

CacheBox

For ColdFusion

By Samuel Isaac Dealey

Contents

What is CacheBox?	3
CacheBox Has PEPP	3
Resources	3
Why CacheBox?	3
How Does It Work?	4
What Caching Engines Are Supported?	5
How Easy Is CacheBox?	5
A Namespace Primer	6
Can I use CacheBox with Shared Hosting?	7
What should I cache?	7
Getting Started	8
Installation	8
Hello Mr. Bond	8
Configuring the CacheBox Service (optional)	9
What's In the Box?	10
Storage Contexts	10
CacheBox Agents	11
What Can I Do With An agent?	11
Fetch and Store	11
Delete and Expire	12
How Do I Configure An Agent?	12
CacheBox Nanny	14
The CacheBox Service	15
Monitor and Reap (Scheduled Task)	15
Storage Types	15
Eviction Policies	15
BrAAiiNs!!!	16
Logging and Email	16
The Management Application	17
Home Page	17
Applications and Agents	18

Cluster	2
Options.....	20
Advanced Topics	22
Singletons.....	22
Session Cache.....	22
Custom Intelligence	23
Custom Storage Types	24
Custom Eviction Policies	27
Appendix: CacheBoxAgent Methods	29

Acknowledgements

Great software is never the product of a single individual. Many thanks go to [Rob Brooks-Bilson](#) whose excellent advanced caching techniques presentation for the Adobe Max conference provided invaluable insight into the development of this project. Many thanks also to [John Hirschi](#), [Shayne Sweeney](#) and [Tyler Smalley](#), who've done an excellent job integrating memcached with ColdFusion and on whose work the CacheBox integration of memcached depends. Additional thanks to Luis Majano and the [ColdBox](#) team, for the development of the ColdBox cache with its administration tools and hot-swappable eviction policies, and to Team [Mach-II](#), for development of the Mach-II framework's configurable, custom storage types. The ideas and implementation of both of these tools informed development of the CacheBox project. If you find CacheBox useful, make sure you thank these guys. If you're feeling generous, send them something from their wish-list, we're sure they'll appreciate it.

Contributors and Beta Testers

- [Steve Bryant](#)
- [Mike Henke](#)

What is CacheBox?

CacheBox is a hassle-free, advanced caching solution for any ColdFusion project. There are many other options for caching in ColdFusion. Each option has different syntax and a different set of features making them easy to apply but inconvenient to change. CacheBox is the only solution that provides all the features of the other caching options, plus a few of its own unique features. Of course you could try all the other caching options (that would take a long time and wouldn't be very rewarding) or you could install CacheBox and try them all within minutes, see everything in one place and switch them instantly with hot-swapping storage types. And thanks to its Agent / Service design, CacheBox is also portable, allowing you to create CacheBox-ready applications with no dependencies.

CacheBox Has PEPP

PORTABLE – Agent / Service design means no dependencies – use it everywhere!

EXTENSIBLE – Create custom storage types, eviction policies and intelligence to fit your needs.

PERSONAL – Cache is configured at run-time, using your server's available resources.

POWERFUL – Cluster synchronization, management application, hot-swappable storage types, etc.

Resources

For additional support or to contribute to the CacheBox project, join our discussion group at <http://groups.google.com/group/cfcachebox>.

Why CacheBox?

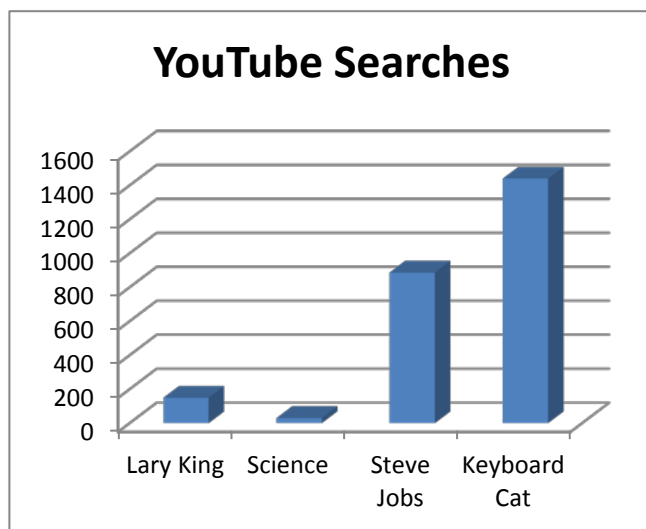
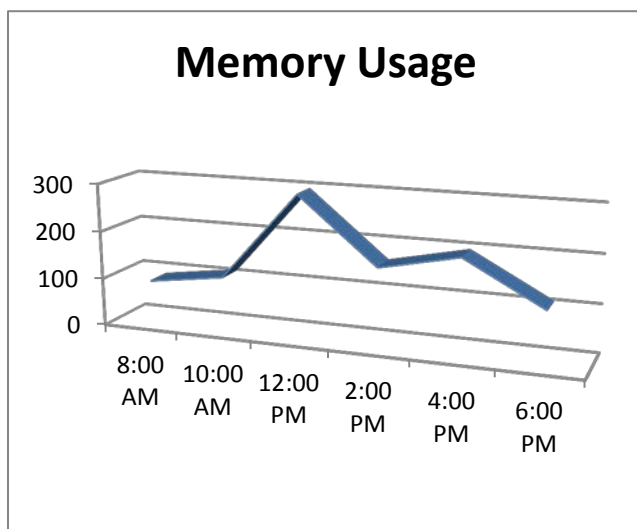
Although many of us have written a lot of caching code over the years, most of us don't really enjoy it. Caching features aren't something we look forward to working on. It's something we have to get out of the way to work on other more interesting things. My inspiration for CacheBox came when our lead developer on a job in Boston expressed frustration with a new ColdBox project. He wanted to install the ColdBox framework and have it "just work". Instead he seemed frustrated by a need to manually configure the framework's object caching. Getting it to behave the way he wanted didn't seem as easy as he'd hoped. I thought to myself, "Why can't it just work?" Why should we have to keep reinventing the wheel? Caching is not a new concept. In fact, it's pretty well-worn territory. So why after all this time hasn't someone created a single pluggable caching system that could work for all these frameworks and applications and save us the headache? That's the goal of CacheBox, to provide caching that "just works". If we've done our job right, it should be a hassle-free solution you can use on every project.

If you're looking at caching tools for ColdFusion, you probably want to improve performance of an application, or ensure it can handle some heavy traffic (or both). So you look into the different caching options that are available to you, both natively and third-party solutions like Memcached. The question is which one do you choose? Each solution is designed for a particular kind of caching problem and they all have different strengths and weaknesses... so which one do you choose for your project? Why not try them all? That's a lot of options, and testing each one can be a daunting and time-consuming task with little reward. Normally it would require rewriting a fair amount of code to test each one. How would you like a tool that lets you test and use any or all of them, without changing any of your code? CacheBox does this by hot-swapping different caching strategies at run-time and more...

How Does It Work?

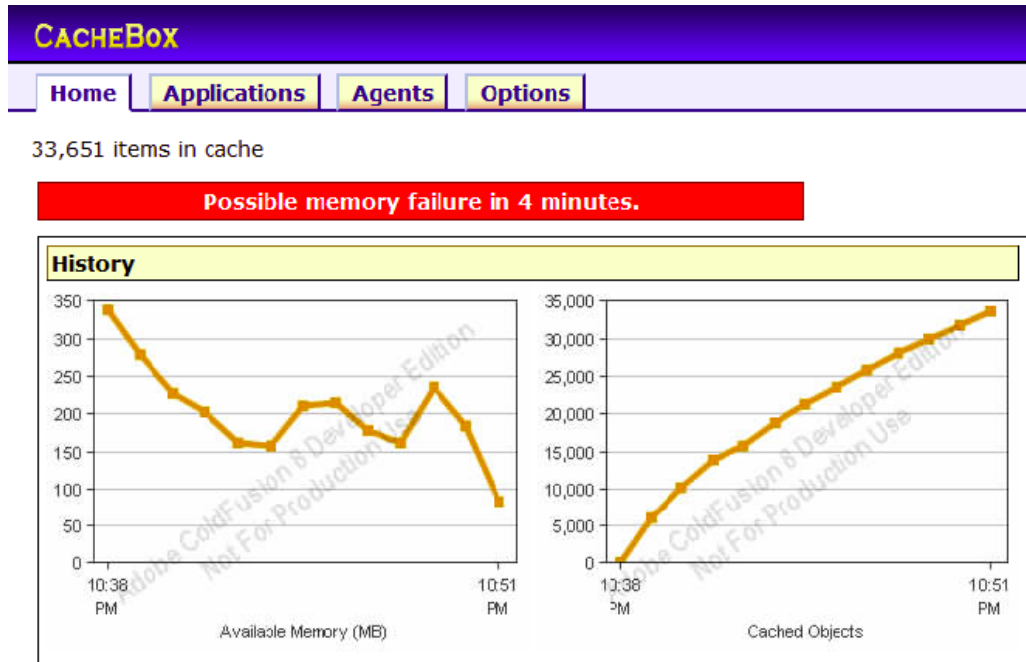
Work less and get more done. Are you skeptical? Let me ask you a few questions. How much do you know about your cache? Do you know what's in your cache right now? Do you know how long it's been there? Do you know how often people request it? Do you know what your server's resources look like throughout the course of a day? Do you know what a demand graph for your content would look like?

You probably answered no to a few of those questions, because the reality is that as a developer, you don't have time to monitor your server's resources and cache utilization all day. Even if you did, most of the available caching systems don't tell you much (if anything) about your cache, so although you might know that you have less than 100 ColdFusion queries cached, you have no clue which queries they are, how often they're requested, or how full your cached-query queue is. Because we don't really get to see into our caching systems and because our work time is a premium, we tend to build linear caching systems. For example, ColdFusion query caching is a linear system – it holds up to N queries in queue and when it reaches $N+1$, it removes the oldest one from the queue. But these linear strategies are inefficient because we're using them to address a non-linear problem. System resources like available memory and processor utilization are non-linear, and demand for content is also non-linear.

