

# *Tuning Performance*

---

This chapter tells you:

- how to determine if your system suffers from any of several common performance bottlenecks, and, if so, how to remedy the situation, and
- the salient points of garbage collection, if you have traced a performance problem to that part of the system.

Diagnosis of these or other problems requires understanding how to use statmonitor and VSD, the GemStone utility programs for measuring system performance, to monitor cache statistics. Cache statistics are described in detail in “Cache Statistics” on page 8-21. Appendix H, “statmonitor and VSD Reference,” describes how to use VSD and statmonitor.

## **11.1 Common Performance Bottlenecks**

The following most commonly experienced performance bottlenecks each have an associated template to help you determine if this is a problem for your application.

- The shared page cache is too small.
- The commit record backlog is too large.

- The page server is swamped.
- The sweep phase of epoch garbage collection is inefficient.
- The garbage collection Gem is overloaded.
- In-Gem garbage collection is excessive.

## Shared Page Cache Too Small

Statistic	Default Filter	Process	Explanation
FreeFrameCount		shared page cache monitor	See page 8-26.
FramesFromFindFree	per second	all	See page 8-26.
FramesFromFreeList	per second	all	See page 8-26.

### Background

The shared page cache is a representation of part of the repository in RAM, allowing the Stone and Gem processes to read objects from memory rather than disk, similar to the way many applications use virtual memory. Because memory access is significantly faster than disk I/O, applications can run faster if they find most of the objects they require already in the shared page cache.

Both on disk and in the cache, objects reside on eight-kilobyte pages. The shared page cache has a fixed number of eight-kilobyte *frames* (the precise number depending on the cache size); each frame can hold one page.

When a Gem seeks an object, it first looks in the shared page cache; only if the object is not already there does it search on disk. When it finds the object, it must obtain a free frame before it can read the page into the shared page cache.

The shared page cache monitor maintains a list of cache frames not currently in use: the number of frames in this list is reported by the statistic **FreeFrameCount**. When the Stone or a Gem uses a frame to hold a page, the frame is taken off the free frame list. Frames are added to the free list at checkpoints, or when a Gem logs out. However, when a page server writes a page to disk (perhaps in response to a committed transaction), it does not always add the frame that contained that page to the free frame list right away. Instead, it waits until the number of free frames in the list is less than a certain percentage of the frames in the cache. Thus, the free frame list is not always up-to-date.

#### NOTE

*For GemStone/S 5.1.4, this percentage was 5%. In GemStone/S 5.1.5 and 6.0, this percentage is 12.5%.*

A Gem in need of a free frame can get one in either of two ways:

- It can take a frame from the free frame list. If it finds one, statmonitor increments the statistic **FramesFromFreeList**.
- If it doesn't find one, the Gem scans the cache for a free frame that hasn't yet been added to the free list. If it finds one, statmonitor increments the statistic **FramesFromFindFree**.

Which method the Gem uses to get a free frame is determined by the Gem configuration parameter `GEM_FREE_FRAME_LIMIT`, set in the Gem's configuration file or at runtime. The algorithm is:

```
if FreeFrameCount > GEM_FREE_FRAME_LIMIT
    then take frame from free list
else scan cache for frame
```

By default, `GEM_FREE_FRAME_LIMIT` specifies that each Gem is entitled to take from the free list until the free list falls below 10% of the frames in the shared page cache. The garbage collection Gem (GcGem) is a special case: it takes frames from the free list until the number of free frames falls below 5% of the frames in the shared page cache.

For example, a 100 MB shared page cache contains 12,800 frames (100 MB / 8 KB). By default, then, each Gem must scan the cache instead of taking from the free list, when the free list holds fewer than 1280 frames (10% of 12,800 = 1280). The GcGem won't have to scan the cache until the free frame list falls below 640 (5% of 1280).

If no free frame can be found with either method, the Gem looks for a dirty page and writes it to disk itself, thus freeing its frame. This is the only occasion for a Gem to write a page to disk, and it is expensive.

The Stone always has a free frame limit of 0, meaning the Stone can take frames from the free list until the last one is gone. The Stone needs free frames to process a commit; if this slows down, all users suffer slower performance.

## Evaluation

Getting a frame from the free list is significantly faster than having to scan the entire cache for a free frame. Therefore, the ratio of `FramesFromFreeList` to `FramesFromFindFree` can reveal a significant opportunity for increasing performance for your application.

Another sign that the shared page cache is too small is that `FreeFrameCount` is almost always at or near the free frame limit.

Finally, if you see that Gems are doing `pageWrites`, as described in “To Monitor Page Reads and Writes by a Session” on page 8-14, your shared page cache is almost certainly too small, or the Gem’s free frame limit is too high.

For the Stone, the value of `FramesFromFindFree` should always be zero. If it is not, the shared page cache is too small, or not being used appropriately.

Additional helpful statistics:

- **LocalPageCacheMisses** and **LocalPageCacheHits** are another way to determine whether a Gem needs free frames. `LocalPageCacheMisses` is incremented when a Gem tries to access an object that it has not yet used, and finds it is not in its private page cache. Because the object may be in the shared page cache (if some other process has already accessed it), this statistic can be somewhat misleading, but, in general, if the Gem experiences more misses than hits, it is trying to access objects frequently, and has frequent need to find frames.
- **PageReads** reports the number of times a Gem has to read a page from disk. This occurs every time it needs an object not already in its private cache, or in the shared page cache.
- **GlobalDirtyPageCount** is the number of dirty pages that the page server cannot yet write to the repository.
- **PageWrites** reports the number of times a Gem has to write a page to disk in order to free a frame, because it was unable to find a free frame either from the free list or by searching the shared page cache.

## Solutions

If a Gem more frequently scans the shared page cache for free frames instead of finding them on the free list, consider starting one or more free list page servers. (A description and instructions are available in “Adding Page Servers” on page 1-45.) Add one at a time until `FramesFromFindFree` goes to zero, or near zero. Increasing the size of the shared page cache may also be helpful.

Worse, if a Gem is writing pages, consider increasing the size of the shared page cache. For details on setting the page cache size, see “To Set the Page Cache Options and the Number of Sessions” on page 1-14.

Another possible remedy could be redesigning your application so that it needs fewer pages. Clustering objects often used together on a single page, or choosing

different data structures, can under some circumstances significantly reduce the number of pages the Gem requires.

*NOTE*

*Gems can be configured individually. To do so, see “Naming Executable Configuration Files” on page A-6.*

## Commit Record Backlog Too Large

Statistic	Default Filter	Process	Explanation
TotalCommits	per second	Stone	See page 8-39.
CommitRecordCount	per second	Stone	See page 8-24.
CommitQueueSize	per second	Stone	See page 8-23.

### Background

A *commit record* is the structure the Stone uses to provide a Gem with its view of the repository. Every time a Gem commits a transaction, it creates a commit record—a list of the objects read and written by the Gem that created it, and other associated information. The Stone maintains a collection of commit records ordered from oldest to newest; when a Gem commits or aborts a transaction and gets an updated view of the repository, its view is the state of the persistent objects as represented by the latest commit record.

