# Checkpointing Design in STLdb

The revised checkpointing has the checkpoint process focused on recognizing pages that have been modified since the last checkpoint. Tracking is handled by detecting writes using mprotect. The checkpoint algorithm acquires a point-in-time view of the whole region when there are no transactions in effect (by establishing an exclusive transaction lock for a moment.) This allows it to establish the modified pages that will now be written out. It then commences to do so, writing out new copies of the modified pages to establish the new checkpoint on disk.

The process of writing the modified pages must not overwrite the previous versions of those pages, until all modifications have been written out for a new checkpoint. So on disk, there is the version of the page from the last checkpoint, and a pending new version, which becomes the new standard copy at a point in time, when a checkpoint commits.

This concept has an advantage: it means that in a paged region, a modified region would be free to page out at any time. If a transaction wasn’t actually done with it, it would get pulled back in and finished before the next checkpoint committed.

# Extending the Concept to a paged shared region

To support a region in which the region itself is smaller than the full size of the database, you would use a paged pointer, in which all bits beyond the 12 LSB are actually a pageno, and the 12 LSB are the offset in the page.

Whenever a pointer is used, the associated page is located in the region, and paged in if necessary. Memory allocation can’t cross page boundaries, because there’s no guarantee that pages remain sequential when brought into shared memory. For this reason, a “page” in this case might be larger than a disk pagesize. i.e. Some reasonable chunk size that the initial free space can be divided into, and which forms the smallest sized chunk for paging of data.

Once a page is brought in, it is protected, and writes are detected via mprotect.

The pages are managed in an LRU list. Whenever one needs to be swapped out to make room for a demand read, if it is unmodified it can be discarded. If it has been modified, it must be written out without overwriting the last checkpointed copy of that page. When paging a page in, it’s most recent write (if any) is brought in.

The checkpoint algorithm is roughly the same as current – get all modified pages still in the region out to disk as new copies of those pages, based on a point-in-time view of the modifications. Then the older versions of those pages become free space.

Logging, under this approach, can remain the application of logical operations to the point-in-time view of the database that is the last checkpoint.

# Dropping use of the Signal Handling Approach:

This approach shifts the responsibility for signaling that a page write is about to occur to the container code, including to the free list manager of the region. All modifications of some portion of the model must be signaled ahead of time so that page modifications can be tracked, and the need to write the page to an in-progress checkpoint can be recognized.

I.e. Instead of the signal handler detecting this, and lifting the protection, an explicit API is used for the same.

The use of mprotect() can be employed in debug mode to help detect modifications of memory which are not anticipated.

Has an impact on the design of containers which seems unavoidable.