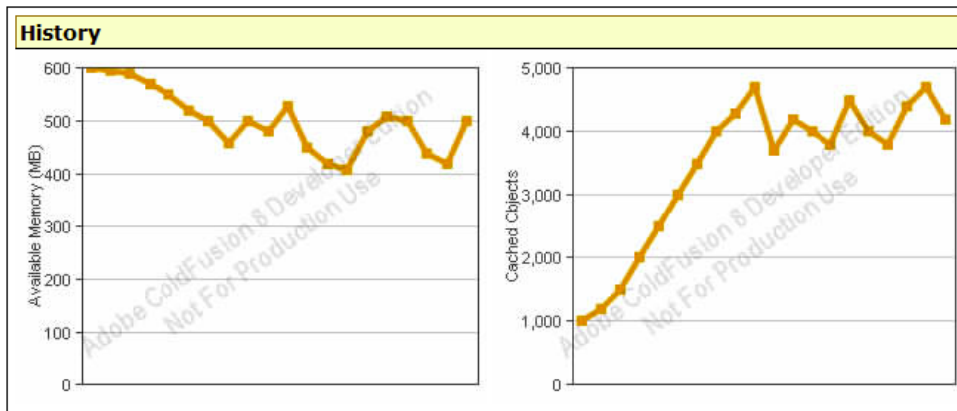


Although you can't constantly monitor your server's cache (even if you wanted to), there is someone who can. The server can monitor itself. By giving the server the intelligence to not only log but act on these non-linear events such as content popularity and traffic spikes, the server is able to automatically construct an optimal, non-linear solution to match the non-linear challenge of cache management.

So if the server detects a potential disaster like this:



It can analyze the problem, optimize the cache and avert the disaster on its own like this:



Because the server can monitor itself much more frequently and more consistently than you or I can, it can make incremental decisions about resources throughout the day to provide much more informed, accurate and efficient cache optimization.

What Caching Engines Are Supported?

By default, CacheBox includes support for caching in memory, to file, to database, Java SoftReferences, Railo's Cluster scope, the native ehCache instance added in ColdFusion 9 and Memcached (requires the [cfmemcached](#) project also). Additional storage methods are easily added.

How Easy Is CacheBox?

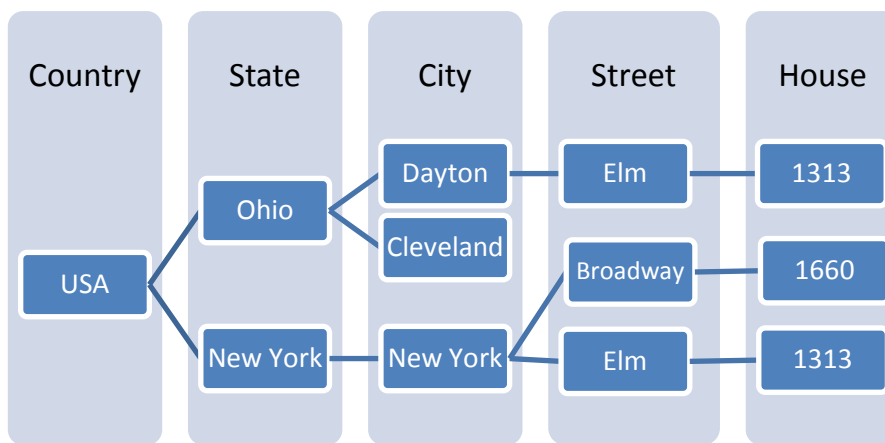
Are you wondering how many hours of studying you'll have to do to get CacheBox working? Don't worry. CacheBox is very easy to use. You will need to be able to create ColdFusion Components (CFC), and you need to have a basic understanding of namespaces and that's about it. Later on, you may want to learn more about in-process versus out-of-process caching because CacheBox supports both side-by-side however, you don't need to understand them right now.

A Namespace Primer

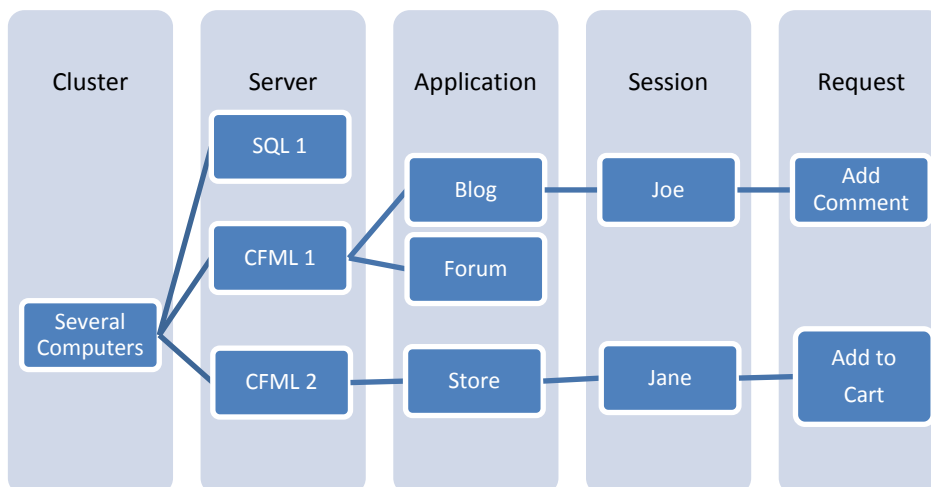
A namespace is a fancy way of saying “a way to identify things”. You can think of a namespace as the street you live on, where each house has its own number or “name”. So if you live at 1313 Elm Street, then your house is the only house that can be at 1313. In order for another house to be at 1313, it would have to be on another street (or in another “namespace”). Just like street names and house numbers help the post office find your house to deliver your mail, namespaces and names help the computer store and find your content.

Namespaces also tend to be nested. For example, in the namespace of the world, there is only one country named “United States of America” and only one country named “Canada”. Our country then has a namespace within it, so for example in the United States there is only one state named Ohio and only one state named New York. And each state has another namespace within it, so in Ohio there is only one city named Columbus and only one city named Cleveland.

So my individual house might be at 1313 Elm Street, Dayton, OH, USA. Of all the places in the world, there is only one house at that address, which is made up of five individual namespaces.



So any time you have a structure like any of the ColdFusion server’s built-in scopes (server, application, session, arguments, etc.), you have a namespace.



What would it look like if you tried to put two houses at the same spot in the middle of your street? It would probably look like a mess and it wouldn’t work very well. When you try to put two things at the same space within a namespace, for example, two houses at 1313 on the same street, that’s called a “namespace collision” or “namespace conflict”. You may also hear this called a “key collision” because the names and the things they represent are often called “key / value pairs”. Namespace collisions are important to avoid because colliding names will cause a wide variety of undesirable bugs in your application.

Can I use CacheBox with Shared Hosting?

Yes. CacheBox is designed to allow multiple copies per server without namespace collisions or other problems. The management application will only show content for your copy of CacheBox and because the management data is separate, optimization may not be as efficient. Although it's less than optimal, this should not prevent you from running CacheBox in a shared environment.

What should I cache?

That's a good question. It's also one that doesn't have a straightforward answer. My first caveat is this: you should manage anything you can through CacheBox. So if you're using query caching in ColdFusion now and you add CacheBox, the framework will work better once you switch all your cached queries to CacheBox instead of native query caching in ColdFusion. Although this does mean slightly more work at first when you convert your query caching, it will also mean better performance in the long-run. CacheBox can also serve as a substitute for using the `<CFCACHE>` tag for page fragments or potentially for entire pages. ColdFusion template caching on the other hand can't be managed through CacheBox, so this rule of thumb doesn't apply.

In his Adobe Max session on Advanced ColdFusion Caching Strategies in 2008, Rob Brooks-Bilson suggested a rule of thumb that you should cache as late as possible. So if you can cache an entire page, cache the entire page. If you need dynamic content in the middle of the page, then of course this isn't possible, but you should execute as much code as possible before caching the result. I think this advice might have been helpful to me several years ago when I developed a caching strategy for a complex ecommerce pricing system that ultimately turned out too slow to use in production. It's important to remember that the caching code itself will take time to execute and sheer speed isn't always its first goal. If I had been thinking about caching later, I might have developed a much better caching strategy for our ecommerce system.

On the other hand caching late often means caching content (usually html page fragments), and there are other cases in which caching database queries or objects (as in the case of the Transfer ORM) are also quite useful.

Lastly, caching is most effective for content that is viewed often and changes infrequently. Content that is changed too frequently needs to be updated in cache and may cause more problems than simply not caching it. Content that is viewed too infrequently will simply waste memory or other resources in cache.

Getting Started

This section describes the essential requirements to get started with CacheBox. After reading this section you should know how to install CacheBox, some optional configuration options, and how to store and retrieve data from the cache.

Installation

I'm something of a stickler when it comes to making things easy to use. That's why I've always insisted that the installation instructions for the onTap framework would have one and only one step: "unzip it". The same is true for CacheBox, with one minor exception. The CacheBox framework includes a management application that allows you to examine the content of your cache at run-time. While this transparency is good for you as a developer, it could also be a security risk, which is why it includes a password. After you've extracted your copy of CacheBox, you will need to view the management application in a browser and set the password. You should always make sure you set the password immediately after placing your CacheBox installation because anyone could set it and get into your cache until you do. If you need to reset the password, simply delete the file `/settings/password.cfm`.