At any given time, each Gem is connected to exactly one commit record, but each commit record can be connected to more than one Gem. That is, if Gem A commits a transaction and Gem B aborts immediately thereafter, they’ll both share the same view of the repository: the view represented by the commit record created by Gem A.

The *commit record backlog* is the collection of all commit records connected to all active Gems—that is, all active views of the system. Commit records are kept in the shared page cache, where they contend for frames along with other resources. A commit record cannot be removed from the shared page cache until:

- no Gem session is looking at the view it represents, and
- it is the oldest commit record.

Therefore, if a single Gem stays in a transaction for a long time without committing or aborting (thereby freeing its commit record), newer commit records cannot be removed, even if no Gem references them. If this situation remains unchanged for a long time, a significant commit record backlog can build up.

The Stone's only means of correcting the situation is to signal the Gem to abort its transaction, thereby releasing the commit record, permitting it (and perhaps other unreferenced commit records) to be removed from the shared page cache.

A Gem will receive this signal, called a SigAbort, if:

- it is outside a transaction, and
- it refers to the oldest commit record, and
- the commit record backlog is greater than the specified maximum.

If a Gem is in transactionless mode, it is never in a transaction and never needs to commit changes. In this case, GemStone/S handles SigAborts for you transparently.

If a Gem is in automatic transaction mode, it is never outside a transaction and therefore will never receive a SigAbort. Automatic transaction mode can therefore cause significant performance problems if Gems do not commit or abort frequently.

If a Gem is in manual transaction mode, you'll need to write a handler to catch SigAbort signals and respond appropriately. (The class GsSession has a generic SigAbort handler method, which you can customize by writing your own block.)

If a Gem receives a SigAbort and does not respond in the length of time specified by the STN\_GEM\_ABORT\_TIMEOUT configuration parameter (by default, one minute), then the Stone sends it a SigLostOTRoot error and then waits the number of seconds specified in the configuration parameter STN\_GEM\_LOSTOT\_TIMEOUT for the Gem to acknowledge receiving the signal and abort. If the Gem does not acknowledge this signal, the Stone either terminates the Gem or (if the value of STN\_GEM\_LOSTOT\_TIMEOUT is -1) revokes the Gem's access to the old commit record, which can then be removed.

A Gem will receive a SigLostOTRoot if:

- it was sent a SigAbort, and
- the time specified by STN\_GEM\_ABORT\_TIMEOUT has passed, and
- the Gem has not responded by aborting and releasing the commit record.

If a Gem is hung, it will not be able to respond to either of these signals. To clean up the Gem, the last fallback is the STN\_GEM\_TIMEOUT configuration parameter (by default, set to 0, disabling the timeout). This is the number of minutes after which lack of Gem interaction causes the Stone to forcibly log out the errant Gem.

A related configuration parameter, `GEM_RPCGCL_TIMEOUT`, lets you control the number of minutes after which lack of communication between a remote Gem and your application causes the Gem to terminate.

(For detailed information about all these configuration parameters, see Appendix A).

The Stone is the process responsible for removing commit records from the shared page cache. While it is doing so, it is not processing commits from Gems trying to commit, thereby causing the commit queue to grow and commit performance to deteriorate for all Gems.

When a commit record causes a large backlog, other problems ensue:

- *Shadow pages*—pages holding the old, unmodified state of modified objects—cannot be listed as needing reclaiming.
- Voting cannot complete, thus impeding garbage collection; all logged-in Gems must be committing or aborting for memory to be reclaimed. (For details of garbage collection, see Chapter 10, “Managing Growth.”)
- Even though pages can be reclaimed, the old pages cannot be disposed of while commit records still in the queue might refer to shadow objects on those pages.
- As the commit record backlog grows, so does the amount of work necessary for the session with the oldest commit record to compute the read/write set union to determine commit conflicts.
- Finally, if commit records fill the shared page cache, they are written to disk. Temporary objects may also be written to disk, from where it is much more difficult to reclaim the pages they occupy and the object identifiers they use. The repository grows larger,

## Evaluation

If **CommitRecordCount** is greater than twice `STN_MAX_SESSIONS`, or if the commit record count is continually growing, application performance is probably impacted by page swapping. Even if **CommitRecordCount** is lower than this maximum, swapping could still be a problem, especially if **FreeFrameCount** is also low.

To find the session or sessions connected to the oldest commit record, evaluate:

```
System sessionsReferencingOldestCr
```

**TotalCommits** indicates the transaction load and how fast the commit record backlog can grow.

**CommitQueueSize** indicates that the Stone is busy, most likely serving the commit requests of other Gems. A large commit queue size means that the cost of doing commits—of reading the read/write set union for all logged-in Gems in order to determine commit conflicts—is also growing.

Additional helpful statistics:

- **SigAbortsSent** reports the number of times the Stone has signaled a given Gem to abort, although the session may be in a sleep or I/O wait state and not yet aware of having received the signal.
- **SigAbortsReceived** reports the number of times the Stone has signaled a given Gem to abort, that it has received and recognized.
- **LostOtsSent** is the number of Lost OT Root signals the Stone has sent to a given Gem, although the session may be in a sleep or I/O wait state and not yet aware of having received the signal.
- **LostOtsReceived** is the number of Lost OT Root signals received and recognized by a given Gem.
- **CommitCount** and **AbortCount** are the number of times a Gem has committed or aborted a transaction, respectively.
- **TransactionLevel** allows you to determine the transaction status of a Gem:
  - 1 = in a transaction
  - 0 = not in a transaction
  - 1 = in transactionless mode

*NOTE*

*If statmonitor samples are too infrequent, you may miss transitions between levels.*

## Solutions

If appropriate, configure your system with a short value for `STN_GEM_LOSTOT_TIMEOUT`, so that unresponsive Gems can be terminated quickly. For a discussion of the trade-offs involved, see “`STN_GEM_LOSTOT_TIMEOUT`” on page A-22.

Alternatively, if appropriate, stop the session(s) connected to the oldest commit record by evaluating:

```
System stopSession: aSessionId
```

You can also lower the value of `STN_SIGNAL_ABORT_CR_BACKLOG` so that the Stone does not permit the commit record backlog to grow as large.

Another option, recommended for production systems, is to set the configuration parameter `STN_GEM_TIMEOUT` to automatically log off unresponsive sessions.

*NOTE*

*A Gem running a long query will not time out and be logged off if it is getting pages and object IDs from the Stone. Requesting locks will also count as activity and will not invoke the timeout.*

Make sure Gems ordinarily work in manual transaction mode and keep transactions short. You can also redesign your application so that sleeping Gems do so in transactionless mode.

When a Gem is sleeping, it will not respond to signals. Therefore, it's a good idea to program your `sleep` method in a loop, so that the Gem sleeps for half the amount of time specified in `STN_GEM_TIMEOUT`, then awakens briefly before returning to sleep. The interpreter activity will detect a signal, if any has been sent.

