

System administrators can "tune" each of these caches (by using settings in the pegarules.xml file), to set each cache to the most efficient size for their particular application. In addition, it is *possible* to clear the caches, using buttons on the Monitor Servlet. However, **clearing the caches should be done rarely – if ever – and only in certain specific circumstances. Clearing caches has a noticeable negative effect on the performance of the whole system.**

Performance "Tuning" for Caches

There are two major areas involved in making sure a customer's PegaRULES Process Commander application will run at the highest possible level of performance: *design* and *tuning.*

First, the Process Commander application must be **designed** to be as efficient as possible, using all of the powerful features provided in our product. (For example, customers should create Declarative rules for certain of their calculations, rather than having to write complex activities to accomplish these computations.) For the best design guidelines, developers should follow the stated Guardrails and the methodology outlined in *Designing Your Application with SmartBuild* when creating their application.

Second, even if the Process Commander application is designed for peak efficiency, the customer's system must be **tuned** in order to make sure that all of the system structures are built to the appropriate level for the customer's configuration.

NOTE: Although this article describes tuning a production environment, it may also be applied to large-scale development environments. ("Large-scale" would be a system where 50 to 100 developers [or more] were all developing the same Process Commander application. For a development environment where there are only two or three developers, the default cache settings should be fine.)

System Pulse

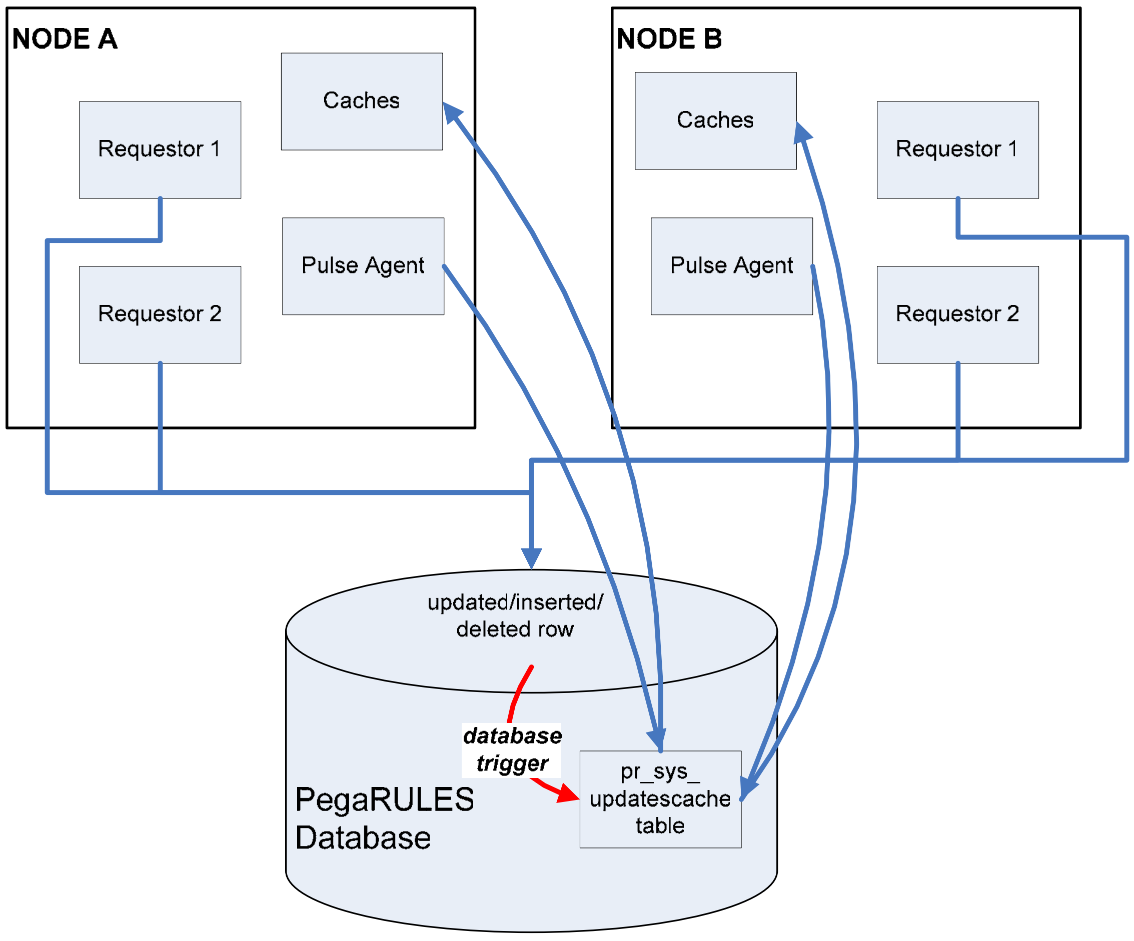
When PegaRULES is installed on a multi-node system, then a copy of the various caches are stored on each node, and each of those nodes must be updated with rule changes. This update process is managed by the **System Pulse** functionality.

Saving a rule change (update, add, or delete) to one of the Rule tables in the database will fire a PegaRULES database trigger, which will then register the change in the **pr_sys_updatescache** table (in the same database). Each node records all their changes in this table; the types of event which are saved to this table include:

- Cache – for any changes to the rule cache (changes to the rules)
- Index – for Lucene changes
- DELLC – when the lookuplist cache is deleted
- RFDEL – for any rule-file- deletes

Saving a rule change will also automatically invalidate the appropriate information in the caches on that node; however, all the other nodes in the system now have out-of-date information about that rule.

Once per minute, the pulse (which is part of the standard PegaRULES agent) on each node wakes up (independently) and queries the pr_sys_updatescache table. The query retrieves records which are "not from this node" (which this node did *not* create), and which have a timestamp which falls within the window of time starting with the last pulse (or system startup) and now. In this way, all of the changes originating on other nodes are selected, and the appropriate cache entries on this node are invalidated. The next time one of the rules which has been invalidated is called, it will not be found in the cache, and the updated version of the rule will be read out of the database and be eligible for caching again.



NOTES ON MULTI-NODE SYSTEMS:

- If one user updates a rule on one node, and then a second user views or uses that rule on another node before the System Pulse has run, the second user will see the *old* rule information. (By default, the System Pulse runs once a minute, so in a production system, this situation should not happen often.)

- The policy for the database cache is that the database ‘owns’ the commit boundary. While instances are being updated, the cache always lets the database decide when the commit is visible to other threads. Between the time when the update is initiated and the update is committed, the cache instance is considered “dirty;” any other thread looking for that instance will get the dirty status, which will cause the database to be read. This means that the visibility of the data being committed is controlled by the database in question and its transaction isolation settings.

- If one user makes a change to a rule that invalidates it in the cache, and another user makes a change that would invalidate that same rule, the second user may get a message stating that “the entry in the cache is already dirty.” This simply means that the entry in question had already been invalidated, and is not a problem with the system.

Clearing Caches

As stated above, clearing the caches (described in the below sections) should be done rarely, if ever. Beginning in version 4.1, the Import Rules process was changed to include an automatic cache adjustment for the imported rules, so **caches do *not* need to be cleared for the Import Rules process.**

For imports which use the ImportExport Servlet, the LookupList and Static Content Caches *only* must be cleared. Again, the other caches are handled by the import facility; no other caches should be cleared by the developer.

When upgrading to another version (Version 4.2 Service Pack 3 to Service Pack 5, for example), the developer should shut down and restart the system after installing the new Service Pack. Shutting down, redeploying, and restarting will effectively clear all of the caches; the developer does not need to clear them manually.

If there is a situation where the caches must be cleared (other than what is described in this document), that may indicate an issue with the product. **Please call Support with a full description of what was being done, the problem that was seen, and how clearing the cache solved the problem** so that this may be researched.

Tools

Process Commander includes a number of tools which may be used to monitor the system and application performance. One of the main tools is the **System Console,** which is available for users with System Administrator access (in the **Monitor System for *nodename*** section of the Dashboard). The System Console monitors the system and displays important information which will be used in this tuning process.

In addition to the System Console, the *Performance Analyzer* (PAL) tracking features should be used to track performance. (A full explanation of PAL is beyond the scope of this article. For complete details, please reference the *Using Performance Tools* technology paper.)

NOTE: PAL is designed to be used all through the development process, to highlight performance issues as they are introduced (instead of waiting until the end of development, when tracing the source of the issue has become more complex, and there is also less time available to consider performance-enhancing design changes). However, **the tuning described in this article should be done *after* the application is completed** – either in the pre-production testing phase, or during a pilot.

The Rule Cache

The Rule cache holds data about the reads that are done from the PegaRULES database for rule classes. Reading information from the database is a very expensive operation, in terms of system time and resources; therefore, the system attempts to do this only when absolutely necessary.