In most cases your CacheBox management application is accessible at <http://127.0.0.1/cachebox/>. If your installation is not accessible at this URL, you will also need to configure the *pollingURL* in `config.cfc`. If you're like most people, you've just tried to open `/cachebox/config.cfc` and discovered that you can't because the file doesn't exist. Don't panic. ☺ Read "Configuring the CacheBox Service" following the "Hello Mr Bond" example.

Choose a password for the CacheBox Management Application.

Password:

Log In

Hello Mr. Bond

The only thing you need to start using the CacheBox is an agent (`/cachebox/cacheboxagent.cfc`). You will need to give your agent a unique name to make sure that the CacheBox service doesn't confuse your agent with other agents on the server. Once you've done that, then you can fetch and store content through your agent, using a unique identifier for each content item you want to store. Items are placed in cache using the *store()* method and retrieved from cache using the *fetch()* method. This hello world example shows how to use an agent named "James_Bond" to store intelligence using the code key "secret_x".

```
<!-- Call our agent, James Bond -->
<cfset agent = CreateObject( "component",
    "/cachebox/cacheboxagent" ).init( "James_Bond" ) />

<!-- Tell James the secret -->
<cfset agent.STORE( "secret_x" , "Hello World" ) />

<!-- Villains capture Bond and force him to talk! -->
<cfset message = agent.FETCH( "secret_x" ) />

<!-- Make sure the message is legitimate -->
<cfif message.status eq 0>

    <!-- The message was retrieved okay! -->
    <cfoutput>#message.content#</cfoutput>

<cfelse>

    <!-- The message was not found in cache -->
    ... Kill Mr. Bond! ...

</cfif>
```

Once you've executed this sample code from an application on your server, you can then view the CacheBox management application at <http://localhost/cachebox> (or wherever you placed your copy of the CacheBox framework). Within the

management application you can view all the agents that have registered with the CacheBox service, analyze their content, remove content from the cache (one at a time or for an entire agent or application), and configure various storage types and eviction policies for your agents.

Use this information wisely; the fate of the free world is in your hands!

Configuring the CacheBox Service (optional)

In most cases you shouldn't need to configure the CacheBox framework, it should just work. If you have an unusual use case in which you do need to tweak or modify the CacheBox framework, it should be easy to do that also.

If you're like most people, you might look at the Application.cfc and see that it creates a cacheboxservice.cfc and pulls a configuration object from there. The configuration object is defaultconfig.cfc. You might be tempted to edit the defaultconfig.cfc file; **DON'T**. Like the onTap framework, CacheBox is designed with future-proofing in mind. Any changes you might need to make to the framework should be possible via the config.cfc. All you have to do is create the file as a CFC that extends defaultconfig.cfc and anything you need to change about the framework can be overridden in config.cfc. This will protect you from most changes in future versions of the CacheBox core framework.

Your CFC should look like this:

```
<!-- /cachebox/config.cfc -->
<cfcomponent extends="defaultconfig">

    <!-- your custom code here -->

</cfcomponent>
```

Information about some of the configuration tweaks you might like to make can be found in the comments in defaultconfig.cfc.

What's In the Box?

The Hello Mr. Bond example in the previous section shows you just a small glimpse into the CacheBox. There you see how cache is stored and retrieved from the service through an agent, and you get to see the management application. What you haven't seen is the service object (cacheboxservice.cfc) that manages everything, configurable storage types and eviction policies and how all these things work together. This section gives you an in-depth look at what's in CacheBox and what you can do with it.

Storage Contexts

When you create a CacheBox Agent, you give it a name and a context (default is *APPLICATION*). Both of these properties identify the namespace within which the agent stores content, the same way your street address identifies your house. Supported contexts include *CLUSTER*, *SERVER* and *APPLICATION*. Agents in separate applications will share the same namespace and content if they have the same name and are on the same server (in the *SERVER* context) or in the same cluster (in the *CLUSTER* context). If you change the name or context of an agent during development, it effectively becomes a different agent and loses all of its configured settings and content.

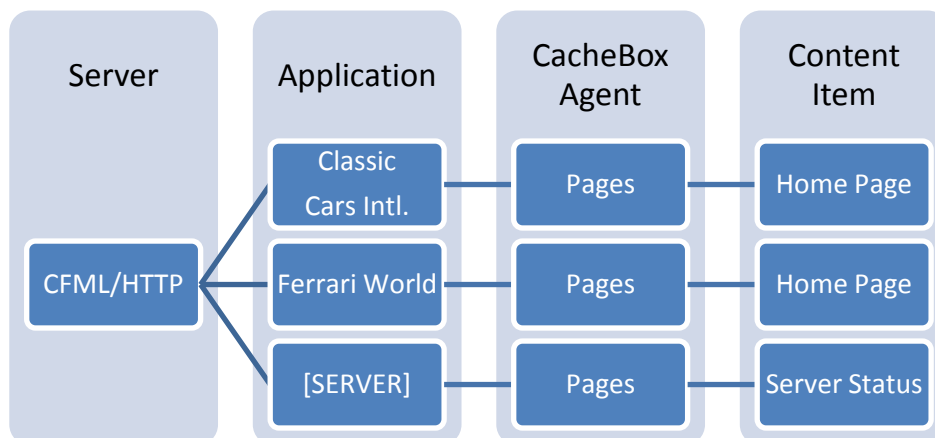
So for example if you create an agent in the *APPLICATION* context named "Pages" and then store some HTML with that agent using the name "Home_Page", you then have your home page content stored within the *APPLICATION* -> Pages namespace.

```
<!-- Create an APPLICATION Agent -->
<cfset agent = CreateObject("component",
    "/cachebox/cacheboxagent").init( AgentName = "Pages" , Context = "application" ) />

<!-- Store the Home Page with our agent -->
<cfset agent.STORE( "Home_Page" , HTML ) />
```

What this means is that content stored in the "Pages" agent is unique to the application where it was created. So if you have two clients, Classic Cars International (CCI) and Ferrari World (FW), both sites can be hosted on the same server and each can have the same "Pages" agent and different content stored for the "Home_Page" cache entry, because the content is unique to each application.

If you want to share content between these two applications, you can easily do that by creating the agent in the *SERVER* context instead. For example, you might want to cache your hosting company's "server_status" page in another "Pages" agent in the *SERVER* context and both sites would share the same cache for that page. If you want the same content to be shared between multiple servers, simply create the agent in the *CLUSTER* context. Each server in the cluster must have its own copy of CacheBox installed.



Although the CacheBox service provides many methods of storing cache, not all storage types can support all contexts. For example, the *CLUSTER* context allows multiple servers to share cached content between them, which requires an external

storage method (such as a database or Memcached server), which may not be available. When a requested context is not available, CacheBox gracefully degrades down to the next available context. So for example, your agent may request to store cache for the *CLUSTER* context, but if there is no available storage for that context, the service will use the *SERVER* context instead. If there is no CacheBox service at all, the agent will degrade down to the *APPLICATION* context and manage its own cache.¹

CacheBox Agents

In an ideal environment, the only thing your application knows about CacheBox is its agents (`cachebox/cacheboxagent.cfc`). Each application may have many agents for different kinds of content, and they will all connect to a single service object on the server. So for example full page content may be stored with a “Pages” agent, while an ORM like Transfer might cache data access objects (DAO) in a “Transfer” agent. The agents do two things, first they simplify your application code and secondly they restrict your application’s control of the cache. Both of these things mean less work for you. The reason the agent restricts control of the cache is to limit your application’s dependency on the CacheBox framework and to channel most cache management through the framework and management application. When the CacheBox service is installed, the agent behaves like an advisor, offering suggestions about how cache should be handled, rather than imposing rules or managing the cache directly. When the service is not installed or not available, the agent manages its own cache¹ while ignoring contexts and eviction policies.

Agents for your application should be created once and placed in the application scope, instead of creating them on each request. Although this should improve performance, the primary reason for keeping agents in the application scope is to allow them to manage cache if they are unable to connect to the CacheBox service because it is not installed or for some other reason.

What Can I Do With An agent?

- Add content to cache: `agent.store(cachename,content)`
- Retrieve content from cache: `agent.fetch(cachename)`
- Remove an individual piece of content from cache: `agent.delete(cachename)`
- Remove several related pieces of content from cache: `agent.expire(cachename & “%”)`
- Remove all the agent’s content from cache: `agent.reset()` or `agent.expire()`
- Check if the agent is connected to a CacheBox service: `agent.isConnected()`
- Miscellaneous: check the agent version number, access the service object, check the applied context and applied eviction policy (which may be different from the requested context or eviction policy), etc.

Fetch and Store

Most of the work you do with CacheBox will be done through the `store()` and `fetch()` methods, which either add content to the cache or return content from the cache respectively. Just as you need to provide a name for your agent, you also need to provide a name for your content. Other caching systems usually refer to this as a “key”; CacheBox refers to it as a “cache name”. The name is required to uniquely identify your content in the cache agent’s namespace; otherwise you won’t be able to get your content back out, just like the post office won’t be able to deliver mail to your house unless the address includes the house number on your street. The cache name only needs to be unique within the agent, so if your agent is named “pages”, it’s okay for the cache name to be a `page_id` variable, it doesn’t need to be “page_#page_id#”.

Both the `store()` and `fetch()` methods return a structure with two keys, a status key and a content key. The status key is a numeric value that may indicate a failure. A status of zero (0) indicates that the fetch or store operation worked perfectly. A status of one (1) indicates that the content was not found in storage. Mostly you need to know that a status of anything other than zero returned from a `fetch()` operation indicates that the content is not yet cached and needs to be generated. If you are storing singleton objects, skip to the [Advanced Topics](#) section at the end of the documentation. If you don’t know what a singleton is then you’re probably not storing singletons.

¹ It’s important to note that your CacheBox agent must be stored in the application scope to manage its own cache if the CacheBox service is unavailable.