Finally, for a temporary work-around, you can increase the value of `STN_MAX_SESSIONS`, thus allowing the system to better tolerate the problem while you work on a real solution.

## Swamped Page Server

Statistic	Default Filter	Process	Explanation
AioDirtyCount	per second	page server	See page 8-21.
AioCkptCount	per second	page server	See page 8-21.
LocalDirtyPageCount	per second	shared page cache monitor	See page 8-29.

## Background

The asynchronous I/O page server (also called the AIO page server and hereafter called simply *page server*) is the process responsible for writing *dirty pages*—pages with modified objects—from the shared page cache to the repository extents. The Stone starts one or more page servers as part of its own startup (the number depends on the value of the `STN_NUM_LOCAL_AIO_SERVERS` configuration parameter). Thereafter, the page server scans the shared page cache for dirty pages and writes them to disk asynchronously, freeing the Gem from having to perform this I/O-intensive task.

At a checkpoint, the page server writes all the dirty pages from the shared page cache to the repository. In full logging mode, checkpoints occur as specified by the Stone's configuration parameter `STN_CHECKPOINT_INTERVAL`. In partial logging

mode, a checkpoint can occur more often, if the size of a transaction exceeds the value set by the configuration parameter `STN_TRAN_LOG_LIMIT`.

Checkpoints are most often triggered by a Gem committing a transaction; they also occur at the start of a new transaction log or repository backup. Checkpoints may take seconds or minutes, but a checkpoint in progress does not block the system: transactions can commit as usual. In addition to all dirty pages being written to the extents, checkpoint records are written to each extent and to the transaction log.

If the page server cannot write dirty pages to disk fast enough to keep up with the Gems committing transactions, many other system processes will be delayed for lack of free frames in the shared page cache. Also, the page server will have more work to perform at each checkpoint, which can slow other processes such as page reclamation.

## Evaluation

The statistics in this template help you determine if the page server can keep up with the demands created by Gems committing transactions. Compare **AioDirtyCount** to **AioCkptCount**: if a great many more pages are written at checkpoints (**AioCkptCount**) than otherwise (**AioDirtyCount**), the page server is probably falling behind and having to catch up at each checkpoint.

If both numbers are high, the page server could be operating at maximum.

If **LocalDirtyPageCount** is high and drops only slowly, if at all, the page server is probably not operating at peak efficiency.

Page reclamation worsens the problem: a slow page server means too few free frames in the shared page cache for the GcGem to use for copying live objects. Thus, it will spend more time scanning the cache and possibly swapping to disk.

Additional helpful statistics:

- **GlobalDirtyPageCount** (shared page cache monitor only) reports the number of dirty pages in the shared page cache that are dirty but not yet eligible for writing to disk because they're not yet committed. If this value is very large, then very large transactions may be filling the cache. If the Stone is also using this shared page cache, another possibility is that the Stone's private page cache is too small.
- **CheckpointCount** (Stone only) reports the number of checkpoints since the Stone was started. If partial logging is in effect, a rapidly increasing **CheckpointCount** indicates that `STN_TRAN_LOG_LIMIT` may be set too small.

## Solutions

If the page server cannot keep up with demand, the obvious remedy is to increase the number of page servers; however, whether you realize any benefits depends on other characteristics of your system. Parallelization is of no use without parallel resources; multiple page servers boost performance only for systems with:

- over four extents (one page server can ordinarily handle up to four extents),
- multiple CPUs (to allow parallel execution), and
- extents on separate spindles or the equivalent (to allow concurrent writes to disk).

In any case, over four page servers are unlikely to provide additional benefit, unless your repository is particularly large.

Other options exist for increasing system throughput:

- Ensure that the extents, the system swap file, and the transaction logs are all on separate disk spindles.
- Consider using raw partitions if you are not already doing so.
- Experiment with non-RAID devices, if appropriate, to see if they improve performance.
- If you have spare CPU cycles, increase the I/O rate of the page server.
- Consider redesigning your application to make more efficient use of I/O operations. For example, traversing sequentially through an `OrderedCollection` or `Array` can cause an application to perform a large number of I/O operations if the referenced objects are not already in the shared page cache.
- If `GlobalDirtyPageCount` is high, redesign your application to use smaller transactions, if possible.
- If the shared page cache in question is used by the Stone and `GlobalDirtyPageCount` is high, it may help to increase the size of the Stone's private page cache.
- If partial logging is in effect and `CheckpointCount` is rapidly increasing, it may help to increase the value of the Stone's `STN_TRAN_LOG_LIMIT` configuration parameter.

## Inefficient Epoch Sweep

Statistic	Default Filter	Process	Explanation
EpochGcCount	per second	Stone	See page 8-24.
PossibleDeadSize		Stone	See page 8-35.
ProgressCount		GcGem	See page 8-35.

### Background

*Garbage collection* is the reclamation of pages on disk and object identifiers. The first phase of garbage collection, called *mark/sweep*, identifies objects that might be dead.

*Epoch garbage collection* identifies the possibly dead objects from a finite set of recent transactions, called the *epoch*, instead of the entire repository. It checks all objects created from the start time to the end time of the epoch, looking for objects that have been dereferenced. Epoch garbage collection is efficient because, for many applications, the vast majority of objects are short-lived, created to service temporary application needs and not intended to persist in the database. This is especially true of applications in which numerous small transactions mostly update a few previously committed objects.

*NOTE*

*For a more complete description of garbage collection, see Chapter 10, especially the section entitled "Epoch Garbage Collection" on page 10-14.*

Epoch garbage collection supplements `markForCollection`; it does not replace it. It does not find:

- objects created before the beginning of the epoch that have been dereferenced during the epoch, nor
- objects created during the epoch and not dereferenced until after the epoch has ended.

Mark/sweep actually encompasses two operations on the set of all objects created during the epoch:

1. The *mark* phase identifies all *live objects*—objects that can be reached through references starting with the AllUsers root object.
2. The *sweep* phase identifies *possibly dead objects*—all objects created in the epoch that were not identified as live objects in the first sweep, minus any unused object identifiers.

The reason these objects are considered “possibly” instead of “definitely” dead is that mark/sweep can take a significant period of time, during which a Gem may somehow commit one of the objects previously identified as possibly dead. These operations use considerable CPU cycles and disk I/O: to identify live objects, all pages containing objects created during the epoch are copied from disk into the shared page cache, if they are not already there. And the object table must be swept for unused object identifiers, so a great many object table pages are read into the GcGem’s private page cache. Either or both of these operations can cause swapping, and hence be time-consuming.

Epoch garbage collection is one of several responsibilities of the GcGem. While it is running, the GcGem is not available to perform its other functions.

You can disable epoch garbage collection or change how often it runs. Assuming it’s enabled, these GcGem configuration parameters control the length of an epoch:

- `epochGcTimeLimit`  
The maximum frequency of epoch garbage collection—by default, 15 minutes.
- `epochGcTransLimit`  
The number of transactions required to trigger epoch garbage collection—by default, 5000.
- `epochGcByteLimit`  
The number of bytes of new or modified committed objects required to trigger epoch garbage collection—by default, 5 million.
- `deferEpochReclaimThreshold`  
The number of pages needing to be reclaimed (another responsibility of the GcGem’s) that will cause the GcGem to defer epoch garbage collection in favor of reclaiming pages—by default, 1000.