Whenever a Process Commander application needs information from the database, methods such as the following should be used:

- Obj-Open
- Obj-Open by Handle
- Obj-Save
- Obj-Delete

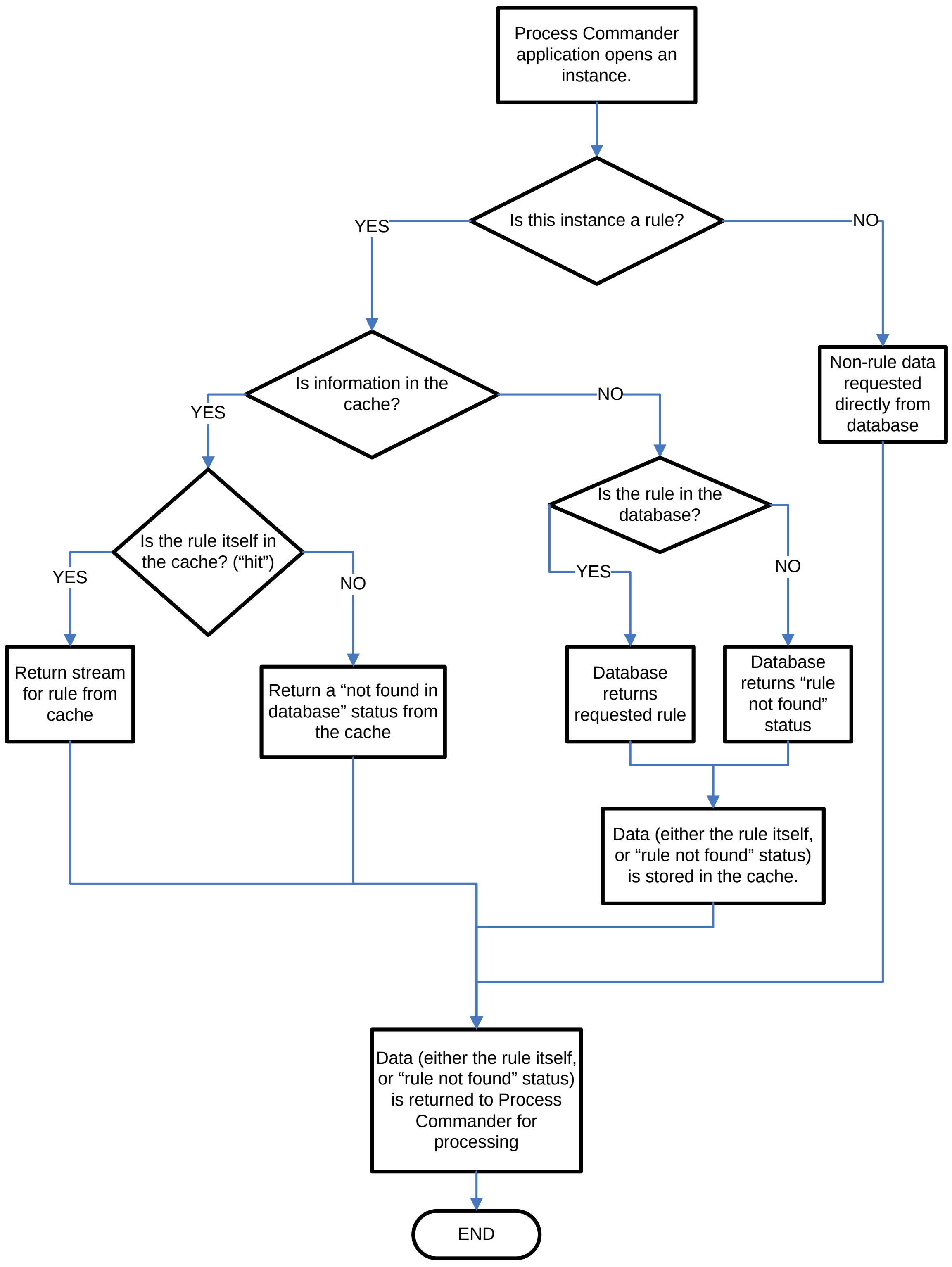
For any non-rule information (such as work objects, which are descended from Work-), the system will bypass the cache and retrieve the information directly from the database. For any rule requests, the system checks the Rule Cache, to see whether this information has been requested before. The cache will return one of three results:

- The data is cached here (“cache hit”)
- The cache has registered the fact that this rule is not in the database (“not found”)
- The cache has never been asked about this data; check the database (“cache miss”)

The first time a rule is requested from the database, the cache will have no information about it. Therefore, it passes the request on to the database, and the system attempts to retrieve the information from the database. If the information is not available from the database, then the system records that fact in the Rule cache and returns a “rule not found” error to the user. (The next time that the application requires that data, the system finds in the cache that the data is not available in the database, and doesn’t have to go out to the actual database and spend the time searching it.)

If the data *is* available in the database, the system will retrieve the rule and store it in the Rule cache, and then send the rule information back to whatever processing requested it. The next time that information is required, it can be retrieved from the cache without having to go all the way to the database for it.

This process is illustrated in the flowchart below:


Process Commander application opens an instance.
Is this instance a rule?
YES
NO
Is information in the cache?
Non-rule data requested directly from database
YES
NO
Is the rule itself in the cache? (“hit”)
Is the rule in the database?
YES
NO
YES
NO
Return stream for rule from cache
Return a “not found in database” status from the cache
Database returns requested rule
Database returns “rule not found” status
Data (either the rule itself, or “rule not found” status) is stored in the cache.
Data (either the rule itself, or “rule not found” status) is returned to Process Commander for processing
END

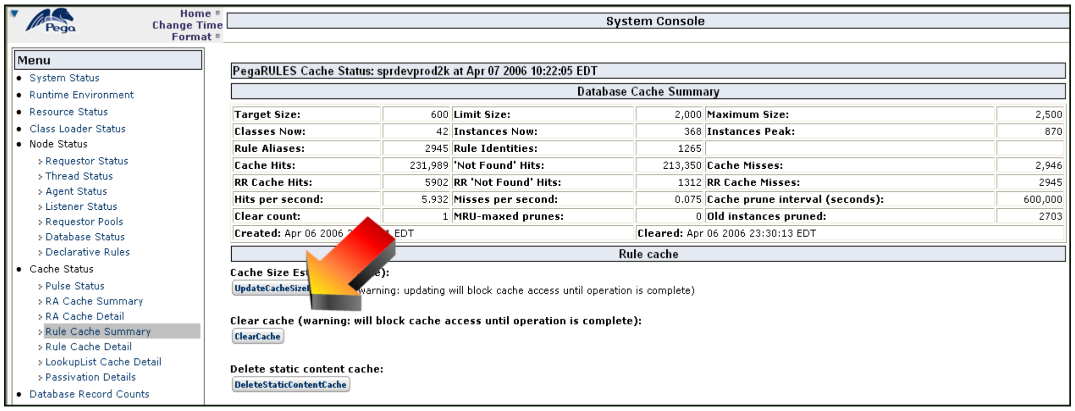
The Rule cache must be kept up-to-date with any rule changes. Therefore, for any changes to a rule (Obj-Save or Obj-Delete), the caching process is a bit different. For those methods, the system will check the cache for any rule related to the changed rule and invalidate those entries, so that subsequent requests for this rule will go to the database and get the updated rule information (rather than getting the outdated information from the cache). The system will then save or delete the rule data to the database.

An additional efficiency was added to the Rule cache system. There are some rules which may only be used once – for starting up the system, for example, or for logging in one user. Caching these rules would waste time and cache space (because they won't be used again). Therefore, the system keeps track of how many times a rule is requested; rules do not get recorded in the cache until they have been accessed a minimum of three times. Entries that are in this "limbo" state are considered on probation, and act much like invalidated entries – they continue to cause a database read to get the data until they have been requested three times.

Tuning the Rule Cache

After the application has been created, the developer should determine the appropriate size for the Rule cache. Every system is different in how it accesses rules from the database[1]. The developer should start up the Process Commander application, and run the system for an hour or so. During this time, they should exercise all the important parts of the application – create and resolve all the different types of work objects, run reports that might be used frequently, make sure any defined agent processing has occurred and that all required service and connect rules have been used, and any other processing that will occur in normal use. (NOTE: It is especially important to go through everything at least once, to make sure that all the rules used have been Rules Assembled the first time they are called.) The developer should keep track of all the functions that were run.

After the developer is certain that the system has exercised all of the main-line rules, they should clear the Rule cache. From the System Console, highlight the **Rule Cache Summary** page. Click on the **ClearCache** button to clear the database cache.