Delete and Expire

The agent component provides two methods of removing content from cache. These are with the methods *delete()* and *expire()*. In most cases it shouldn't matter which you choose. Both methods allow you to remove multiple content items by using the percent symbol (%) as a wildcard character. So to remove all content with a cache name that begins with "product_5_", you could use *agent.delete("product_5_%")* or *agent.expire("product_5_%")*. In both cases, a *fetch()* operation immediately following this will not find any content for "product_5_X" (where X is anything), so they're functionally the same. The difference between these methods internally is that the *delete()* method removes the content immediately, while the *expire()* method only marks the content for later deletion. The importance of this is that while the *expire()* method is likely to execute faster than the *delete()* method, it doesn't immediately free up the server's memory. Instead the content stays in memory until the next reap-cycle. By default a reap cycle occurs every minute, so content won't stay in memory for very long. So if you receive complaints from users about a page being slow and that page is deleting a lot of cache content, switching to the *expire()* method may resolve that issue.

How Do I Configure An Agent?

A CacheBox agent is configured through its *init()* method, which provides several arguments for configuration. The only argument required when you create your agent is the AgentName, which uniquely identifies your agent and eliminates confusion with other agents, so it's important to give your agents descriptive names. Below is the full list of *init* arguments for an agent.

Argument	Type	Required	Default	Description
AgentName	String	Yes	N/A	Uniquely identifies your agent
Context	String	No	Application	CLUSTER, SERVER or APPLICATION – see below
Evict	String	No	AUTO	The suggested eviction policy for your agent – ignored if the CacheBox service is unavailable – see below
ReapListener	Object	No	N/A	A component to receive notification when objects cached by your agent are removed from the service – see below
CacheService	Object	No	N/A	Not Recommended – see below
ApplicationName	String	No	*	Not Recommended – defaults to the name of your ColdFusion application, set in your Application.cfc

Example: this code creates an agent that stores HTML fragments for an application that will stay in cache until they've been unused for at least 60 minutes. (The eviction policy "auto" is recommended for most non-singleton agents.)

```
<cfset Application.PageFragments =
  CreateObject( "component" , "cacheboxagent" ).init(
    AgentName = "PageFragments",
    Evict = "IDLE:60"
  ) />
```

Context: The agent's context describes the desired context for the cached content. Supported values are *CLUSTER*, *SERVER* and *APPLICATION*. When creating an agent, you should configure it with the largest possible context you think that agent might need in the future. For example, if you are a hosting company and are caching content pages for client sites, then the *APPLICATION* context is appropriate because each client will have their own content pages. On the other hand if you're caching the display of videos that have been uploaded by users (like YouTube for example), then you should use the *CLUSTER* context because the application may eventually outgrow a single server and it's better to be prepared by overestimating the need.

Unlike storage types and eviction policies (which are both hot-swappable), the context of your agents can NOT be changed after the agent is created. As previously mentioned, the service will gracefully degrade if you request a context that isn't available, so the applied context of your agent may not be the same as the context you request. You can get the requested

context of an agent with the *getContext()* method and you can compare this to the context in use with the *getAppliedContext()* method.

Evict: This argument allows you to declare an eviction policy for your agent. Eviction policies are ignored if the CacheBox service is not available. The default eviction policy is *AUTO*, meaning that the CacheBox service is free to assign an eviction policy it decides is optimal. Eviction policies of *FRESH* or *PERF* are auto-optimizing policies, with an emphasis on freshness of content or speed of delivery respectively. If you have an agent that needs permanent retention of its content, specify an eviction policy of *NONE*.

The remaining eviction policies include either a time limit or a cache-size limit. For example the *FIFO* policy means “first in first out” like the ColdFusion cached queries queue. To have a *FIFO* eviction policy, you have to tell the agent how large the queue is allowed to grow before it starts evicting content. If an eviction policy has a time limit or a cache-size limit, indicate the limit by appending a colon and number to the name of the eviction policy. For example, *FIFO:100* will store up to one-hundred items for the agent, while *AGE:20* will store the content for up to twenty minutes. You can get the requested eviction policy of an agent with the *getEvictPolicy()* method. If the agent is not connected to a CacheBox service, the *getAppliedEvictPolicy()* method will return a value of “*NONE*” indicating that the eviction policy is being ignored.²

ReapListener: This argument allows you to declare a component to receive notification when items are removed from cache. This object must contain one method named “ReapCache”, having two arguments named “CacheName” and “Content”. Using a reap listener adds dependency to the system and as such should be avoided if possible. If it is not possible to avoid using a reap listener, this example shows how to write your listener:

```
<cfcomponent displayname="ReapListener" output="false">
    <cffunction name="ReapCache" access="public" output="false">
        <cfargument name="CacheName" type="string" required="true" />
        <cfargument name="Content" type="any" required="true" />

        <!--- Perform any necessary clean-up with the content --->

    </cffunction>
</cfcomponent>
```

CacheService: This argument is provided primarily for sticklers who don’t like the way CacheBox eliminates dependencies. It allows you to declare the CacheBox service to use when you create the agent and may also be useful for testing purposes.

ApplicationName: This argument indicates the name of the application to which this agent belongs. This is primarily for agents in the application context. In general, you should omit this argument. If you have some specific reason to need to fetch cache content from an alternate application, it might be useful to declare an agent into the other application, but it’s not recommended.

² The value “*NONE*” may also be returned from *getAppliedEvictPolicy()* if it’s been set as the eviction policy for the agent in the management application. The agent’s *isConnected()* method can be used to distinguish between these two cases.

CacheBox Nanny

The CacheBox service is very efficient at handling groups of cache collected for an agent. The trade for this design is that it's not as good at handling custom eviction policies for individual content items. So for example, your "Pages" agent may have an eviction policy of age:20 or age:30, but you can't have an eviction policy of age:60 for just the "HomePage" item and age:20 for all the other pages. The nanny object is designed to give you more control over content expiration at the client side by wrapping itself around an agent and providing additional content expiration features that mirror the *CachedWithin* and *CachedAfter* attributes in the ColdFusion *CFQUERY* and *CFCACHE* tags. The nanny also allows you to specify necessary eviction for content that must be periodically refreshed, even if the CacheBox service is unavailable. By comparison, an unmodified CacheBoxAgent component will simply ignore the requested eviction policy if the service is unavailable.

Here is an example of a nanny in use.

```
<!--- Create an APPLICATION Agent --->
<cfset agent = CreateObject( "component" , "cacheboxagent" ).init( "testagent" ) />

<!--- Create a Nanny to enhance the agent --->
<cfset agent = CreateObject( "component" , "cacheboxnanny" ).init( agent ) />

<!--- Store a message for up to ten seconds --->
<cfset span = CreateTimeSpan( 0, 0, 0, 10 ) />
<cfset temp = agent.store( "hw" , "Hello World" , span ) />

<!--- lets monitor the storage --->
<cfloop condition="temp.status eq 0">
    <!--- wait for three seconds --->
    <cfset sleep( 3000 ) />

    <!---
    Check the content and display it
    We should see "Hello World" 3 times
    The 4th dump should display empty content
    --->

    <cfset temp = agent.fetch( "hw" ) />
    <cfdump var="#temp#" />
    <cfflush />
</cfloop>
```

The CacheBox Service

CacheBox is designed somewhat like cable TV. When you rent a new apartment, it is usually “cable ready” when you move in, but the cable TV service is not available. You can of course still watch TV, you just don’t get access to all the cool cable stations like Discovery and SyFy unless you subscribe to the cable service. A CacheBox agent functions like a television set, giving you the ability to perform simple content caching. Once you install the CacheBox service then the agents stop managing their own cache and hand it off instead to the service object (`cachebox/cacheboxservice.cfc`), giving you access to all its advanced caching features. The service creates the config object, which in turn manages agents, storage types and eviction policies. Because it’s impossible to know which application will be accessed first when a server is started, the agents will start the service if it hasn’t been started already. This gives you the ability to eliminate the CacheBox framework as a dependency in your application. All you need to do is copy the `cacheboxagent.cfc` into your application and create objects of type “*myapp.cacheboxagent*” instead of creating objects of type “*cachebox.cacheboxagent*”.

Monitor and Reap (Scheduled Task)

The primary concern of the service is to manage and optimize the cache, to allow for optimal performance of your applications. This means keeping frequently used content in cache to speed up delivery of pages, but it also means removing stale content from the cache to preserve the server’s available memory and prevent out-of-memory errors. By default, the `config.cfc` creates a scheduled task to remove (reap) stale cache when the framework starts. The same scheduled task also updates the history statistics the framework uses to optimize the cache and that are displayed in the management application. Using a scheduled task for optimization and content expiration means that your site visitors are never waiting for long-running cache-management tasks to complete before they receive their content (and your very limited *CFTHREAD* pool is never being used for cache management either). If for some reason you don’t want to use the standard *CFSCCHEDULE* method (e.g. you don’t want to use a scheduled task in a shared hosting environment where you don’t have access to the ColdFusion administrator), you can override the `updateMonitoringTask()` method in your `config.cfc`.

Storage Types

The CacheBox service comes with a variety of storage types, some of which require configuration (*FILE*, *DATABASE* and *MEMCACHED*). Others work with no configuration (*DEFAULT*, *SOFT*, *CLUSTER* and *NONE*). Setting an agent’s storage type to *NONE* will disable caching for that agent (statistics will still be reflected in the management application, but content will not be stored). If you want to store cache in some way that’s not currently supported by CacheBox, you can extend the application with custom storage types. Some storage types require additional configuration, which is found in an XML file in `settings/storage`. If you’re familiar with XML, you might be tempted to configure storage types manually by editing the XML files; **DON’T**. Storage types are loaded once when the service starts and manually edited XML files will not be reloaded. The management application will update and reload the configuration. For more on custom storage types, see the [Advanced Topics](#) chapter.