Epoch garbage collection occurs when:

```
(pages that need reclaiming < deferEpochReclaimThreshold) AND  
(the time since last epoch > epochGcTimeLimit ) AND  
( (transactions since last epoch > epochGcTransLimit) OR  
(bytes committed since last epoch > epochGcByteLimit ) )
```

## Evaluation

**EpochGcCount** per second should reveal that the GcGem is performing epoch garbage collection regularly, not deferring it because of a backlog of pages needing to be reclaimed.

**PossibleDeadSize** should also show a fairly regular graph. It should reveal that each epoch garbage collection is finding enough possibly dead objects to make it worth doing.

**ProgressCount** should show a fairly regular graph as well. During epoch garbage collection, ProgressCount increases as GcGem marks and sweeps the objects created during the epoch. First, it shows the number of objects marked. It then goes to zero, then up again as it shows the number of objects identified as possibly dead.

If too many objects must be swept, performance may slow because a large number of pages may have to be read into the shared page cache.

Additional helpful statistics:

- **PagesNeedReclaimSize** reports the approximate number of pages that need to be reclaimed. This statistic is updated only every 15 seconds, and includes only pages on the Stone's list. It is always an overestimate, and larger values are less accurate than smaller values.
- **GcPagesNeedReclaiming** reports the number of pages on the Stone's list of pages needing to be reclaimed, plus shadow pages trapped in commit records that will need reclaiming when the commit records are removed from the system. It is this value that's compared to `deferEpochReclaimThreshold` to determine whether the GcGem will perform an epoch garbage collection or reclaim pages.
- **GcDeferEpochThreshold** is the value of the GcGem's configuration parameter `deferEpochReclaimThreshold`.

## Solutions

Problems caused by epoch garbage collection are typically either:

- the epoch is the wrong length (too frequent or too infrequent), or
- the application is creating too many short-lived objects.

You can tune the GcGem parameters to change the epoch length. The default period of 15 minutes tends to work better for applications that commit one or more transactions per second. To be helpful, epochs should last longer than the average lifetime of short-lived objects; for details, see “Epoch Garbage Collection” on page 10-14. In general, the longer the epoch, the greater the need for additional storage during the epoch. Also, bursts of garbage collection activity will be less frequent but longer. However, you’ll be able to run `markForCollection` less frequently.

If statistics reveal that few objects are garbage-collected, consider disabling epoch garbage collection or running it less frequently, when relatively few users are on the system.

Another approach is to monitor system usage and adjust epoch garbage collection to match—for example, by running it on shift boundaries, if users work in shifts.

Instructions for changing epoch length and other GcGem parameters while the system is running are provided in “Epoch Garbage Collection” on page 10-14. You can also change these parameters in a configuration file specific to the GcGem, so that the changes remain in effect if the system is stopped and restarted.

Instructions for doing so are provided in “The GcGem” on page A-6.

If `PossibleDeadSize` and slow performance indicate too many short-lived objects are being read during the sweep, you can:

- enlarge the shared page cache to handle the extra pages,
- shorten the epoch length, or
- redesign your application to create less garbage.

If application developers haven’t explicitly addressed the issue of garbage creation, it’s not unusual for to discover that the application puts unnecessary demands on the garbage collection.

To see what kinds of objects are created when executing a GemStone method, use `ProfMonitor` with object tracing turned on.

## Overloaded GcGem

Statistic	Default Filter	Process	Explanation
PagesNeedReclaimSize		Stone	See page 8-34.
GcPagesNeedReclaiming		Stone	See page 8-27.
EpochGcCount	per second	Stone	See page 8-24.
CommitCount	per second	GcGem	See page 8-23.

### Background

In addition to epoch garbage collection, the GcGem has several responsibilities:

- It finalizes voting among the Gems during repository-wide garbage collection.
- It reads the list of possible dead objects looking for special cases: indexed collections and compiled methods, both of which require extra processing.
- It reclaims pages with shadow objects and dead objects.

These tasks all must contend for the GcGem; see the previous discussion starting on page 11-12 for details, and Chapter 10 for an in-depth discussion. However, of them all, page reclamation takes longest—that's why GemStone 6.0 provides new GcGems specialized for page reclaim.

When garbage collection is complete (either `markForCollection` or epoch garbage collection) and a commit record is removed, the Stone adds the commit record's shadow pages to a list of pages that need reclaiming — **GcPagesNeedReclaiming**. The GcGem works from this list to reclaim pages in batches.

Page reclamation starts when the size of `GcPagesNeedReclaiming` exceeds its `reclaimMinPages` configuration parameter (GcGems dedicated to reclaiming pages with shadow objects use this), or when `DeadNotReclaimedSize` is greater than zero (GcGems dedicated to reclaiming pages with dead objects use this). The GcGem's `reclaimMaxPages` parameter determines the batch size: how many pages the GcGem reclaims per commit.

When a page is reclaimed, all the still-living objects on that page are copied to a new page, topped off with other live objects from other pages until nothing is left but shadow objects, dead objects, or free space. This is an expensive operation, particularly in terms of disk I/O: for example, the GcGem typically copies most of the object table into its private page cache, so that it can reclaim object identifiers.

During page reclaim, the GcGem performs a great many pages reads from the repository.

The GcGem places the new pages in its private page cache and therefore needs a high number of cache frames. The GcGem moves new pages into the shared page cache when it completes a batch and commits the transaction.

## Evaluation

**PagesNeedReclaimSize** reports the approximate number of pages that need to be reclaimed. This statistic is updated only every 15 seconds, and includes only pages on the Stone's list.

**GcPagesNeedReclaiming** reports the number of pages on the Stone's list of pages needing to be reclaimed, plus shadow pages trapped in commit records that will need reclaiming when the commit records are removed from the system. It is this value that's compared to `deferEpochReclaimThreshold` to determine whether the GcGem will perform an epoch garbage collection or reclaim pages.

**EpochGcCount** per second should reveal that the GcGem is performing epoch garbage collection regularly, not deferring it because of a backlog of pages needing to be reclaimed.

The GcGem's **CommitCount** per second should reveal that it is able to process batches of pages and commit the new pages, thus permitting the Stone to dispose of old commit records and return pages and object identifiers to their respective free pools.

Additional helpful statistics:

- **ReclaimCount** reports the number of times the GcGem has reclaimed pages since the Stone was started.
- **ReclaimedPagesCount** reports the number of pages the GcGem has reclaimed since the Stone was started.
- **GcReclaimMaxPages** reports the current value of the GcGem's `reclaimMaxPages` parameter—by default, 200. This means that the GcGem will reclaim 200 empty, shadow, or dead pages in a batch before making itself available for other work, if any.
- As in the discussion of “Shared Page Cache Too Small” on page 11-2, the ratio of **FramesFromFindFree** to **FramesFromFreeList** can tell you if the GcGem has to spend too much time scanning the shared page cache for free frames.