System Console

PegaRULES Cache Status: sprdevprod2k at Apr 07 2006 10:22:05 EDT

Database Cache Summary

Target Size:	600	**Limit Size:**	2,000	**Maximum Size:**	2,500
Classes Now:	42	**Instances Now:**	368	**Instances Peak:**	870
Rule Aliases:	2945	**Rule Identities:**	1265		
Cache Hits:	231,989	**'Not Found' Hits:**	213,350	**Cache Misses:**	2,946
RR Cache Hits:	5902	**RR 'Not Found' Hits:**	1312	**RR Cache Misses:**	2945
Hits per second:	5.932	**Misses per second:**	0.075	**Cache prune interval (seconds):**	600,000
Clear count:	1	**MRU-maxed prunes:**	0	**Old instances pruned:**	2703
Created: Apr 06 2006 [illegible] EDT			**Cleared:** Apr 06 2006 23:30:13 EDT		

Rule cache

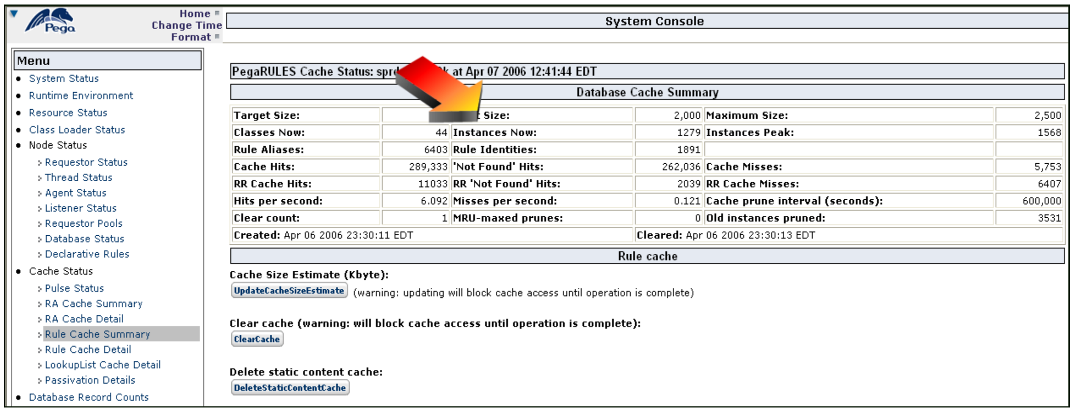
Cache Size Es[illegible]):
UpdateCacheSize[illegible] [illegible]warning: updating will block cache access until operation is complete)

Clear cache (warning: will block cache access until operation is complete):
ClearCache

Delete static content cache:
DeleteStaticContentCache

[1] Note that not every rule access is to the database or cache. The Rules Assembly process will generally access the rules that it converts to Java only once in order to do the assembly – after that, accessing that rule will directly execute the generated Java code. Se the section below on the Rules Assembly Process.

After the cache has been cleared, the developer should go back into the system and exercise all the important parts of the application again (as had been done before). After the system has run for another hour or so, and everything has been run at least once (and preferably several times), the developer should go back into the Rule Cache Summary page and look at the **Database Cache Summary** statistics, especially the **Instances Now** reading.



PegaRULES Cache Status: spr[illegible] at Apr 07 2006 12:41:44 EDT

Database Cache Summary

Target Size:		[illegible] Size:	2,000	Maximum Size:	2,500
Classes Now:	44	Instances Now:	1279	Instances Peak:	1568
Rule Aliases:	6403	Rule Identities:	1891		
Cache Hits:	289,333	'Not Found' Hits:	262,036	Cache Misses:	5,753
RR Cache Hits:	11033	RR 'Not Found' Hits:	2039	RR Cache Misses:	6407
Hits per second:	6.092	Misses per second:	0.121	Cache prune interval (seconds):	600,000
Clear count:	1	MRU-maxed prunes:	0	Old instances pruned:	3531
Created: Apr 06 2006 23:30:11 EDT			Cleared: Apr 06 2006 23:30:13 EDT		

Rule cache

Cache Size Estimate (Kbyte):

(warning: updating will block cache access until operation is complete)

Clear cache (warning: will block cache access until operation is complete):

Delete static content cache:

Instances Now is the approximate size of the rule cache required for the database. (NOTE: This includes a count of probationary items.)

Rule Cache Settings

Once the appropriate size of the Rules cache has been determined, settings should be added to the **pegarules.xml** file (or changed, if they already exist). In the **cache** node, change the **instancecountlimit** setting to be 1 ½ to 2X the value of Instances Now. (In the example above, Instances Now = 1279 rules; the *instancecountlimit* could be set to between 1900 and 2600.)

```
<node name="cache">
    <map>
      <entry key="instancecountlimit" value="2500" />
  </map>
</node>
```

All nodes in a multi-node system should be given the same settings, unless the system is designed with "specialized" nodes that will handle specific processing (such as processing for services or agents only). Each of these specialized nodes should also go through the above tuning procedure, to tune them separately.

Apart from the above tuning procedure, the only time to clear this cache is if a problem happens with the system pulse in a multi-node system, and the caches on each node are in an unknown state. Clearing the Rule cache at that point will invalidate all the entries in all the caches, so that (after the system pulse is repaired) all the data that is cached is up-to-date.

Otherwise, all that clearing the Rule cache will do is slow the system down. This cache was created to provide efficiency in rule lookup; clearing the cache means that all that cached information will have to be rebuilt, and will provide no other benefit.

Rule Assembly Cache

Background – the Rules Assembly Process

Rules Assembly is the process whereby all of the rules which are run by the Process Commander system are generated into Java code for the system to execute. Since Rules Assembly is an expensive process (in terms of system resources), the process has been broken down into several steps to maximize performance and efficiency.

When the system calls a rule to be run, if that rule was never executed, the Rules Assembly process must:

- assemble the rule, including all the other rules required (for example, if this is an activity rule, it may call other rules)
- generate the Java code for the rule
- compile the code
- load and execute the resulting class file

Of these steps, the code compilation is the most expensive. Therefore, several Rules caches have been added to try to minimize Java code generation and compilation.

During Rules Assembly, one of the key factors is the user's RuleSet List, as that controls which rules the user may see. Two users with different RuleSet Lists may call the same named rule, but due to their different Lists, the rules selected and the code ultimately generated may be quite different. Similarly, it is possible for two users with different-yet-related RuleSet Lists to call for the same named rule; because the rules are defined entirely in the RuleSets they have in common, the generated class is exactly the same. Due to these factors, a cache structure is maintained in memory, keyed by a combination of the RuleSet List and Rule Name (its *purpose*) that maintains a reference to the generated class file's loaded instance.

A further refinement of this setup handles the case of developers (users who have checkout rights) who may have the same RuleSet Lists, but will have a unique personal RuleSet in their RuleSet Lists, as those are specific to each user. Some users who have access to check out rules may be running a different set of rules (based on what is in their personal RuleSet List); therefore, the rule execution is again different. The Rules Assembly cache tracks these entries in two separate lists: Global and Personal.

Thus, the full process for executing a rule is as follows:

1. The user calls a rule.

2. Rule Resolution occurs, to determine what rule should actually be run.

3. The rule is searched for in the Personal Cache. If found, that code is executed.

4. If not found, the source code for this Rule's Java class is assembled *but not compiled* to generate the Java class name, which is used to check whether this code is in the Global Cache. If found, that reference is copied to the personal cache and the code is executed.

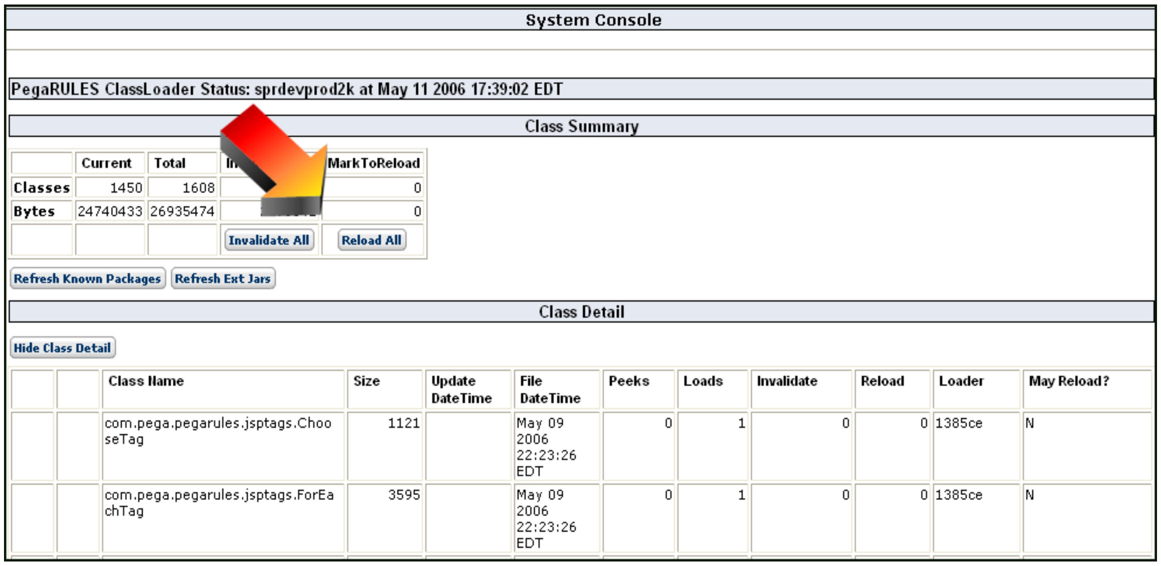
5. If the reference is not found, then the class name is used to check whether this code has been stored on disk as a .class file. If found, the code is loaded by the Java

classloader and executed (the Rules Assembled source code is discarded, as it is still far more efficient to assemble and discard than it is to compile).

6. If the .class file is not found, the Rule's generated source code is compiled, stored in the caches (and on disk), loaded via the Java classloader and executed.