Eviction Policies

In addition to storage types, the CacheBox service comes with a variety of eviction policies for determining when content is stale and should be removed from cache. Like storage types, the CacheBox service can be extended with custom eviction policies by creating an eviction policy component in the eviction directory that extends the `abstractpolicy` component. This component needs a `getExpiredContent()` method to inform the Config object which items to remove. For more information about creating custom eviction policies, see the [Advanced Topics](#) chapter.

BrAAiiNs!!!

The soul of the CacheBox project is its ability to automatically configure caching strategies for its agents – those that specify an eviction policy of auto (default), fresh or perf. This is handled by an intelligence agent (intelligence.cfc), which loads a collection of brain objects to analyze the agent statistics and suggest possible caching strategies. Anyone can add new brains to the service. All you have to do is create a CFC that extends the abstract.cfc in the settings/intelligence directory. Once the new brain is installed, it will be automatically loaded the next time cache strategy recommendations are requested by the service. If you want more control over the way the intelligence agent creates recommendations, you can place a modified version of intelligence.cfc (or better yet, a component that extends it) in the settings directory. For more information about custom brains, see the [Advanced Topics](#) chapter.

Logging and Email

Email alerts and logging features aren't enabled by default when you install CacheBox on your server.

If you want to receive email alerts when the service predicts a possible memory failure, you can enable this by copying the file email.cfc into the settings directory and editing the CFMAIL tag attributes as necessary. Be aware that the default polling interval is one minute, so you may find that you receive one email every minute for the first 10-20 minutes after the server or the ColdFusion service is restarted as the cache is initially loaded and potential memory failure is repeatedly predicted. Future versions of CacheBox may not be as vocal about early predictions. Any changes made to this component will be automatically applied the next time the service sends email.

Logging of cache events may be similarly enabled by copying the file log.cfc also into the settings directory. There are a variety of instance variables in the logging component for setting:

- Frequency of history logs (default is 20 minutes)
- Type of agent configuration events to log (AutoConfig, Manual and Cluster Synchronization)
- Location of log files
- Format and extension of log files (default is tab-delimited, quoted identifiers in .log.cfm files)
- Log Rotation – by date and/or size and the number of previous logs to retain

If additional logging features are desired, such as logging to a database instead of a text file, the fact that this component is a CFC should make adding those features relatively easy. Any changes made to the log.cfc will be automatically applied the next time a log event executes.

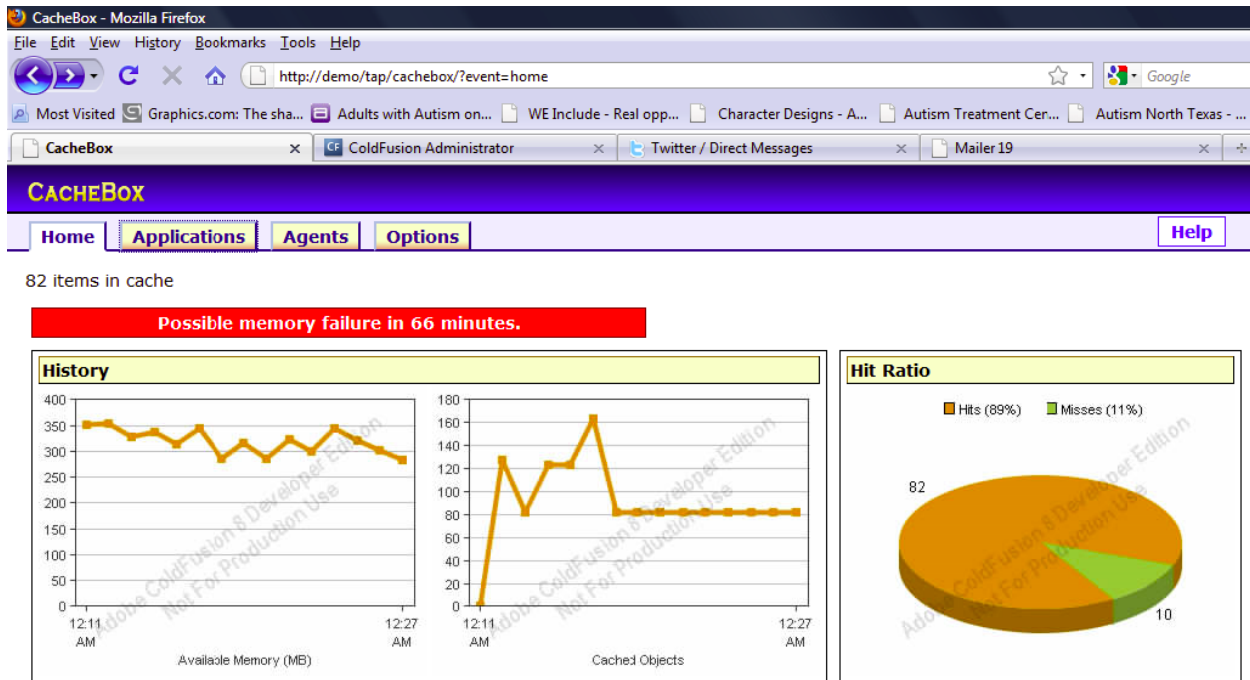
Unlike other logs, errors are logged in the logs/errors/ directory as a dump of the entire error structure, one per file, named by date and time. This directory may fill up quickly if you modify one of the other logging methods and introduce a syntax error.

The Management Application

The purpose of the CacheBox management application is to give you better insight into and control over your cache. Yes, in an ideal world the CacheBox service would anticipate and handle any and all caching needs. In the real world sometimes when you build a better mousetrap, someone else builds a better mouse. So although we hope that the service will handle most applications with little or no configuration, the management application provides the extra tools you might need if you happen to have unusual requirements that call for manual configuration. Most of the management application should be fairly self-explanatory.

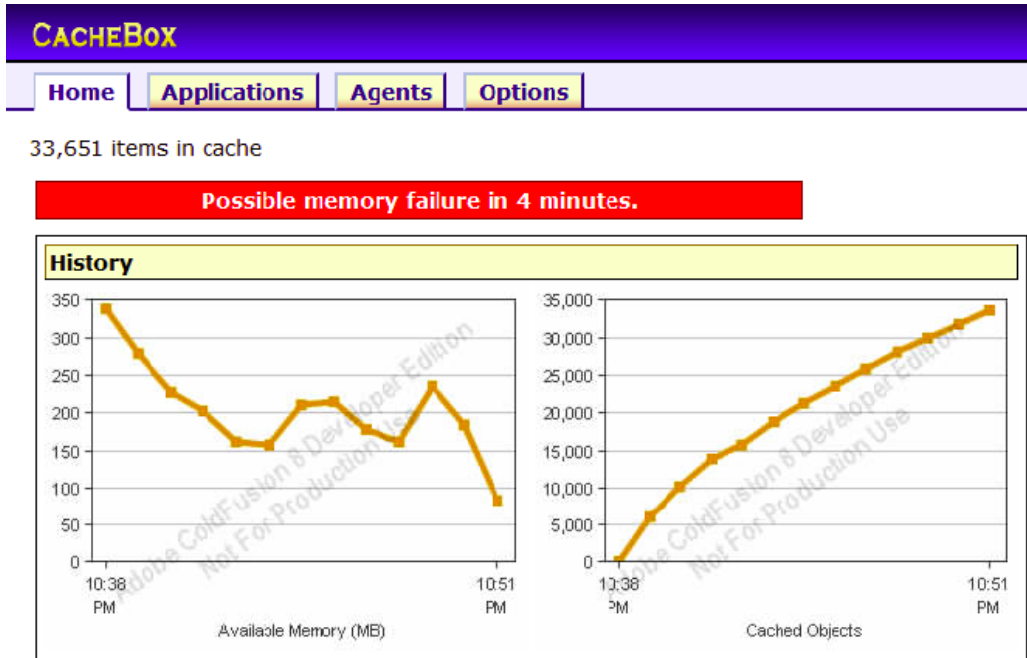
Home Page

Upon opening the application home page, you'll see tabs at the top, a help link in the upper-right corner (which currently opens this PDF), a history graph showing available memory and cache usage for the past 20 minutes and a pie chart showing the hit ratio. The charts on this page give you a very brief snapshot of current cache performance. You want the hit ratio to be high, but not necessarily 99%. It's okay for the hit ratio to be 70% or even lower if the service is evicting stale content, because evicting the stale content helps to preserve available memory.



Because the service has access to all this data each time the service is polled (via a scheduled task), the service will know about potential problems before you do. It begins optimizing the cache agents and assigning new eviction policies once it predicts a possible memory failure within two hours. If after optimization the projection continues to drop to below one hour, the red warning message will appear on the CacheBox home page as you see above. (You can modify these times in config.cfc.) This warning may or may not be something you need to jump to address. In the example above, the service doesn't have quite a full twenty minutes of data for the prediction because it's only been running for 15 minutes. Less data means a less accurate prediction, however, even with a relatively accurate prediction, garbage collection and fluctuations in demand for the content can change the outcome. The warning is designed to make you pay attention, not necessarily leap to action.

If you do have a truly imminent memory failure, the obvious sign will be that the lines of the history graphs will point downward in the middle, as if they were creating a down-arrow to indicate “your system is going down”. Here’s an example of what that looks like:

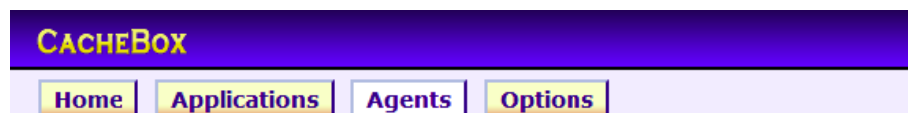


If you open the management application and you see something like this, you should definitely start doing something about it right away.