## Solutions

If you find that the GcGem is not keeping up with page reclamation, consider using one of the GcGems specialized for reclaiming pages; descriptions and instructions are available in “GcGems Specialized to Reclaim Pages” on page 10-30. To run one or more of these GcGems, you will have to temporarily halt the generic GcGem, which you can then restart after the page-reclaim backlog has been resolved. If this is an ongoing problem, you may wish to regularly schedule one or more page-reclaim GcGems.

If these specialized GcGems are impractical for you, consider increasing the GcGem’s private page cache from its default size—a larger private page cache can hold more object table pages without disk swapping. Consider increasing the GcGem’s private page cache to at least 20 MB. Use the procedure described starting on page 11-15 to increase this value in a customized configuration file.

(You may also wish to change the GcGem’s `GEM_FREE_FRAME_LIMIT` parameter as well. For details, see the discussion beginning on page 11-2.)

You may also wish to tune the reclaim parameters—for example, the maximum and minimum batch sizes, to make batches either smaller (so that the GcGem isn’t busy as long) or larger (so that the GcGem can get more pages reclaimed at once), depending on whether the statistics reveal a backlog of pages that need reclaiming.

Finally, if the ratio of `FramesFromFindFree` to `FramesFromFreeList` indicates that the GcGem has to spend too much time scanning the shared page cache for free frames, consider reducing the GcGem’s `FreeFrameLimit` (for more details, see “Shared Page Cache Too Small” on page 11-2), or starting additional free list page servers (for instructions, see “To Add Free List Page Servers” on page 1-47).

## Excessive In-Gem Garbage Collection

Statistic	Default Filter	Process	Explanation
ScavengeCount	per second	Gem	See page 8-36.
TimeInScavenges	per second	Gem	See page 8-38.
MakeRoomInOldSpaceCount	per second	Gem	See page 8-30.
NotConnectedObjsSetSize	per second	Gem	See page 8-32.

## Background

In addition to access to the shared page cache, each Gem has private memory. Much of this memory—the *local object memory*—is intended to serve as the Gem’s

private scratch space and is therefore organized by objects. However, some of it—the Gem’s *private page cache*—is organized in 8 KB pages like the shared page cache.

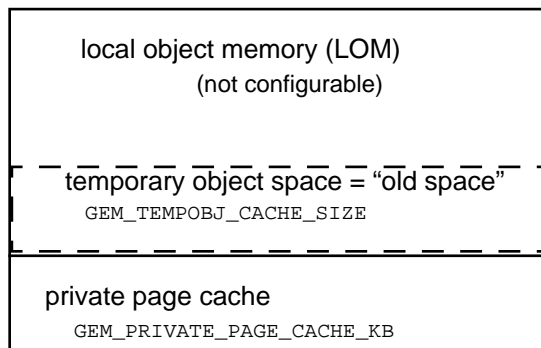
To aid memory reclamation, local object memory is divided into generation spaces. For the most part, generation spaces are inaccessible to application developers and cannot be configured; however, the longest-lived generation space is a special case. Known as *temporary object space* or simply *old space*, the size of this portion of local object memory is configurable. For an object, old space is the last stop before being placed on a page.

**NOTE**

*For additional background and basic guidelines, as well as specific tuning instructions, see “How to Tune Session Performance” on page 2-11.*

Figure 11.1 shows the structure of a Gem’s private memory:

**Figure 11.1 In-Gem Memory**



The Gem creates new objects smaller than 8 KB in local object memory, assigning each an object identifier. If the object is destined for persistence, it moves through local object memory generation by generation, at last ending in old space. Upon commit, the object is assigned a page in the private page cache and, from there, moves into the shared page cache.

New objects 8 KB or larger are created in the private page cache to start with.

When the Gem searches for already extant persistent objects, it searches first in its private page cache, then the shared page cache, and last in the extents. The Gem uses its private page cache to modify existing persistent objects; if the object is not already there, the Gem copies it to its private page cache and then modifies it.

The Gem performs three kinds of garbage-collection, corresponding to its three kinds of memory:

- Each generation space in local object memory is automatically garbage-collected by a process called *generation-scavenging*. A new object survives progressive generation-scavenges if it continues to be referenced.
- When old space is full, it's garbage-collected by a special generation-scavenge called `makeRoomInOldSpace`.
- If old space is still full, temporary objects overflow into the private page cache, where they become known as *not-connected objects*, and form part of the `notConnectedSet`. Here they are eligible for not-connected garbage collection,

### Tuning the `notConnectedSet`

When an application executes, it inevitably makes temporary objects. It is best if these objects are garbage-collected while still in local object memory; complications ensue when temporaries overflow into the private page cache and become part of the `notConnectedSet`.

Objects become part of the `notConnectedSet` in one of four ways:

- Old space filled and the temporary object overflowed into the private page cache.
- The temporary object was 8 KB or larger when created and therefore began in the private page cache.
- The object was smaller than 8 KB but was referenced by a temporary larger than 8 KB. All such objects are moved into the private page cache upon a commit, where they join the large temporary in the `notConnectedSet`.
- A commit failed. When a Gem starts a commit operation, it moves the objects intended to become persistent into the private page cache. If the commit fails, these objects join the `notConnectedSet`.

The `notConnectedSet` is garbage-collected automatically after a generation-scavenge if the following two parameters are exceeded:

- **`notConnectedThreshold`**  
The minimum number of objects in the `notConnectedSet` required to trigger `notConnectedSet` garbage collection—by default, 2000.
- **`notConnectedDelta`**  
The minimum number of objects by which the `notConnectedSet` is required to grow since the last `notConnectedSet` garbage collection, in order to trigger the next `notConnectedSet` garbage collection—by default, 300.

Garbage collection of the `notConnectedSet` is the most expensive form of in-Gem garbage collection. The Gem has to sweep the contents of the `notConnectedSet` from a known root, searching for permanent references, a CPU-intensive operation. If the private page cache pages have been swapped to disk, the operation becomes I/O-intensive as well.

Objects in the `notConnectedSet` are uncommitted until a successful commit. Upon commit, some objects in the `notConnectedSet` become committed objects, eligible to be written to the extents by the page server.

If an object in the `notConnectedSet` is unreferenced when not-connected garbage collection runs, the Gem can remove the object from the `notConnectedSet` and delete it. After an object has been committed, however, the Gem cannot safely delete it. Instead, the object must be removed by one of the two forms of repository-wide garbage collection: epoch garbage collection, or `markForCollection`.

## Evaluation

**ScavengeCount** reports the number of times generation-scavenging has run since the Gem was started. **TimeInScavenges** reports the cumulative CPU time, in milliseconds, spent in generation-scavenging. Objects collected during generation-scavenging are objects that don't enter the `notConnectedSet`, never become committed and written to disk. Together, these two statistics can be an indication of how efficiently your application is coping with temporaries.