Cache Functionality

The Global and the Personal Caches are thus integral to system performance, since if they weren't there, Rules Assembly and Java code generation and compilation would have to occur over and over every time a rule is used. Since one interaction with the server – such as opening a work item – could run thousands of rules, making sure Rules Assembly is as efficient as possible is vital.


System Console

PegaRULES ClassLoader Status: sprdevprod2k at May 11 2006 17:39:02 EDT

Class Summary

	Current	Total	In[illegible]	MarkToReload
Classes	1450	1608	[illegible]	0
Bytes	24740433	26935474	[illegible]	0
			Invalidate All	Reload All

Class Detail

		Class Name	Size	Update DateTime	File DateTime	Peeks	Loads	Invalidate	Reload	Loader	May Reload?
		com.pega.pegarules.jsptags.ChooseTag	1121		May 09 2006 22:23:26 EDT	0	1	0	0	1385ce	N
		com.pega.pegarules.jsptags.ForEachTag	3595		May 09 2006 22:23:26 EDT	0	1	0	0	1385ce	N

In the System Console, in the **Class Loader Status** page, it is possible to invalidate all classes in the Rules Assembly cache, and also to reload them all. In the normal course of business, it should **never** be necessary to do this. Clicking *Invalidate All* will unload *everything* - the generated Java code (forcing it to be regenerated and recompiled), the Rule-Utility-Library rules, any third-party .jar or class files which were referenced in rules. This will be a noticeable performance hit, and should never be required.

Clicking **ReloadAll** will reload into the system all the classes; this will take quite a while, but should get the system back to the prior performance level, as it regenerates all the classes at once.

The size of the Global and Personal caches may be changed by the customer. The optimal size for each cache will vary based on the customer's application and rules use, and should be tuned by the application developer for maximum performance efficiency.

If a cache is *too large*, then *everything* gets cached, even if the rules are used only rarely. This will take up a lot of memory in the system (as well as space on the disk), and doesn't always provide any benefit.

However, more importantly, if the caches are *too small,* then the system begins to “thrash.” If the limit on the cache is set to (for example) 1000 rules, and the opening of a work item takes 2000 rules, then what happens is that Rules Assembly will occur for the first 1000 rules used in this process (as it should). This information is cached. Then the next 1000 rules are *also* Rule-Assembled, and *they* are cached; as assembly continues, the first 1000 are invalidated to make room in the cache for the next 1000. The next time someone tries to open a work item, the first 1000 rules must be re-Rules-Assembled (as they were invalidated), and then cached, invalidating the old rules there; then the second 1000 rules are *again* Rules-Assembled (as they were just invalidated), and re-cached. The system spends all its time redoing Rules Assembly and Java code generation and compilation, which results in extremely poor performance.

Tuning the Rules Assembly Cache

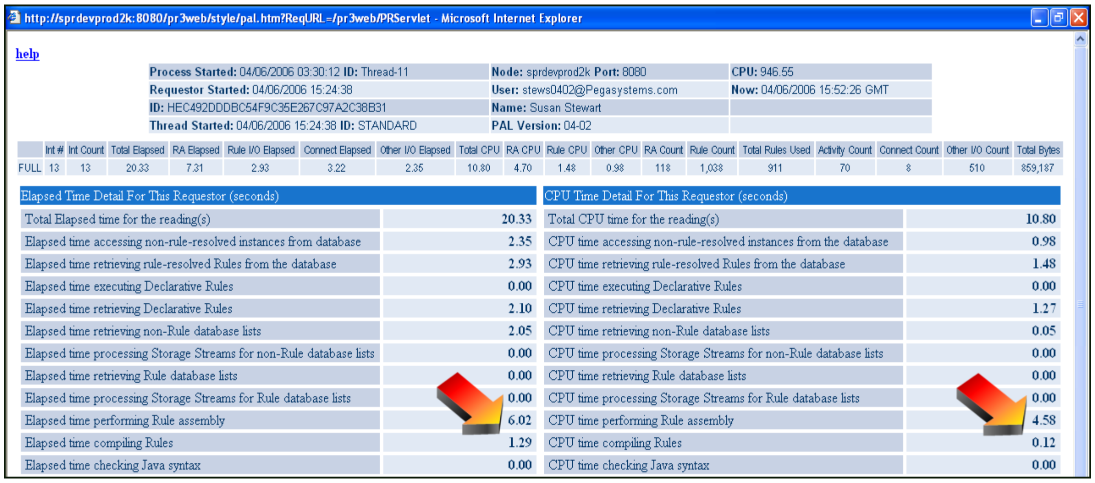
Using PAL

After the application is completed, the developer should take PAL readings while creating one of each type of work object in the application. The first time each type of work object is created, Rules Assembly will occur. *Rules Assembly* is the process of generating and compiling Java code for each of the rules used in the work object processing; this Java code is generated once, and then stored on disk for future use.

Because Rules Assembly has a big impact on performance and execution times, Pegasystems recommends that all performance evaluations are done in two passes – a first pass to assemble all of the rules into Java, and a second which is the actual measurement run.

When taking PAL readings on the second run-through, the developer should see time spent in the following PAL counters:

- Elapsed time performing Rule assembly
- Elapsed time compiling Rules
- CPU time performing Rule assembly
- CPU time compiling Rules



http://sprdevprod2k:8080/pr3web/style/pal.htm?ReqURL=/pr3web/PRServlet - Microsoft Internet Explorer

help

Process Started: 04/06/2006 03:30:12 **ID:** Thread-11	**Node:** sprdevprod2k **Port:** 8080	**CPU:** 946.55
Requestor Started: 04/06/2006 15:24:38	**User:** stews0402@Pegasystems.com	**Now:** 04/06/2006 15:52:26 GMT
ID: HEC492DDDBC54F9C35E267C97A2C38B31	**Name:** Susan Stewart	
Thread Started: 04/06/2006 15:24:38 **ID:** STANDARD	**PAL Version:** 04-02	

	Int #	Int Count	Total Elapsed	RA Elapsed	Rule I/O Elapsed	Connect Elapsed	Other I/O Elapsed	Total CPU	RA CPU	Rule CPU	Other CPU	RA Count	Rule Count	Total Rules Used	Activity Count	Connect Count	Other I/O Count	Total Bytes
FULL	13	13	20.33	7.31	2.93	3.22	2.35	10.80	4.70	1.48	0.98	118	1,038	911	70	8	510	859,187

Elapsed Time Detail For This Requestor (seconds)		CPU Time Detail For This Requestor (seconds)	
Total Elapsed time for the reading(s)	20.33	Total CPU time for the reading(s)	10.80
Elapsed time accessing non-rule-resolved instances from database	2.35	CPU time accessing non-rule-resolved instances from the database	0.98
Elapsed time retrieving rule-resolved Rules from the database	2.93	CPU time retrieving rule-resolved Rules from the database	1.48
Elapsed time executing Declarative Rules	0.00	CPU time executing Declarative Rules	0.00
Elapsed time retrieving Declarative Rules	2.10	CPU time retrieving Declarative Rules	1.27
Elapsed time retrieving non-Rule database lists	2.05	CPU time retrieving non-Rule database lists	0.05
Elapsed time processing Storage Streams for non-Rule database lists	0.00	CPU time processing Storage Streams for non-Rule database lists	0.00
Elapsed time retrieving Rule database lists	0.00	CPU time retrieving Rule database lists	0.00
Elapsed time processing Storage Streams for Rule database lists	0.00	CPU time processing Storage Streams for Rule database lists	0.00
Elapsed time performing Rule assembly	6.02	CPU time performing Rule assembly	4.58
Elapsed time compiling Rules	1.29	CPU time compiling Rules	0.12
Elapsed time checking Java syntax	0.00	CPU time checking Java syntax	0.00

The **Rule assembly** counts track the amount of time spent generating the Java source code for the rules used in the work object. This is a moderately expensive process, which the system seeks to avoid repeating (when possible).