Applications and Agents

These tabs allow you to see lists of cache statistics for the agents that have registered with the service or the applications that have content stored on their behalf.

The blue row in this table shows the name of the application to which the “testagent” agent is assigned by its context. The blue curved arrow icon flushes the agent, removing all items stored on its behalf from the cache. The magnifying glass and the name of the agent are linked to a detail page to view and optionally manually configure it.

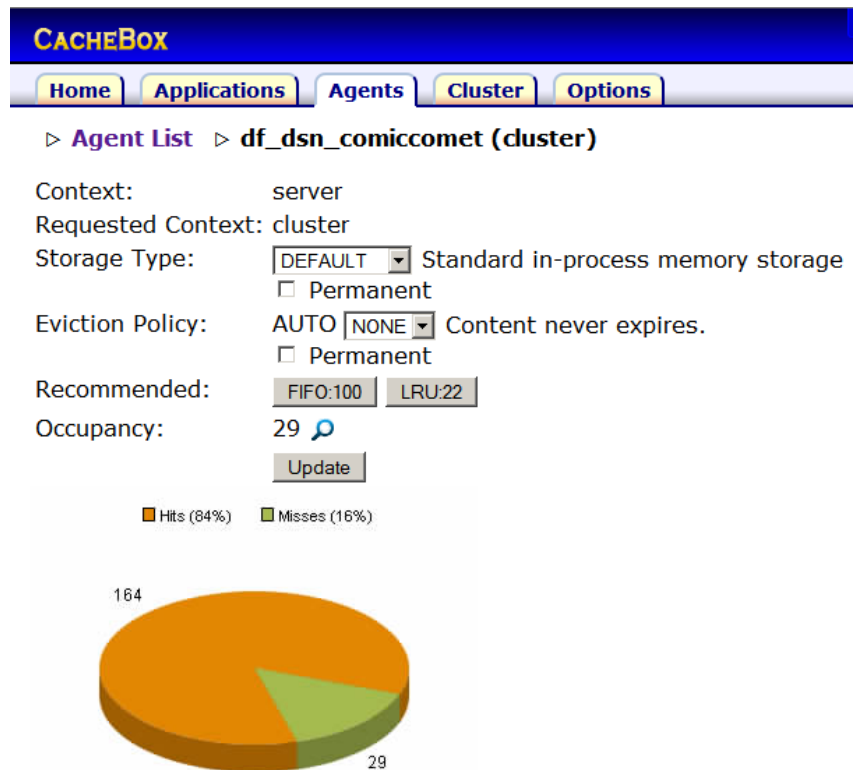


Application

Agent	Occupancy	Oldest	Last Hit	Storage	Eviction
c_apache_htdocs_tap_cachebox					
  testagent	82	1:01 AM	0 minutes	none	age (1)

Agent Detail: This form shows stats and configuration for the agent you selected and allows you to manually configure it. Only storage types that support the agent's applied context will appear in this form. If you select the permanent checkbox below either the storage type or the eviction policy, that property will be stored in an XML file (settings/agents.xml.cfm) and used as the default value if the server is restarted. If you're familiar with XML, you might be tempted to manually edit this XML file; **DON'T**. Use this form, it will update the XML file and reload it into the application for you.

The Recommended buttons will populate the eviction policy form with the suggested eviction policies, but will not apply them – press the Update button at the bottom to apply the selected policy.



The magnifying glass next to the occupancy count brings you to a content list for the selected agent.

Agent Content: If the storage data contains records for items that have been requested but not stored, you'll see a link below the frequency legend to toggle the display of miss-counters. These records may happen occasionally even when the application is working correctly, due to malformed URLs provided by site visitors. To the left of each item in the list you see a red X to remove the item from cache. Clicking the name of the item allows you to view the content for that record. If the content is a query, structure, array, object, binary or XML, this will display a dump of the object. If the content is text, it will display a form that allows

you to modify the cache at run-time. If you are modifying cache at run-time, chances are there is another problem you should address in your application, but this will allow you to fix the symptom while you work on the problem.

CACHEBOX

Home Applications Agents Cluster Options

Agent List tap_href (tapogee) Content

Frequency:

☒ Hide Misses

	Name	Stored	Last Hit	Hits	Misses	Frequency
001	tap_href		3 hrs	0	12	
002	<input checked="" type="checkbox"/> c	8 hrs	8 hrs	1	0	8 hrs
003	<input checked="" type="checkbox"/> d	8 hrs	8 hrs	1	0	8 hrs
004	<input checked="" type="checkbox"/> docs	8 hrs	15 min	32	0	15 min
005	http://on.tapogee.com		3 hrs	0	2	
006	<input checked="" type="checkbox"/> img	8 hrs	8 hrs	1	0	8 hrs
007	<input checked="" type="checkbox"/> inc	8 hrs	8 hrs	1	0	8 hrs
008	<input checked="" type="checkbox"/> p	8 hrs	8 hrs	1	0	8 hrs
009	<input checked="" type="checkbox"/> plugins	8 hrs	8 hrs	1	0	8 hrs
010	<input checked="" type="checkbox"/> r	8 hrs	8 hrs	1	0	8 hrs
011	<input checked="" type="checkbox"/> style	8 hrs	15 min	53	0	9 min
012	<input checked="" type="checkbox"/> t	8 hrs	8 hrs	1	0	8 hrs

Cluster

The cluster tab provides a way to keep the CacheBox agents and storage types in your cluster in-sync. At the top of the page is a list of the currently declared servers in your cluster, showing their status and an ID for the server. The top-most item in the list is the server you're viewing currently (usually labeled "localhost"), highlighted to remind you that you're currently viewing it. Selecting the link on the name of a server will take you to the management application for that server (note that you will not be allowed to view the management application unless you have created a config.cfc and edited the `applyRequestSecurity()` method). If there are no additional servers in the list, you can add them by entering the IP address or full URL to the CacheBox management application in the form below. Once entered, the servers will initially mistrust each-other and display a form to allow you to establish trust. Select the link to the alternate server, copy the ServerID from the top, press the back-button on your browser, paste the ServerID into the form and press the "Get Trust" button. Do this for each server in your cluster. If there are any servers left out, the system may not sync configuration across all the servers when you update a storage type or an agent.

CACHEBOX

[Home](#) | [Applications](#) | [Agents](#) | [Cluster](#) | [Options](#)

Server	Status	ServerID
demo	OK	C0252791-DFEE-3B0C-A6FC43EFE929A026
192.168.1.101/tap/cachebox	UNAVAILABLE	
192.168.1.100/tap/cachebox	OK	C0252791-DFEE-3B0C-A6FC43EFE929A026

Enter one server per line as an IP Address or full URL.

192.168.1.101/tap/cachebox
192.168.1.100/tap/cachebox

Options

The last tab provides two links to flush cache for the server or cluster contexts, as well as lists of the installed storage types and eviction policies.

Storage types that require configuration will appear with a gear icon to the left of the type name. Selecting the linked name of the storage type will display the form to configure that type – each type will have a different configuration form. Not all of the installed storage types may be available. The ready column will indicate which storage types are available. Unavailable storage types will not appear in the agent configuration form, however, you may be able to make the storage type available if it requires configuration that has not been provided. The *FILE* storage type for example requires that you provide a directory name in which to store content before it is ready. The *CLUSTER* storage type on the other hand depends on Railo's cluster scope and so there is not configuration that can make it ready; either the cluster scope is available or it's not.

CACHEBOX

[Home](#) | [Applications](#) | [Agents](#) | [Cluster](#) | [Options](#)

[Reset Cluster Cache](#) [Reset Server Cache](#) [Optimize Now](#)

Storage Type	Context	Ready	Description
CLUSTER	CLUSTER	No	In-memory storage using the cluster scope (Railo)
DATABASE	CLUSTER	Yes	Saves cached content to a database table
DEFAULT	SERVER	Yes	Standard in-process memory storage
FILE	SERVER	Yes	Saves cached content to a physical file
MEMCACHED	CLUSTER	No	External storage using a memcached server with cfmemcached
NATIVE	SERVER	Yes	Uses the native ehCache instance in ColdFusion 9
NONE	CLUSTER	Yes	Disables caching for a specific agent
SOFT	SERVER	Yes	Java soft-references allow the JVM to manage content expiration

Eviction Policy	Description
AGE	Expires N minutes after creation
FIFO	First In First Out: Expires after N records in cache
IDLE	Expires after N minutes unused
LFU	Least Frequently Used: Expires after N records in cache
LRU	Least Recently Used: Expires after N records in cache
NONE	Content never expires.

Selecting the Optimize Now link at the top of the options page will bring you to the Agent Recommendations page. Here you'll get a list of suggested eviction policies for all your registered agents and can select and apply the suggestions you like. The service will only suggest a given eviction policy if that policy isn't already the applied policy on an agent, and only one policy may be chosen for each agent.

These recommendations are made by the brains in the settings/intelligence directory. This is the same process performed when the service auto-configures itself when a memory-failure event is predicted. The difference is that during auto-configuration the service stops at the first available recommendation for each agent because it can only apply one at a time. Here, the service offers you all the suggestions the brains have to offer and lets you pick.