If **MakeRoomInOldSpaceCount** increments frequently, local object memory may be overflowing and adding to the `notConnectedSet`.

**NotConnectedObjsSetSize** reports the size of the `notConnectedSet`. If its value grows consistently, temporary object space may be too small, or your application may be creating too many large temporaries. Compare the `NotConnectedObjsSetSize` with `MakeRoomInOldSpaceCount` to determine if frequent overflows from temporary object space are causing the `notConnectedSet` to grow.

If the **NotConnectedObjsSetSize** grows after each commit, one or more large temporaries is probably referring to other, smaller temporaries which are being pulled into the `notConnectedSet` upon commit. Further diagnostic tools:

- To determine if an object is committed, send it the message `isCommitted`.
- To determine if an object is connected, send it the message `isConnected`.
- To determine the size of an object in bytes, send it the message `physicalSize`.

Additional helpful statistics:

- **DeadObjCount** reports the number of dead objects collected in the Gem.
- **FailedCommitCount** is useful for determining if the Gem has suffered a great many failed attempts to commit. Such attempts could grow the `notConnectedSet` significantly. Compare this statistic with **CommitCount** to see the ratio of successful to failed commits.
- **GcNotConnectedCount** measures how many times the Gem has garbage-collected the `notConnectedSet` since the Gem was started. Compare this with `NotConnectedObjsSetSize` to determine if garbage collection is decreasing the `notConnectedSet`.
- **GcNotConnectedDeadCount** measures the total number of dead objects that the Gem found during all garbage-collections of the `notConnectedSet` since the Gem started. Apply a per-second filter to view how many dead objects the Gem found during the last garbage-collection cycle of the `notConnectedSet`.

## Solutions

If **MakeRoomInOldSpaceCount** increments frequently, increasing the size of old space may improve performance.

If your application makes many new persistent objects, you may instead wish to commit more frequently. This will cause new persistent objects to move out of old space and into the shared page cache more often, thus making more room in old space.

Finally, redesign your application to make more efficient use of temporaries. For example, reuse large temporary objects such as collections. Collections are dynamically resizable in GemStone Smalltalk; instead of making a new one, resize and old one to 0 and reuse it.

If the `NotConnectedObjsSetSize` keeps growing, collect `notConnectedSet` garbage before committing. To do so, evaluate:

```
System markNotConnectedForCollection
```

This operation is also useful after a failed commit.

Or, if statistics indicate that `notConnectedSet` garbage-collection should happen more often on a regular basis, lower the values of the parameters **notConnectedThreshold** and **notConnectedDelta**.

## 11.2 Garbage Collection for Tuners

This section is for those who have identified garbage collection as a serious performance bottleneck. Below is a capsule summary of a long and complex process: for complete details, see Chapter 10, “Managing Growth.”

Following this is a discussion of problems you might encounter during these various steps, their symptoms, and potential remedies. Following that discussion are three examples on an actual database with default and new values for modified parameters.

You’ll get more out of this section if you take the time now to consider these questions:

- What problems are you experiencing? Host problems such as CPU availability, I/O bandwidth, disk swapping, network bandwidth are best diagnosed with host operating system tools. When you have characterized the problem to this level, it can be helpful to look further.
- What system behavior is causing the problem? Slow commits? Repository growth? System login delays?
- In tuning garbage collection, what is your goal? Do you wish it to run as fast as possible, or to disturb other users as little as possible? Or perhaps you’d like a balance between these two goals?

### Example Garbage Collection Cycle

GemStone/S has several kinds of garbage collection: the automatically run epoch garbage collection and in-Gem garbage collection of several kinds, as well as the explicitly run `markForCollection`, `markGcCandidates`, and various kinds of `startGC`. This particular example starts with a run of `markForCollection`:

1. The Gem running `markForCollection` finds all the live objects by traversing references, starting at the system root `AllUsers`.
2. It computes the *set of possible dead objects* as follows:
  - Subtract the live objects from the universe of possible objects: objects whose IDs range from zero to the highest object ID in the system.
  - Also, subtract all the unassigned object IDs in that range.

The object table is a key data structure for this step.

3. The Gem running `markForCollection` sends the Stone the list of possibly dead and returns.

The Stone now holds the possible dead set in RAM until the next checkpoint. If the system should fail at this point, you'll have to run `markForCollection` again.

*NOTE*

*Discussion of the rest of these steps applies also to epoch and special-candidate garbage collection.*

4. Now every Gem currently logged in the system must search the possible dead set for any objects to which it holds references. Then it must commit or abort, at which time it votes to either keep an object in the set, or remove it (if it holds a reference).

Here's where Gems that sleep in transactions cause trouble. Without committing or aborting, they do not vote. The vote cannot be finalized, garbage collection halts at this point, and commit records accumulate.

5. But what about Gems that aren't on the system now, but were when garbage collection started? Their modified objects are in the commit record backlog, in the write sets of each commit record, which the GcGem reads in order to vote on their behalf.

While doing so, the GcGem is unavailable for its other functions.

6. The resulting set now holds nothing but unreferenced objects. If some of those objects are compiled methods or indexed collections, however, additional work must be done. The GcGem now hunts through the set of possible dead looking for those special cases.
7. The resulting objects are now deemed dead.
8. Pages are reclaimed. Repository shrinkage is now visible.
9. Page identifiers and object identifiers are returned to the free pools.

## Step-by-step Tuning

Each step in garbage collection presents a different opportunity for optimization.

### Step 1. Identify live objects.

Step 1, identifying all the live objects—the *mark* phase—is where `markForCollection` takes most of its time.

## Factors affecting duration

size of markForCollection Gem's mark/sweep buffer	larger is faster
number of objects in repository	fewer is faster
maximum disk I/O speed	faster is faster (surprise!)
shared page cache size	larger is faster
ratio of object table size to shared page cache size	faster if entire object table fits in shared page cache
number of other Gems sharing the shared page cache	fewer is faster
markForCollection Gem's private page cache size	larger is faster
markForCollection Gem's GEM_IO_LIMIT	larger is faster

## Statistics

### ProgressCount

During this step, progress count displays the number of objects marked as live. The final value, at this step, is the total number of live objects.

### Tunable parameters

size of markForCollection Gem's mark/sweep buffer	increase this
shared page cache size	larger is faster
ratio of object table size to shared page cache size	faster if entire object table fits in shared page cache
markForCollection Gem's private page cache size	increase this
markForCollection Gem's GEM_IO_LIMIT	depends on system goals
System's STN_GEM_ABORT_TIMEOUT	increase this if you increase GEM_IO_LIMIT
System's DBF_ALLOCATION_MODE	equally weighted for multiple extents

**Step 2. Compute possible dead set.****Factors affecting duration**

number of possible dead objects in repository	fewer is faster
the largest object identifier, known as the <i>OOP high water mark</i>	lower is faster
ratio of object table size to shared page cache size	faster if entire object table fits in shared page cache
markForCollection Gem's GEM_IO_LIMIT	larger is faster

**Statistics****ProgressCount**

During this step, progress count displays the number of objects identified as possibly dead. The final value, at this step, is the total number of possibly dead objects.