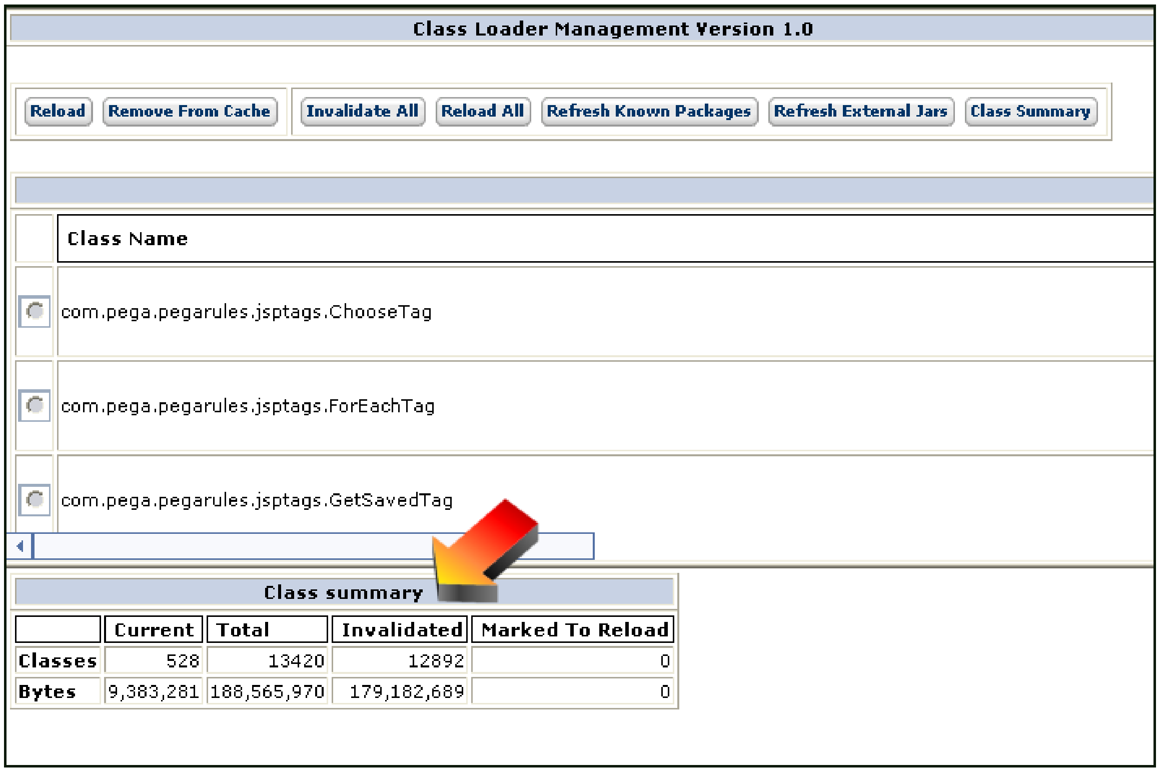
The **compiling Rules** counts track the amount of time required for the rule source (generated in Rule assembly) to be compiled by a Java compiler and loaded into memory. This is a *very* expensive process, which the system seeks to avoid repeating as much as possible.

Once the rules have been generated the first time, these counters should be zero for subsequent readings during the creation of additional work objects. **If Rule Assembly continues to occur, this may point to a tuning issue with the Rules Assembly structures.**

NOTE: Following our “Build for Change” philosophy, if any rules are changed in the system, they would automatically be regenerated and recompiled as needed (and the above counts would reflect these actions).

Using the System Console

Another pointer may be found in the **System Console** data. In the **Classloader Status** page, check the **Class Summary** section:



	Class Name
	com.pega.pegarules.jsptags.ChooseTag
	com.pega.pegarules.jsptags.ForEachTag
	com.pega.pegarules.jsptags.GetSavedTag

Class summary

	Current	Total	Invalidated	Marked To Reload
Classes	528	13420	12892	0
Bytes	9,383,281	188,565,970	179,182,689	0

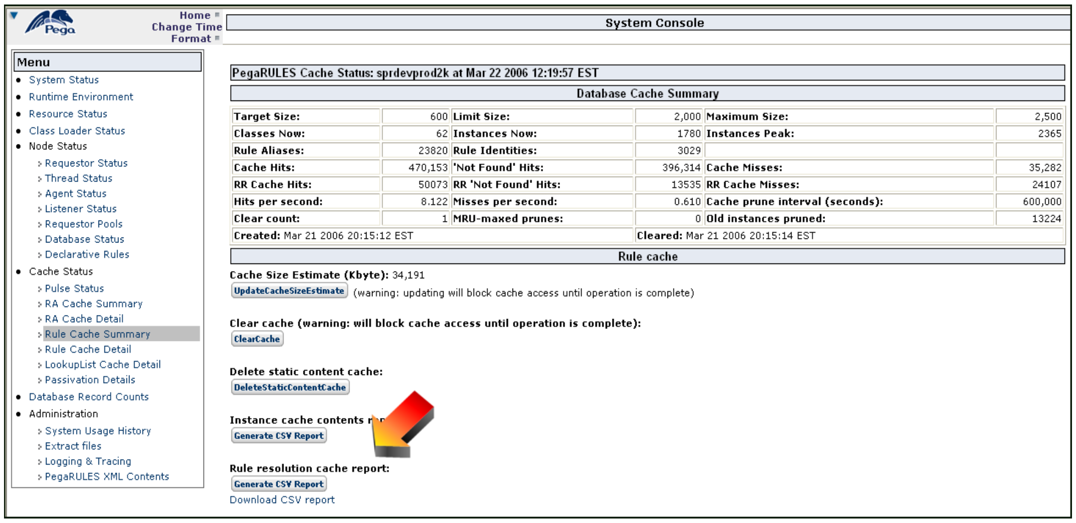
This section shows how many classes are currently stored in the system, and how many have been invalidated. If the same number of classes is invalidated every time a work object is created, that indicates that the “working set” of the rules required to implement the application processing has grown to be larger than the default structures provided by Rules Assembly.

Note that in the above example, the system has a **Current** list of 528 classes in the cache, and 12,892 classes have been **Invalidated.** This is an indication that the "working set" for Rules Assembly is too small. There should always be *some* invalidated classes, but if the number of invalid classes is several orders of magnitude larger than the current cached classes, that is a real problem. This will also cause Rules Assembly to reoccur (continuing to have readings in the Rule assemblies and Rules Compiled PAL counters).

To determine what would be an appropriate size for the customer cache, begin by determining how many unique RuleSet Lists are being used in the system. This may be found by using the Rule Resolution cache report (again, from the System Console).

NOTE: These steps should be followed after the system has been used in production for some time, so there is enough data about the system to give a correct picture of the situation.

On the **Rule Cache Summary** page, click on **Generate CSV Report** under **Rule resolution cache report**.



PegaRULES Cache Status: sprdevprod2k at Mar 22 2006 12:19:57 EST

Database Cache Summary

Target Size:	600	Limit Size:	2,000	Maximum Size:	2,500
Classes Now:	62	Instances Now:	1780	Instances Peak:	2365
Rule Aliases:	23820	Rule Identities:	3029		
Cache Hits:	470,153	'Not Found' Hits:	396,314	Cache Misses:	35,282
RR Cache Hits:	50073	RR 'Not Found' Hits:	13535	RR Cache Misses:	24107
Hits per second:	8.122	Misses per second:	0.610	Cache prune interval (seconds):	600,000
Clear count:	1	MRU-maxed prunes:	0	Old instances pruned:	13224
Created: Mar 21 2006 20:15:12 EST			Cleared: Mar 21 2006 20:15:14 EST		

Rule cache

Cache Size Estimate (Kbyte): 34,191
UpdateCacheSizeEstimate (warning: updating will block cache access until operation is complete)

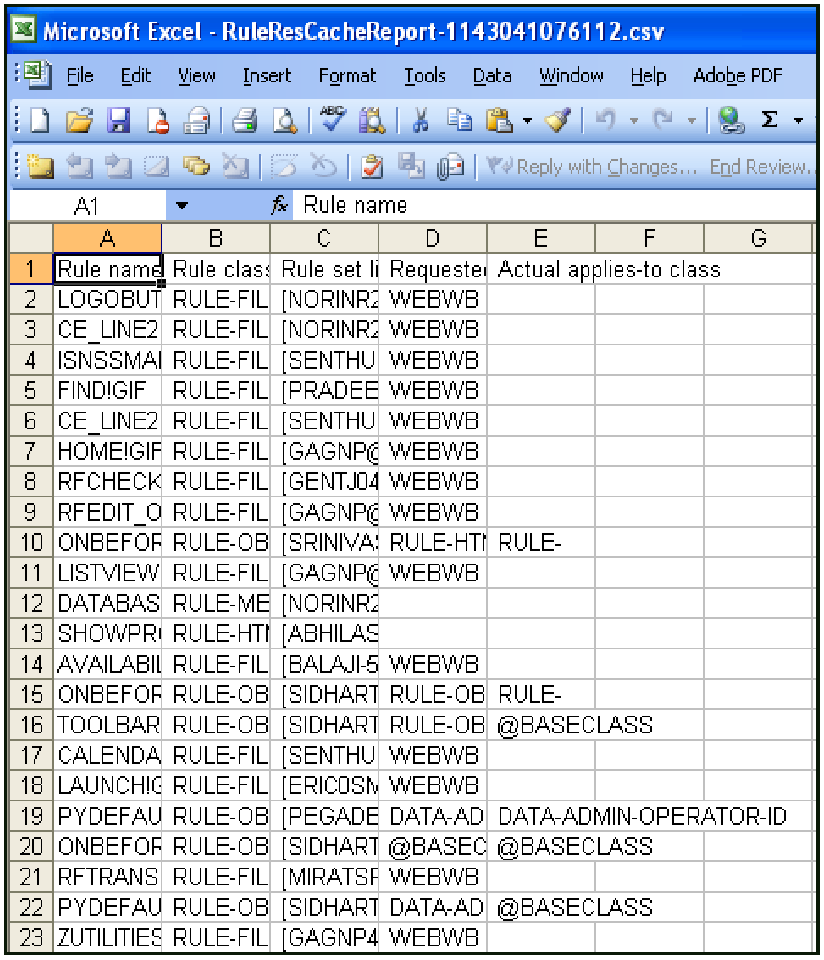
Clear cache (warning: will block cache access until operation is complete):
ClearCache