CACHEBOX

[Home](#)
[Applications](#)
[Agents](#)
[Cluster](#)
[Options](#)

> [Agent List](#) > **Recommendations**

Application

Agent	Storage Type	Evict Policy	Recommendation
Comiccomet			
membersontap	default	none	<input checked="" type="checkbox"/> fifo (100)
moderationrules	default	auto: none	no content
mot_members	default	auto: none	no content
mot_policy	default	auto: none	<input checked="" type="checkbox"/> fifo (100)
ontopic	default	auto: none	<input checked="" type="checkbox"/> fifo (100)
tap_filecache	default	idle (5)	no content
tap_href	default	none	<input checked="" type="checkbox"/> fifo (100)
tap_html	default	auto: none	<input checked="" type="checkbox"/> fifo (100)
tap_ioc	default	none	<input checked="" type="checkbox"/> fifo (100)
tap_path	default	none	<input checked="" type="checkbox"/> fifo (100)
Tapogee			
tap_filecache	default	idle (5)	no content
tap_href	default	fifo (100)	
tap_images	default	auto: fifo (100)	
tap_ioc	default	fifo (100)	
tap_path	default	fifo (100)	

Server

Agent	Storage Type	Evict Policy	Recommendation
df_dsn_comiccomet	default	auto: none	<input checked="" type="checkbox"/> fifo (100) <input type="checkbox"/> lru (22)
ontopic_brand	default	idle (60)	<input checked="" type="checkbox"/> fifo (100)
ontopic_comments	default	auto: none	no content
fonthauspagecache	default	age (1440)	no content

☐ Make Permanent

Apply Recommendations

Advanced Topics

Singletons

In order for any caching service to provide an answer for the management of singletons (assuming you want to use the caching system for singleton objects), it must provide a means of addressing the “dog pile” condition. The dog pile is a race condition in which two separate requests (two separate site visitors) attempt to fetch a cached object at the same time, both receive a “miss” from the cache system and proceed to create the object. The reason why this is important is because a singleton object should be unique within an application. So for example if your application has a singleton called the FlightDynamicsOfficer (FIDO) that’s responsible for ensuring safe and successful shuttle launches, you need to be certain that there is one and only one FIDO at run-time. This is the reason why the *store()* method of the CacheBoxAgent CFC returns the same structure returned by the *fetch()* method. The content value returned from the *store()* operation will always be the first object stored and any other subsequently created objects can be thrown away.

Here is an example of singleton creation using CacheBox.

```
FIDO = agent.FETCH( "FIDO" );

if (FIDO.status) {
    // a non-zero status is a cache miss, create the object
    objFIDO = CreateObject( "component" , "FIDO" ).init();

    // now we need to store the object, but...
    // we also need to be sure we have a singleton
    FIDO = agent.STORE( "FIDO" , objFIDO );
}

// whether we stored or fetched the object,
// it's in the content variable
return FIDO.content;
```

If a request did encounter the race condition and the service attempted to store the singleton twice, then the status value returned from the *store()* operation will be two (2), indicating a “failure”. This only applies to the storage of objects. If your content is a string like an HTML fragment, then the service will simply overwrite the stored content when new content is provided for the cache name. If for some reason you needed to overwrite an object in cache, you would need to first *delete()* or *expire()* the previous object, then store the new object and check the status from the store operation to ensure it stored correctly.

Session Cache

The CacheBox service isn’t designed in particular to handle content that might be stored in the session scope. There is however an easy way to store session content in CacheBox. Create an agent in the *APPLICATION* context for session storage. Any time you add content for a session, prefix the CacheName with the user’s SessionID. Then when the session expires, delete content from the agent using the wildcard % character to remove all content for the expired session.

```
<cfscript>
    // Create Session Agent
    Agent = CreateObject( "component", "cacheboxagent" ).init( "SessionMan" );

    // Store a session value
    Agent.STORE( session.sessionid & ".MyThing" );

    // Remove all content for an expired session
    Agent.DELETE( session.sessionid & ".%" );
</cfscript>
```

Custom Intelligence

The soul of the CacheBox project is its ability to automatically configure caching strategies for its agents – those that specify an eviction policy of auto (default), fresh or perf. This is handled by an intelligence agent (intelligence.cfc), which loads a collection of brain objects to analyze the agent statistics and suggest possible caching strategies. Anyone can add new brains to the service. All you have to do is create a CFC that extends the abstract.cfc in the settings/intelligence directory. Once the new brain is installed, it will be automatically loaded the next time cache strategy recommendations are requested by the service. This can be from the Management Application or it can be from the auto-configuration process that occurs if a memory failure is predicted during the monitoring cycle. The difference between these two events is that the Management Application will offer you all available strategy suggestions, while the auto-configuration process will stop on the first suggestion made for each agent, because it can only apply one suggestion. If you want more control over the way the intelligence agent creates recommendations, you can place a modified version of intelligence.cfc (or better yet, a component that extends it) in the settings directory.

If you look at the idle brain (“idle.cfc”) in the settings/intelligence directory, you’ll see that it only has one method of its own (that’s not inherited from the abstract brain). The *getRecommendation()* method implements a simple test to see if this brain thinks its particular recommendation (or set of possible recommendations) are a good fit for a specific agent, based on that agent’s current statistics. If this brain can make multiple suggestions, it must pick one suggestion for each agent, since the return type of this method is *Struct* and not *Array*. If you have multiple suggestions that you’d like to make for each agent, then you’ll need to place each suggestion in its own brain. Once a brain is installed in the intelligence directory, it is automatically loaded the next time suggestions are requested either in the Management Application, or by the auto-configuration process that occurs during the monitoring cycle.

```
<cfcomponent extends="abstract"
hint="suggests an idle eviction policy for agents with items in cache slower than 3x
the average frequency for the agent">

    <!--- hintText is displayed as hover-text on the
    agent recommendations page of the Management Application --->
    <cfset instance.hintText = "Hit frequency slower than 3x the average may indicate
    storage that could be better used." />

    <cffunction name="getRecommendation" output="false"
    access="public" returntype="struct"
    hint="returns a single recommendation for a specified agent">
        <cfargument name="stats" type="struct" required="true"
        hint="statistics for a single cache agent" />
        <cfargument name="auto" type="numeric" required="true"
        hint="a multiplier value representing the type of auto-configuration - perf is
        a number greater than 1, fresh is a number below 1, auto is 1 - if an agent isn't
        auto-configurable, this value will also be 1" />
        <cfset var result = getDefaultRec() />
        <cfset var testVal = ceiling(3 * auto * stats.meanFrequency) />

        <!---
        by default the limit is 3x the mean frequency for the cache agent
        multiplying by the auto value will set the limit lower for fresh, so
        content is evicted faster, and higher for perf so content is held longer
        --->

        <cfif stats.meanFrequency gte 1 and stats.minFrequency gt testVal>
            <cfset result.evictPolicy = "idle" />
            <cfset result.evictAfter = testVal />
        </cfif>

        <cfreturn result />
    </cffunction>
</cfcomponent>
```


In order to make use of the statistics for analysis, you'll need to know what statistics are available for a given agent. Here's a brief list of what you'll find in the stats structure.

Variable	Type	Description
AgentID	String	The canonical ID of the agent as used by the service
AgentName	String	The last part of the agent ID
AppName	String	The name of the application this agent is for – empty for server and cluster agents
Context	String	SRV, CLU, APP – indicates the applied context of this agent
EvictAfter	Numeric	The threshold for the currently applied eviction policy
EvictPolicy	String	The currently applied eviction policy
Hits	Numeric	The number of times content from this agent has been served
LastHit	Numeric	The number of minutes since this agent last served content
MaxFrequency	Numeric	Indicates the lowest ratio of time to number of hits for a single content item in this agent – in other words, the content item that is most frequently requested from this agent
MeanAge	Numeric	The average number of minutes a content item has remained in cache – see Oldest
MeanFrequency	Numeric	The average frequency that a single content item is requested – compare to MinFrequency and MaxFrequency
MinFrequency	Numeric	Indicates the highest ratio of time to number of hits for a single content item in this agent – in other words, the content item that is least frequently requested from this agent
Misses	Numeric	The total number of fetch requests for items that weren't already cached (a miss)
Occupancy	Numeric	The total number of items stored for this agent
Oldest	Numeric	The current age of the oldest content item in minutes, i.e. 90 means the item has been in cache for 1.5 hours – see MeanAge
StorageType	String	The currently configured storage medium for this agent

Custom Storage Types

What if you want to store your cache in some way that's not supported out of the box? What if you want to cache java objects, but you want to cache them in file? This isn't impossible to do, but it's not supported by the default file storage type in part because it requires serializing your java objects, which can be dangerous if you're not 100% certain you know what you're doing.

You can expand the CacheBox framework to use a new type of storage by creating a CFC in the `cachebox/storage/` directory that extends the default storage type (`cachebox/storage/default.cfc`). A handful of component variables allow you to expose some information about your storage type to the application.

Variable	Type	Default	Description
Instance.Config	Object	n/a	This is the CacheBox config object, giving the storage type access to the remaining application – the built-in storage types don't use it
Instance.Context	Integer	2	The largest context this storage type can support, e.g. a storage type that supports SERVER storage must also support APPLICATION storage, but not CLUSTER storage 1 = CLUSTER : 2 = SERVER : 3 = APPLICATION
Instance.Settings	String	n/a	A file name that will hold configuration settings for your storage type, relative to the settings directory
This.Description	String	n/a	Provides the description of the storage type that appears in the Management Application

There are also several methods that can or must be implemented for your storage type to work. Methods that are not required are inherited from the Default storage type.