**Tunable parameters**

shared page cache size

This step goes faster if the entire object table fits into the shared page cache. To compute the approximate size of the object table, evaluate:

```
(System _oopHighWaterMark // 4) * 6
```

**Step 3. Return possible dead set to Stone.**

This step happens quickly and needs no tuning.

**Statistics****PossibleDeadSize**

The Stone's set of possible dead objects. This value is only approximate.

**Step . Logged-in Gems vote.****Factors affecting duration**

average transaction length	shorter is faster
the size of the possible dead set	smaller is faster
number of Gems logged in during vote	fewer is faster

network bandwidth between Stone's host and remote Gems' host	more is faster
number of objects in notConnectedSet written to disk	fewer is faster

## Statistics

### VoteNotDead

The number of objects in the possible dead set for which a given Gem holds a reference.

## Tunable parameters

Transaction length—voting completes faster if Gems run short transactions, as they vote upon commit or abort.

## Step 5. GcGem votes for logged-out Gems.

### Factors affecting duration

number of objects in the commit record backlog	fewer is faster
the size of the possible dead set	smaller is faster
maximum disk I/O rate	faster is faster
size of the shared page cache	larger is faster
the GcGem's GEM_IO_LIMIT	faster is faster

## Statistics

### ProgressCount

In this step, how far through the write-set union the GcGem has swept. This value peaks at GcPossibleDeadWsUnionSize and falls back to 0.

### GcPossibleDeadWsUnionSize

The size of the write-set union the Stone holds, which the GcGem must search on behalf of Gems no longer logged in.

### GcPossibleDeadSize

The exact number of possible dead objects after voting.

### GcSweepCount

The number of times this GcGem has performed this step since it was started.

## Tunable parameters

None.

### Step 6. GcGem hunts for special cases.

#### Factors affecting duration

number of dead nosequenceable collections with indexes	fewer is faster
the size of the possible dead set	smaller is faster
maximum disk I/O rate	faster is faster
size of the shared page cache	larger is faster
GcGem's GEM_IO_LIMIT	faster is faster

#### Statistics

##### ProgressCount

In this step, how far through the possible dead set the GcGem has swept. This value peaks at GcPossibleDeadSize.

#### Tunable parameters

- shared page cache size  
This step goes faster if the entire object table fits into the shared page cache. To compute the approximate size of the object table, evaluate:  
 $(\text{System\_oopHighWaterMark} // 4) * 6$
- The GcGem's GEM\_IO\_LIMIT  
Reduce this only if evaluation reveals that the GcGem is degrading system-wide performance due to disk I/O. Other functions of the GcGem will also be affected.

### Step 7. The possibly dead are now dead.

This step happens quickly and needs no tuning.

#### Statistics

##### PossibleDeadSize and GcPossibleDeadSize

These values fall to zero during this step.

**DeadNotReclaimedSize**

The number of objects finally in the possible dead set.

**Step 8. Pages and object IDs are reclaimed.**

This can be the most time-consuming step. Furthermore, the GcGem performs this while in a transaction, because reclaimed pages and object identifiers must be committed.

**Factors affecting duration**

presence and number of dedicated page-reclaim GcGems	Though they cannot be run with the generic GcGem, scheduling dedicated page-reclaim GcGems to run intermittently can significantly reduce the number of pages needing to be reclaimed,
GcGem's reclaimMaxPages and reclaimMinPages settings	Controls number of pages reclaimed per commit; larger batches mean longer transactions, more pages reclaimed, less other work done.
GcGem's epochGcEnabled setting GcGem's epochTimeLimit setting	If the GcGem has to spend time running epochs, it can spend less time reclaiming pages.
GcGem's FREE_FRAME_LIMIT	lower is faster
GcGem's GEM_IO_LIMIT	faster is faster
GcGem's private page cache size	larger is faster
size of the shared page cache	larger is faster
ratio of object table size to shared page cache size	faster if entire object table fits in shared page cache
number of shadow objects	GcGem reclaims shadow pages before reclaiming pages with dead objects.
number of free frames on free frame list	more is faster
size of commit record backlog	smaller is faster

## Statistics

### **DeadNotReclaimedSize**

The number of objects in the Stone's list of objects ready to be reclaimed. This value should decrease as dead objects are reclaimed.

### **PagesNotReclaimedSize**

The number of pages in the Stone's list of pages ready to be reclaimed. This value should decrease as pages with dead objects are reclaimed.

### **GcPagesNeedReclaiming**

The number of pages in the Stone's list of pages ready to be reclaimed, and the number of dirty pages still trapped in undeleted commit records.

### **GcReclaimNewDataPagesCount**

The number of pages the GcGem has reclaimed so far during this step.

### **FreeFrameCount**

The number of free frames on the free frame list. This value should increase.

## Tunable parameters

- The number of dedicated page-reclaim GcGems  
Consider scheduling these regularly if page reclamation is consistently behind. See "GcGems Specialized to Reclaim Pages" on page 10-30 for details.
- STN\_DEAD\_X\_LOCKING\_ENABLED  
This system configuration parameter is by default set to true, meaning that all objects identified as dead in the previous step are added to the Stone's exclusive lock set. This prevents any other Gem from committing a transaction that modifies them. If the set is large—for example, tens of millions of objects—this can degrade performance for all Gems, as they must search the set before committing any transaction.

If you trace a performance problem to a large number of deadNotReclaimed objects, and you are certain that your application never locates objects directly by object identifier, then set this parameter to false and see if performance improves.

### *CAUTION*

*With STN\_DEAD\_X\_LOCKING\_ENABLED set to false, it becomes possible to commit a reference to a dead object, thus corrupting your database. Modify this setting only if you are sure your application never locates objects directly by object identifier.*

- GcGem's reclaimMaxPages setting (in GcUser's UserGlobals)  
The GcGem commits its page reclamation transaction when it has reclaimed reclaimMaxPages, or when no more pages need reclaiming. The GcGem must perform a lot of work before the transaction can be committed; therefore, large values can slow the system for other Gems. However, a small value could mean that the GcGem falls behind in page reclamation and the repository grows. Typical values for large databases are between 500 and 3000.
- GcGem's GEM\_FREE\_FRAME\_LIMIT (in GcGem's configuration file)  
Gems have free frame limits to ensure that the Stone never runs out of free frames, because if it does, performance suffers throughout the system. If evaluation indicates that performance is slow because the GcGem has to scan the cache for free frames, consider adjusting this value downward cautiously, monitoring to ensure that the free frame list never falls to zero.
- GcGem's epochGcEnabled and epochTimeLimit settings  
If page reclamation is falling behind, consider temporarily disabling epoch garbage collection, or running it less often.
- The GcGem's GEM\_IO\_LIMIT  
Consider loosening the limit, or giving the GcGem access to unlimited I/O operations, if page reclaim is falling behind. You may wish to tighten the limit if page reclamation is slowing other Gem's unacceptably.
- The size of the GcGem's private page cache  
Consider enlarging it if the GcGem is spending a lot of time hunting for free frames.