Delete static content cache:
DeleteStaticContentCache

Instance cache contents r
Generate CSV Report

Rule resolution cache report:
Generate CSV Report
Download CSV report

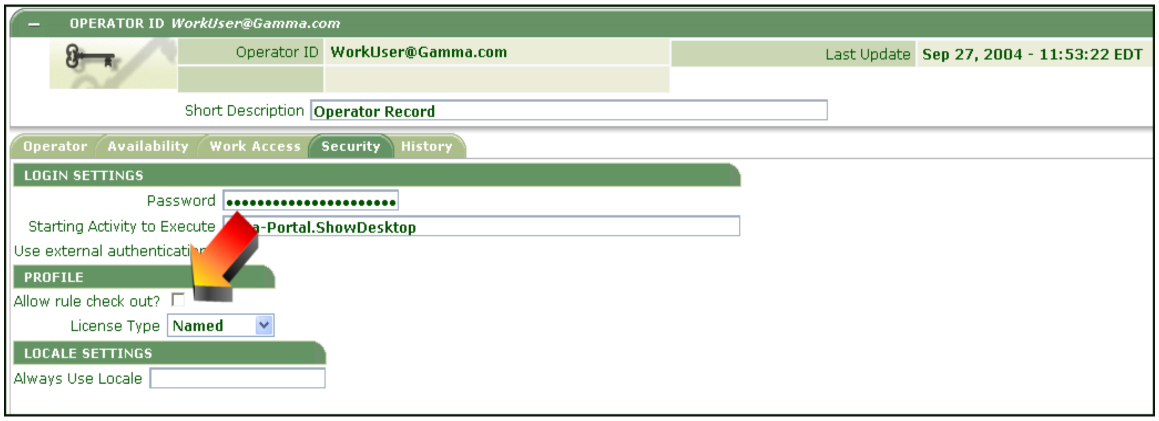
The link *Download CSV Report* will be displayed *after* the button has been clicked. Click on this link to download the report to the local PC, and then open the report using Excel. (The report will have a name like "RuleResCacheReport-1143041076112.csv".)


Microsoft Excel - RuleResCacheReport-1143041076112.csv
File Edit View Insert Format Tools Data Window Help Adobe PDF
Reply with Changes... End Review...
A1
Rule name

	A	B	C	D	E	F	G
1	Rule name	Rule class	Rule set li	Requester	Actual applies-to class		
2	LOGOBUT	RULE-FIL	[NORINR2	WEBWB			
3	CE_LINE2	RULE-FIL	[NORINR2	WEBWB			
4	ISNSSMAI	RULE-FIL	[SENTHU	WEBWB			
5	FIND!GIF	RULE-FIL	[PRADEE	WEBWB			
6	CE_LINE2	RULE-FIL	[SENTHU	WEBWB			
7	HOME!GIF	RULE-FIL	[GAGNP@	WEBWB			
8	RFCHECK	RULE-FIL	[GENTJ04	WEBWB			
9	RFEDIT_O	RULE-FIL	[GAGNP@	WEBWB			
10	ONBEFOR	RULE-OB	[SRINIVAS	RULE-HTI	RULE-		
11	LISTVIEW	RULE-FIL	[GAGNP@	WEBWB			
12	DATABAS	RULE-ME	[NORINR2				
13	SHOWPR	RULE-HTI	[ABHILAS				
14	AVAILABI	RULE-FIL	[BALAJI-5	WEBWB			
15	ONBEFOR	RULE-OB	[SIDHART	RULE-OB	RULE-		
16	TOOLBAR	RULE-OB	[SIDHART	RULE-OB	@BASECLASS		
17	CALENDA	RULE-FIL	[SENTHU	WEBWB			
18	LAUNCH!G	RULE-FIL	[ERICOSM	WEBWB			
19	PYDEFAU	RULE-OB	[PEGADE	DATA-AD	DATA-ADMIN-OPERATOR-ID		
20	ONBEFOR	RULE-OB	[SIDHART	@BASEC	@BASECLASS		
21	RFTRANS	RULE-FIL	[MIRATSF	WEBWB			
22	PYDEFAU	RULE-OB	[SIDHART	DATA-AD	@BASECLASS		
23	ZUTILITIES	RULE-FIL	[GAGNP4	WEBWB			

The third column is the RuleSet List column. Expand the columns a little (for easier viewing), and then sort the spreadsheet on this column. There may be thousands of entries in this spreadsheet, but once sorted, for the purposes of determining unique RuleSet Lists, delete all the duplicates, so that only one of each RuleSet List is left; then count those. In a production system, there probably should be less than 100 unique RuleSet Lists, depending upon the customer's application setup.

Important: If there are a great many unique RuleSet Lists (i.e., over 100) with a production system, then there may be another problem with the users' setup. In the Operator ID record for each user, on the **Security** tab, there is a field in the **Profile** section called **Allow Rule Check Out.**



Operator ID WorkUser@Gamma.com
Last Update Sep 27, 2004 - 11:53:22 EDT
Short Description Operator Record
Password ●●●●●●●●●●●●●●●●●●●●●
Starting Activity to Execute a-Portal.ShowDesktop
Use external authenticati
Allow rule check out? ☐
License Type Named
Always Use Locale

This field controls whether the user has access to check out rules to change them. *No end user should have access to change rules in the system.*

For each user who has **Allow rule check out?** checked, a unique RuleSet List will be generated, which includes a personal RuleSet for that user (to hold checked-out rules). Checking this box for users who will not be checking out rules causes the system to create unnecessary structures and do unnecessary checking during rule processing. *Developers should verify that only users who will be changing rules in the system should have this box checked on their Operator ID.*

Another cause of too many unique RuleSet Lists might be users with the same RuleSet Lists, but in different order. For example, users in one Access Group might have the following Production RuleSets:

AcmeDev:01-03
AcmeEngineering:02-01
AcmeBASE:01-01

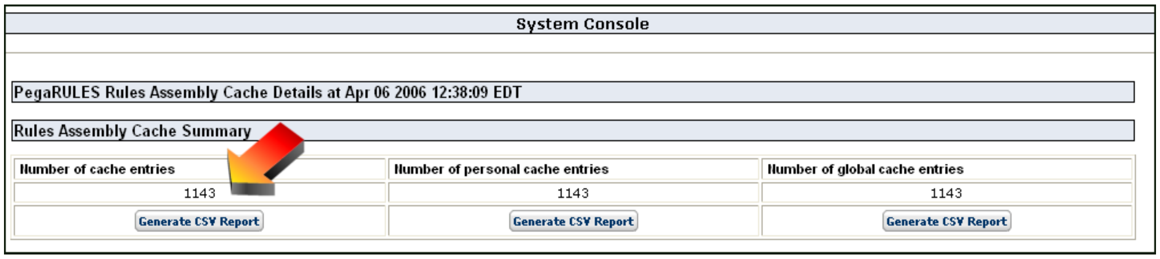
Users in a different access group might have:

AcmeEngineering:02-01
AcmeDev:01-03
AcmeBASE:01-01

Although the users in both of these Access Groups are doing similar work and have similar access, and could use the same Access Group, the fact that they are in separate Access Groups with RuleSets in a *different* order will cause unnecessary unique RuleSet Lists in the system[2].

After determining how many unique RuleSet Lists are being used in the system, determine how many unique rules have been cached, by looking at the **Rules Assembly Cache Summary**, which will display the number of cache entries.

[2] Ruleset Lists in different orders can also be a source of many seemingly random, hard-to-track-down bugs in the system. The selection of what rules are used is dramatically affected by the order of the RuleSets.



System Console

PegaRULES Rules Assembly Cache Details at Apr 06 2006 12:38:09 EDT

Rules Assembly Cache Summary

Number of cache entries	Number of personal cache entries	Number of global cache entries
1143	1143	1143
Generate CSV Report	Generate CSV Report	Generate CSV Report

Multiply this number by the number of unique RuleSet Lists, in order to estimate how large the cache size should be. For example, if the customer has 25 unique RuleSet Lists, and there are 1,000 unique rules being used, then the cache should be set to hold 25,000 rules.

Rules Assembly Settings

Once the appropriate size of the Rules Assembly cache has been determined, settings should be added to the **pegarules.xml** file (or changed, if they already exist). In the **fua** node, change the **instancecountlimit** setting in both the **global** and **personal** subnodes:

```
<node name="fua">
    <node name="global">
        <map>
          <entry key="instancecountlimit" value="20000" />
     </map>
    </node>
    <node name="personal">
     <map>
          <entry key="instancecountlimit" value="20000" />
     </map>
    </node>
</node>
```

The value for both of these entries should start with the value that was calculated from the unique RuleSet Lists and the Number of Cache Entries. After obtaining the beginning value, the developer should monitor the system and if too many classes are being invalidated, run through the above process again and tweak the *instancecountlimit* values.