Method	Returns	Required	Description
Delete(CacheName, Content)	Any*	Yes	Removes content from the cache
Fetch(CacheName, Content)	Any*	Yes	Retrieves content from the cache
Store(CacheName, Content)	Any**	Yes	Stores content in the cache
Configure()	Void	No	Provides a place where the object can perform additional configuration after the <i>init()</i> method without needing to override <i>init</i> or replicate arguments
getConfigForm()	XML String	No	Creates a form for configuring the storage type
getConfigPath()	String	No	Returns the fully-qualified path to the file that stores configuration for this storage type
isReady()	Boolean	No	Indicates to the application that this storage type is configured and working. By default this method returns true, so it's important to override this method if your storage type requires configuration. Alternately you can set <i>instance.isready</i> to false and later update it when configuration loaded by <i>readConfig()</i> or saved by <i>setConfig()</i>
readConfig()	Void	No	Reads saved configuration for this storage type from the file indicated by <i>getConfigPath()</i> and sets them into the instance variables
setConfig(Parameters)	Void	No	Saves the storage-type configuration (structure) to the file indicated by <i>getConfigPath()</i> , then loads the new settings
writeConfig(Config)	Void	No	Writes an XML string to the file indicated by <i>getConfigPath()</i>
* The content argument is the value returned from the Store() method			
** The value returned from Store() is placed in the metadata query that holds statistical information for all cached content			

If your storage type requires configuration, the application will simplify the process of configuring it by providing a preformatted form in the Management Application. All you need to do is supply a name for your configuration file (relative to the /settings/storage directory) and the form fields and labels. The rest of the form will be formatted by the application, and even storage and retrieval of the configuration settings are inherited from storage/Default.cfc unless you need special handling. The XML string you return from *getConfigForm()* should look something like this:

```
<form>
  [<head>... appears at the top of the form ...</head>]

  <input type="text"
    name="[setting]"
    label="The Setting"
    value="[instance.setting]" />

  [<foot>... appears below the submit button ...</foot>]
</form>
```

The head and foot sections are optional.

Here's an example of how this all fits together when creating a storage type object: file.cfc

```
<cfcomponent displayname="CacheBox.storage.file" extends="default" output="false"
hint="I cache content to a physical file -- I can only be used for agents that store
strings, not objects">

    <!--- Supports SERVER and APPLICATION contexts --->
    <cfset instance.context = 2 />

    <!--- the location of the configuration file for file storage --->
    <cfset instance.settings = "file.xml.cfm" />

    <!--- this is the description shown in the Management Application --->
    <cfset this.description = "Saves cached content to a physical file" />

    <!--- this is the only custom configuration variable needed for file storage --->
    <cfset instance.storagedirectory = "" />

    <!--- checks to see if the storage directory exists - if not, it's not ready --->
    <cffunction name="isReady" access="public" output="false" returntype="boolean">
        <cfreturn directoryExists(instance.storageDirectory) />
    </cffunction>

    <cffunction name="store" access="public" output="false" returntype="any">
        <cfargument name="cachename" type="string" required="true" />
        <cfargument name="content" type="any" required="true" />

        <!--- figure out where we should store this content --->
        <cfset var path = getFilePath(arguments.cachename) />
        <cfset var dir = getDirectoryFromPath(path) />

        <!--- if the content is binary, then we'll need to read binary later --->
        <cfset var binary = iif(isBinary(content),1,0) />

        <!--- individual content items might be stored in subdirectories and
        if the directory doesn't exist, the server throws an error on write --->
        <cfif not DirectoryExists(dir)><cfset mkdir(dir) /></cfif>

        <cfset fileWrite( path , arguments.content ) />

        <!--- anything we'll need to know later for a fetch, we should return
        from the store method. In this case we only need to know if it's binary --->
        <cfreturn binary />
    </cffunction>

    <cffunction name="fetch" access="public" output="false" returntype="any">
        <cfargument name="cachename" type="string" required="true" />
        <cfargument name="content" type="any" required="true" />
        <cfset var result = { status = 0, content = "" } />

        <cftry>
            <!--- readbinary if the file was binary when written, or just read it --->
            <cffile action="#iif(val(content),de('readbinary'),de('read'))#"
variable="result.content" file="#getFilePath(arguments.cachename)#" />

            <cfcatch>
                <!--- unable to fetch the content, indicate failure with status --->
                <cfset result.status = 3 />
            </cfcatch>
        </cftry>

        <cfreturn result />
    </cffunction>

    <cffunction name="delete" access="public" output="false">
        <cfargument name="cachename" type="string" required="true" />
        <cfargument name="content" type="any" required="true" />

        <cfset var path = getFilePath(arguments.cachename) />
```

```

    <cfif fileExists(path)><cfset fileDelete(path) /></cfif>
</cffunction>

<cffunction name="getConfigForm" access="public"
output="false" returntype="string">
    <cfset var result = "" />

    <cfsavecontent variable="result">
        <cfoutput>
            <form>
                <input type="text"
                    name="storagedirectory"
                    label="Directory"
                    value="#XmlFormat(instance.storagedirectory)#" />
            </form>
        </cfoutput>
    </cfsavecontent>

    <cfreturn result />
</cffunction>

... additional functions in original source not included here ...

</cfcomponent>

```

Custom Eviction Policies

Let's say hypothetically that you want to evict content only when the content is in an array of specific values that's stored and managed somewhere else. Maybe you're getting content from a webservice and want to tie eviction of the content to modification dates from the service. Then you need a custom eviction policy.

Custom eviction policies are actually quite simple. Each policy is a component in the eviction directory that extends the abstract eviction policy with just two public variables and one important method named *getExpiredContent()*.

Variable	Type	Default	Description
This.Description	String	n/a	Describes each eviction policy in the Management Application – use an N in the description string to indicate the location of the number for a specific cache agent (this also creates the form field for setting the value and must be bounded by white space)
This.LimitLabel	String	Max Objects	If this string is empty, the Management Application will not search for an N in the Description – this value may be used elsewhere in later versions with the intention being "[LimitLabel]: [x]", e.g. "Max Objects: 100"
Instance.Config	Object	n/a	This is the CacheBox config object, which gives the eviction policy access to the rest of the service if desired – none of the built-in eviction policies use it

The most important part of your eviction policy will be the *getExpiredContent()* method, which returns an array, telling the service which objects to expire. Arguments are called by name, so not all arguments are required, but the argument names must match.

Argument	Type	Required	Description
Cache	Query	Yes	A query containing metadata for all the items in the specified agent's cache
CacheName	String	No	This is the AgentID of the cache agent for which content is being reaped
EvictLimit	String	No	The eviction threshold for the specified agent – for eviction policies that have a threshold, this argument is numeric
CurrentTime	Numeric	No	The current time measured in minutes since midnight January 1, 1970

The Cache argument is the only required argument because this query is the method by which evictions are targeted. The array returned from this method contains a series of values from the index column of the Cache query. Those items targeted by their index values are the ones reaped from the cache. Other columns in the Cache query will probably be used to determine which records should be removed. The most important columns will be

Column	Type	Nullable	Description
HitCount	Integer	No	The number of times this content item has been served – 0 for unserved content
Index	Integer	No	Indicates the content item's current location
MissCount	Integer	No	The number of times this content item has been requested and not found in cache
TimeHit	Integer *	Yes	The last time this content was served
TimeStored	Integer *	Yes	The time the content was stored
AgentName	String	No	The name of the cache Agent being reaped
AppName	String	Yes	The name of the application (or serverid) this agent belongs to
CacheName	String	No	The fully-qualified name of the cache content
Context	String	No	The context in which this agent stores content

* Times are provided as an integer minutes since Midnight, January 1, 1970 – this makes it easier to use a query-of-query to determine the number of minutes an item has been in cache or been idle

While this may seem like a lot of complexity, the end result is actually a fairly simple object. Here's an example from the built-in eviction policies: age.cfc.

```
<cfcomponent displayname="CacheBox.eviction.age" output="false"
extends="abstractpolicy" hint="evicts after content reaches a specified age">

    <cfset this.description = "Expires N minutes after creation" />
    <cfset this.limitlabel = "Minutes" />

    <cffunction name="getExpiredContent" access="public" returntype="array"
hint="returns an array of index values for content to expire">
        <cfargument name="evictLimit" type="string" required="true" />
        <cfargument name="currentTime" type="numeric" required="true" />
        <cfargument name="cache" type="query" required="true" />

        <!--- check the evictLimit argument to make sure it's numeric --->
        <cfset var lim = val(evictLimit) />
        <cfset var result = 0 />

        <!--- find any records older than the specified time
        -- check for null values in timeStored to remove miss counts also --->
        <cfquery name="result" dbtype="query" debug="false">
            select index from cache where timeStored is null or timeStored <=
            <cfqueryparam value="#currentTime-lim#" cfsqltype="cf_sql_integer" />
        </cfquery>

        <!--- return the found indexes as an array --->
        <cfreturn listToArray(ValueList(result.index)) />
    </cffunction>
</cfcomponent>
```

Appendix: CacheBoxAgent Methods

Method	Returns	Description
Debug()	Void	Dumps all instance variables
Delete(CacheName)	Void	Immediately removes content from the cache – uses % as a wildcard in the CacheName argument
Expire(CacheName)	Void	Marks content as expired for later removal from cache – uses % as a wildcard in the CacheName argument
Fetch(CacheName)	Struct	Returns previously cached content from the service
getAgentID()	String	Returns the fully-qualified ID of this agent as used by the CacheBox service
getAgentName()	String	Returns the name that identifies this agent - agents for server or cluster scopes must share the same name in order to share the same cache
getAppliedContext()	String	Returns the context in which this agent is applied – may be different from requested context
getAppliedEvictPolicy()	String	Returns the eviction policy currently applied for this agent
getAppName()	String	Returns the application name for an agent in the application context
getContext()	String	Returns the context to which this agent should apply
getEvictPolicy()	String	Returns the requested eviction policy for this agent
getService()	Object	Returns the CacheBox service this agent is registered to (avoid this method whenever possible)
getSize()	Integer	Returns the number of content items stored for this agent
getVersion()	String	Returns the version of the agent
hasReapListener()	Boolean	Returns true if this agent is configured with a reap listener object
isConnected()	Boolean	Returns true if this agent has access to a viable CacheBox service object
isRegistered()	Boolean	Returns true if this agent has been registered in a CacheBox service and given an AgentID
reset()	Void	Expires all cache for the agent
Store(CacheName, Content)	Struct	Places content in the cache