## Step 9. Page IDs and object IDs are returned to free pools.

For the most part, this step happens quickly and needs no tuning; however, difficulties can occur if a large commit record backlog builds up. Reclaimed space isn't actually available until the Stone disposes of the commit record for the page reclamation transaction. If another Gem is holding on to an older commit record, this can't occur.

### Statistics

#### CommitRecordCount

Space occupied by reclaimed pages is not freed until the commit record of the page reclaim transaction is disposed of. A commit record backlog at this step will therefore delay the return of free space.

**DeadObjsCount**

The number of object identifiers returned to the free pool since the Stone was started. This value should increase.

**FreePages**

The number of free pages in the repository. This value increases when the page reclaim commit record is disposed of.

**ReclaimCount**

The number of page reclaim transactions the GcGem has performed since it was started.

**ReclaimedPagesCount**

The number of pages the GcGem has reclaimed since it was started. This value should increase.

## Three Examples

Three examples below tune a database for:

- the fastest garbage collection,
- the least disruptive garbage collection, and
- the best compromise between these two somewhat conflicting goals.

All refer to the system described in Table 11.1:

**Table 11.1 Example Database for Garbage Collection Tuning**

GemStone version:	5.1.4
OOP high water mark:	300,000,000
CPUs:	4
RAM (physical):	2 GB
repository size:	9.5 GB
disk drives:	6
extents:	6, one per dedicated disk even extent allocation
maximum extent:	2 GB
shared page cache:	768,000 KB 750 MB 96,000 pages
average free frames:	4800
minimum free frames:	3,000
free frame limit (Gems)	default: 9600
free frame limit (GcGem)	default: 4800 <b>new: 2160 = 120% of (average - minimum)</b>
object table:	450 MB
object table / shared page cache:	1.667

*NOTE*

*All three examples use a free frame limit for the GcGem that is lower than the default.*

*LEGEND*

*mfcGem = the Gem running markForCollection.*

## Example 1. Faster

The following example tunes the example database for the fastest possible garbage collection cycle.

**Table 11.2 Tuned for Fast Garbage Collection**

Parameter	Where to change it	Setting	Comments
#mfcGcPageBufSize	mfcGem UserGlobals	3000	Increase size of mark/sweep buffer
GEM_PRIVATE_PAGE_CACHE_KB	mfcGem config file	65536	maximum
GEM_IO_LIMIT	mfcGem config file	5000	limited only by host file system performance
STN_GEM_ABORT_TIMEOUT	system config or runtime	15	Increase to long enough to accommodate markForCollection.
GEM_IO_LIMIT	GcUser UserGlobals	5000	limited only by host file system performance
epochGcEnabled	GcUser UserGlobals	false	Disable epochs while running markForCollection
reclaimMaxPages	GcUser UserGlobals	1000	Increase to reclaim more pages.
GEM_PRIVATE_PAGE_CACHE_KB	GcGem config file	65535	Increase to make page reclaim go faster.
GEM_FREE_FRAME_LIMIT	GcGem config file	2160	Decrease to make page reclaim go faster.

## Example 2. Nicer

The following example tunes the example database for the least possible disruption to other system users.

**Table 11.3 Tuned for Least Disruptive Garbage Collection**

Parameter	Where to change it	Setting	Comments
#mfcGcPageBufSize	mfcGem UserGlobals	3000	Increase size of mark/sweep buffer
GEM_PRIVATE_PAGE_CACHE_KB	mfcGem config file	65536	maximum
GEM_IO_LIMIT	mfcGem config file	50	Limit the number of I/O operations per second so that other users can access the file system.
STN_GEM_ABORT_TIMEOUT	system config or runtime	60	Increase to compensate for low GEM_IO_LIMIT.
GEM_IO_LIMIT	GcUser UserGlobals	100	Limit the number of I/O operations per second so that other users can access the file system.
epochGcEnabled	GcUser UserGlobals	true	Default. Allow epochs to run as usual.
reclaimMaxPages	GcUser UserGlobals	300	Decrease due to low I/O limit.
GEM_PRIVATE_PAGE_CACHE_KB	GcGem config file	200	default
GEM_FREE_FRAME_LIMIT	System config file	2160	Allow other Gems to take more free frames from free frame list.
FREE_FRAME_LIMIT	GcGem config file	9600	GcGem must scan cache for free frames more often than other Gems.

### Example 3. Fast enough, nice enough

The following example is a compromise between the goals of fast garbage collection and minimal disruption to system users.

**Table 11.4 Compromise Tuning for Garbage Collection**

Parameter	Where to change it	Setting	Comments
#mfcGcPageBufSize	mfcGem UserGlobals	3000	Increase size of mark/sweep buffer
GEM_PRIVATE_PAGE_CACHE_KB	mfcGem config file	65536	maximum
GEM_IO_LIMIT	mfcGem config file	5000	limited only by host file system performance
STN_GEM_ABORT_TIMEOUT	system config or runtime	15	Increase to long enough to accommodate markForCollection.
GEM_IO_LIMIT	GcUser UserGlobals	5000	limited only by host file system performance
epochGcEnabled	GcUser UserGlobals	true	Default. Allow epochs to run as usual.
reclaimMaxPages	GcUser UserGlobals	500	Intermediate value is a compromise between page reclaim and other Gems' work.
GEM_PRIVATE_PAGE_CACHE_KB	GcGem config file	65535	Increase to make page reclaim go faster.
GEM_FREE_FRAME_LIMIT	GcGem config file	2160	Decrease to make page reclaim go faster.

## Discussion

Two things are always true:

- The Gem running `markForCollection` always wants a larger mark/sweep buffer. To change it, for the Gem running `markForCollection`, evaluate:

```
UserGlobals at: #mfcGcPageBufSize put: newValue
```

- The Gem running `markForCollection` always wants the largest possible private page cache.

These patterns emerge:

- The `markForCollection` Gem's `GEM_IO_RATE` can be a bottleneck. If disk waits are a problem during `markForCollection`, increase it. Allow it to perform as many I/O operations per second as necessary.
- If you give the `markForCollection` Gem an unlimited `GEM_IO_RATE`, increase `STN_GEM_ABORT_TIMEOUT` to accommodate `markForCollection`, or a `SigAbort` and the following `ABORT_LOST_OT_ROOT` error may cause the `markForCollection` to fail, possibly wasting a great deal of time.
- The GcGem's `GEM_IO_RATE` can also be a bottleneck. If disk waits are a problem during voting or epoch garbage collection, increase it.
- Increasing the size of the GcGem's private page cache allows it to reclaim pages faster.
- So does decreasing the size of the GcGem's free frame limit.
- The GcGem's configuration parameter `reclaimMaxPages` is most sensitive to speed vs. niceness. The default is 200; in practice, for this example system, values of 300–1000 provide a wide range of behavior from fast but disruptive to users to slow but nice to users. experimentation determines that a value of 500 is the best compromise.
- Disabling epoch garbage collection while running `markForCollection` is not necessary.

—  
|