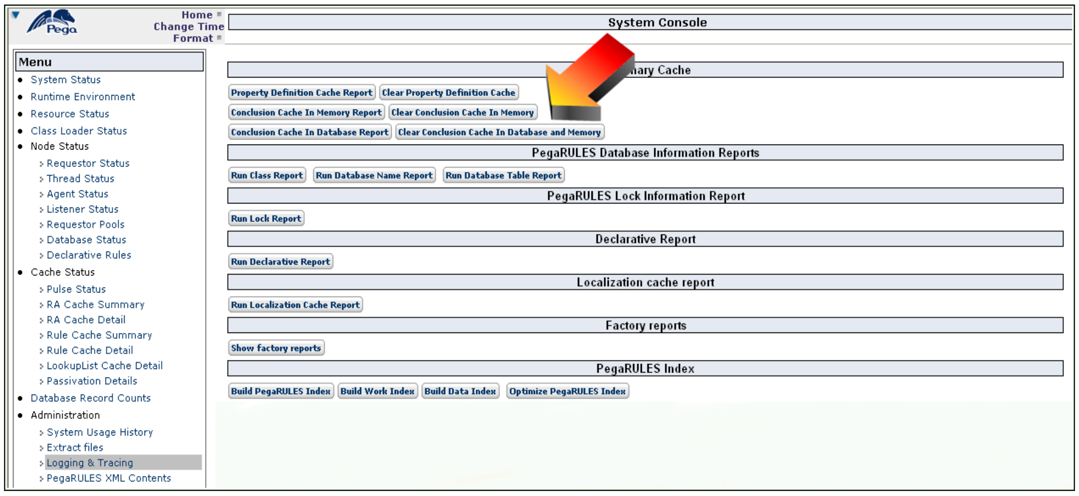
Dictionary Caches

The Dictionary Caches store class and property information.

In the **Logging & Tracing** section of the System Console, find the **Dictionary Cache** section.

The following buttons are available:

- Property Definition Cache Report
- Clear Property Definition Cache
- Conclusion Cache in Memory Report
- Clear Conclusion Cache in Memory
- Conclusion Cache in Database Report
- Clear Conclusion Cache in Database and Memory



“Conclusion” Cache

A *conclusion* is an instance of a Java object (resulting from the Rules Assembly process) that was generated, instantiated, and then persisted to the database for future use. The unique feature of conclusions that differentiate them from “normal” Java object persistence is that these classes also rebuild themselves in response to rule changes (just like all other types of rules). The benefit of conclusions is that they group a number of similar rules, and digest the information down to the minimum amount required at runtime.

For example, the **Field Value** Conclusions will group all the Field Values present for one Field Value name – it will group the “Add Attachments” Field Value from Pega-ProCom, and from Pega-ProCom_de, and Pega-ProCom_fr, and Pega-ProCom_es, etc – a single instance of a common set of data. For localization, if there are two Field Values with the same key name (in the *Field Value* field - example: “Add Item”) defined on different “grouping properties” (the Field Name – examples: “pyButtonLabel,” “pyMessageLabel”), then those Field Values are also grouped together in one Conclusion.

For each Field Value, there is a lot of extraneous information that the system does not require at runtime – the history of the Field Value, who created the rule, the date/time it was created, the last time it was updated, etc. The Conclusion just holds the runtime data (the key values, the RuleSet and Version, etc.) When the "Add Attachments" Field Value is required, the system will first check to see if this data is in memory in the Conclusion cache. If not, the system will retrieve the required information from the database, and then cache it as a Conclusion.

NOTE: If data is in the Conclusion cache, it will *not* be stored in other caches, such as the Rule Cache or the Rules Assembly Cache.

Unlike other caches, the Conclusion data is also stored in the database, as an instance of the **System-Conclusion** class. Although retrieving information from the database is expensive, retrieving a Conclusion is less expensive, as it is a grouping of other rules. Only one database read is required, versus multiple reads if the rules in the Conclusion were read individually.

If any of the rules in a Conclusion grouping are changed, then the cache is cleared for that Conclusion, and the new conclusion with the new data is saved into the System-Conclusion class.

The Conclusion cache stores several different types of data:

Data type	Information Stored in Conclusion Cache
Property	the runtime information required for properties (examples: Class, Name, RuleSet, Key, Entry Code, Edit Validate, Is Blocked, Is Lightweight, StreamName, Table Option)
Property Alias	the runtime information required for instances of Rule-Obj-Property-Alias (examples: Name, Version, Create Date)
Class	the runtime information required for instances of Rule-Obj-Class (includes the properties of that class, which vary from class to class)
Field Value	the runtime information required for instances of Rule-Obj-FieldValue (examples: "applies to" class, Field Name, Field Value, RuleSet, Key, IsBlocked, Localized Value)

To view this data, run the Conclusion Cache reports (either in Memory or in Database).

The Conclusion caches should rarely, if ever, need to be cleared. Clearing these caches will have an enormous negative impact on system performance until they have been rebuilt. The *only* time they should be cleared is if the system doesn't seem to be updating these caches correctly: If a property which was just defined on a page can't be resolved when running an activity, and an error is displayed stating that the system can't find the property even though the rule was created, that *might* be an issue with the Conclusion cache. **Do not clear this cache unnecessarily.**

Property Definition Cache

The Property Definition cache is a cache of all the Rule-Obj-Property instances used by the system. As properties are looked up with great frequency, this cache is needed for further system efficiencies.

NOTE: This cache is different from the Property data in the Conclusion cache (above). The Conclusion cache contains aggregated filtered data about the properties, defined by name. The Definition Cache contains a complete definition about specific property instances.

Static Content Cache

In PegaRULES, static content files contain information presented to users which doesn't need to be constantly processed, such as form images or help files (example: the login "splash" screen, which will not be affected by changes to rules or to work items). In PegaRULES Process Commander, the main Rules which contain static content are:

- Rule-File-Binary
- Rule-File-Form
- Rule-File-Text

As they are requested, these rules are read from the database using the Rule Resolution process, and the files are cached to the server file system (*not* into memory). Since these files don't change rapidly and are expensive to read from the database, they can persist in the file system through system shutdown and startup, and be retrieved as a file, rather than spending the resources to constantly retrieve the data from the database.

The static content data is actually cached in two places:

- on the client (each user's PC)
- on the server

Static Content Cached on the Client

This content is standard browser caching. Whenever a user requests static content (Rule-File- forms, Help forms, and other static content in the WAR file), that information will be cached in the browser on the client side. It is possible to set an expiration tag on this information, which will tell the browser to look first in its cache for data before asking the server again. Looking locally before sending a request across the network causes less traffic between the browser and the server, and gives better system performance; however, caching static content locally will reduce the ability of the system to reflect updates to static content instantly.

An expiration tag is a date (which essentially tells the browser, "don't ask me about this file until *date*"). This tag is set by PegaRULES, and by default is always 24 hours after the initial request for that form by that browser (client setup). Thus, if no other settings change the tag, the cached data will expire 24 hours from the initial request, and the next time that content is required, the system will automatically request it directly from the server again.

It is possible to override the 24-hour default setting by using the **DefaultCachingTimeout** setting in the **Initialization** section of the **pegarules.xml** file. The DefaultCachingTimeout holds the time, in *seconds*, that the system will use the cached data on the client machine.

```
<node name="Initialization">
   <map>
         <entry key="DefaultCachingTimeout" value="43200"/>
   </map>
</node>
```

Note that the user is always free to force a re-request of the file from the server by pressing the REFRESH button in the browser, which will bypass the caching setting and get the file.

NOTES:

- In Internet Explorer, using F5 will *not* force a refresh of Javascript files. In order to get these files refreshed, it is necessary to go into the Tools/Internet Options/Delete Files and check the "Delete All Offline Content" check box as well.

- If there is a change to one of the Rule-File- forms or other static content during the time before expiration, the user will not be automatically signaled to press the Refresh button, and may not see the change until the default time has expired, or the user clears their cache.

Static Content Cached on the Server

As with the Rules Assembly data, the Static Content files are differentiated based on users' RuleSets and whether the user has a Personal RuleSet List. The files are stored in the following directory structures, with directory names which are a hashname based on the user's RuleSet List.

```
contextroot\webwb
contextroot\rule_cache\webwb\RuleSethashname
contextroot\rule_cache\webwb\RuleSethashname\PersonalRuleSethashn
ame
```

The main RuleSet directory is for users without Personal RuleSet Lists. Since the users of most enterprise installations should use the same RuleSet List groupings (possibly differentiated only by their personal checkout RuleSet), there should be a limited number of these subdirectories.

```
example:  prweb\rule_cache\webwb\f63b7f831c773698125a50d5a4fe551a
```

For *each* user using a personal RuleSet List, the second directory (\rule_cache\webwb\*RuleSethashname**PersonalRuleSethashname*) is created under the above directory, to hold their Personal RuleSet.

```
example:
prweb\rule_cache\webwb\f63b7f831c773698125a50d5a4fe551a\
bbc338b9a51f437b817ef3d9890ddcc4
```

Thus, there may only be a few *RuleSethashname* directories, but there may be hundreds of user Personal RuleSet directories under each main RuleSet directory. This structure is important to minimize the number of copies of files that coexist on disk in different directories. The only items that show up in the personal RuleSet directories are static content files that have been checked out for each user.

The Process

The first time information from a static file is requested, the system searches the directory structure for the requested rule, in the following order:

- Personal RuleSet subdirectories (\rule_cache\webwb\RuleSethashname\PersonalRuleSethashname)
- RuleSet List directories (\rule_cache\webwb\RuleSethashname)
- the WebWB directory

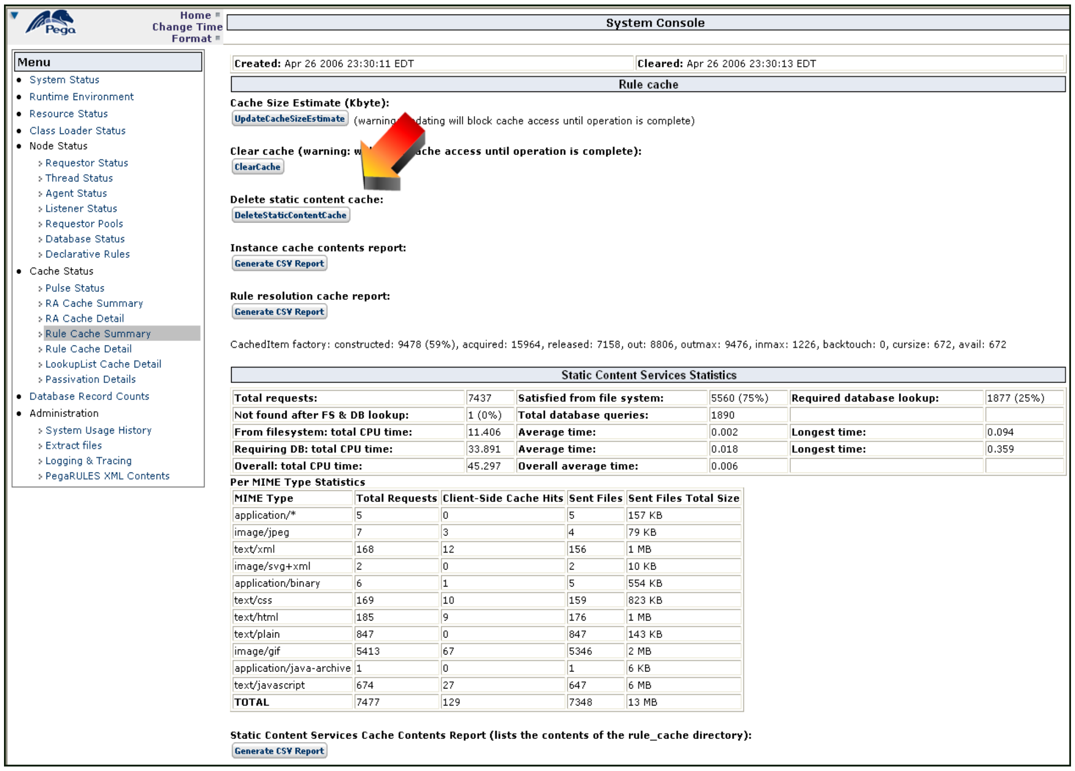
(NOTE: Just as the Rules in the personal RuleSet override the Rules in the main RuleSets, so the static content in the Personal subdirectory overrides the static files in the main subdirectory.)

If the requested file is not found on disk, the system will then go to the database and retrieve the information from there; the content will be returned to the user, and then the file is written to the disk in the appropriate subdirectory:

- If the Rule that is found is not checked out by the user, it is saved in the RuleSet List directory.
- If the Rule that is found is checked out (the RuleSet List of the Rule contains a "@"), it is saved in the user's Personal RuleSet List subdirectory in the appropriate RuleSet List directory.

If a user is changing the static information, when the item is checked out for editing, a copy is written to the user's personal RuleSet directory. After the changes are made, the Rule is checked in again. During the checkin process, the changes are written to the database, and then *all* instances of that Rule *throughout the rule_cache directory structure* are deleted. This step prevents stale data from being displayed to any user - the next time someone requests this static information, no cached file will be found, so the system will retrieve the updated information from the database.

From the System Console, the **Rule Cache Summary** page shows information about the Static Content caching.



Created: Apr 26 2006 23:30:11 EDT **Cleared:** Apr 26 2006 23:30:13 EDT

Rule cache

Cache Size Estimate (Kbyte):
(warning dating will block cache access until operation is complete)

Clear cache (warning: w ache access until operation is complete):

Delete static content cache:

Instance cache contents report:

Rule resolution cache report:

CachedItem factory: constructed: 9478 (59%), acquired: 15964, released: 7158, out: 8806, outmax: 9476, inmax: 1226, backtouch: 0, cursize: 672, avail: 672

Static Content Services Statistics

Total requests:	7437	**Satisfied from file system:**	5560 (75%)	**Required database lookup:**	1877 (25%)
Not found after FS & DB lookup:	1 (0%)	**Total database queries:**	1890		
From filesystem: total CPU time:	11.406	**Average time:**	0.002	**Longest time:**	0.094
Requiring DB: total CPU time:	33.891	**Average time:**	0.018	**Longest time:**	0.359
Overall: total CPU time:	45.297	**Overall average time:**	0.006		

Per MIME Type Statistics

MIME Type	Total Requests	Client-Side Cache Hits	Sent Files	Sent Files Total Size
application/*	5	0	5	157 KB
image/jpeg	7	3	4	79 KB
text/xml	168	12	156	1 MB
image/svg+xml	2	0	2	10 KB
application/binary	6	1	5	554 KB
text/css	169	10	159	823 KB
text/html	185	9	176	1 MB
text/plain	847	0	847	143 KB
image/gif	5413	67	5346	2 MB
application/java-archive	1	0	1	6 KB
text/javascript	674	27	647	6 MB
TOTAL	7477	129	7348	13 MB

Static Content Services Cache Contents Report (lists the contents of the rule_cache directory):

The statistics show how many of the static content images have been cached ("satisfied from file system"). The report at the bottom of this page shows where on disk the images have been stored.

As with the other caches, the static content cache should have to be cleared rarely, if ever. The only known time when this cache must be cleared is if the ImportExport Servlet is used to import rules into the system. The consequences of clearing the cache is that the system must check all the caches before going to the database, and then cache everything again, which will slow performance until the caches are rebuilt.

LookupList Cache

LookupLists contain information that displays in dropdown boxes in Process Commander. These are items (such as the SmartPrompts) which are built on a complex query that takes a long time to run (perhaps more than two seconds). Since this information is so expensive to produce, it is cached to improve performance.

The LookupList cache is stored similarly to the Static Content Cache, in the file system. Files are put in the *contextroot*/webwb/**llc** directory, and grouped by classname.

Example:

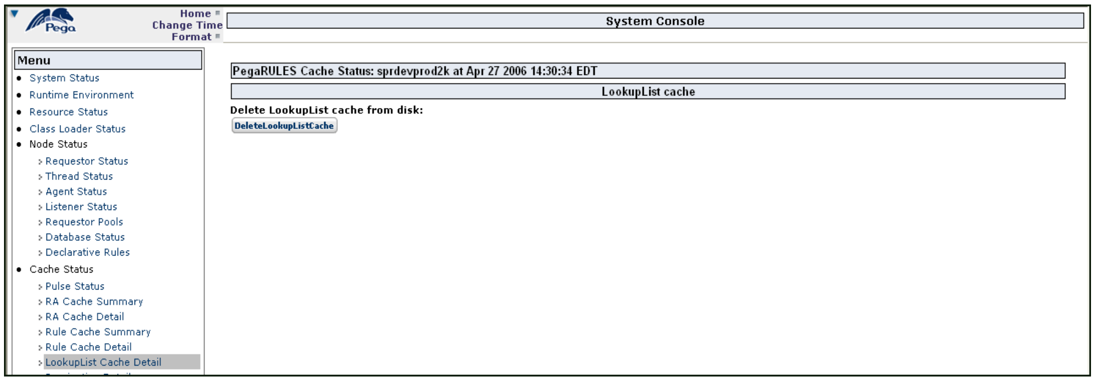
```
prweb/webwb/llc/Rule-Obj-Fieldvalue
prweb/webwb/llc/Rule-HTML-Harness
```

The first time a LookupList is requested, the system searches the directory structure for the requested rule, in appropriate class subdirectory under LLC. If the file is present, it will be returned.

If the file is not already cached, the system will load the file from the database, and also write the file to the LookupList cache on disk.

Again, the LookupList cache should have to be cleared rarely, if ever. If something that is done that modifies rules outside normal processing, such as importing rules using the PRImpExpServlet, then the cache must be cleared to pick up the new data. Also, if there was a problem with the system pulse, and data was not correctly propagated to the other nodes in the system, the LookupList Cache should be cleared (along with various other caches).

The consequences of clearing the cache is that the system must check the cache before going to the database, and then cache everything again, which will slow performance until the caches are rebuilt.



PegaRULES Cache Status: sprdevprod2k at Apr 27 2006 14:30:34 EDT

LookupList cache

Delete LookupList cache from disk: