# Setting up this project

If you are seeing this, then you’ve probably already pulled the code from SVN. If not, you can do this in eclipse by using the ‘subversive’ plugin. Then switch to the ‘SVN RepositoryExploring’ perspective, and create a new SVN repository site using the URL: <https://stldb.svn.sourceforge.net/svnroot/stldb>. Provide the user\_id and password you use with sourceforge. When you browse the repository, you will then see two directories, stldb\_lib and stldb\_test under the root. Check out each of these two directories

I’m currently building & running STLdb using the eclipse CDT environment, using the GNU compiler toolchain. On windows, I’m using cygwin specifically. On MacOS/Linux, the std gcc compiler and gnu make which comes with the OS does the job.

## Windows/Cygwin setup:

1. Install cygwin first (e.g. C:/cygwin) and make sure that your installation includes the gcc compiler, as well as gnu make, tar, sh, etc.
2. Add CDT and subversive to eclipse.
3. Check out the stldb\_lib and stldb\_tests directories as two separate CDT projects.
4. Download Boost version 1\_39 from boost.org. Untar it somewhere (it doesn’t have to be under your CDT workspace.) I have put this in some ‘project’ directory, and you can then always make a symbolic link to it within an eclipse project.
5. Build boost by following its installation instructions:
   1. cd boost\_1\_39\_0
   2. ./bootstrap.sh
   3. ./bjam install

# Implications of row-level locking on container API:

With a normal std::map, if I use the container within a multi-threaded context, I need to lock it, and boost::interprocess::scoped\_lock provides an easy means to do that. All manipulation of the map then becomes exclusive for the duration of those locks.

The idea of a transactional map (or at least one supporting good concurrency) would follow a similar principle in that a lock must be held for the duration of the individual API calls. However it also introduces the idea of longer held locks which last until the transaction is resolved (committed or rolled back). These row level locks (more accurately called entry-level locks) are more granular but also longer-lived.

The problem which arises is what to do when the application invokes a method that ateempts to manipulate a row that already has a entry-level lock in place due to another unresolved transaction. In normal database design, the caller blocks until the transaction is resolved, in which case the caller can then proceed. To do this under the conventions of the boost::interprocess library, I need to have access to the lock, the specific scoped\_lock object held by the caller, so that it can be used to wait on a condition variable which signals the release of the row-level lock. Without the lock being available, the only other option of the API is to throw up an exception indicating ‘row\_locked\_exception’, and requiring that the caller than perform the condition variable wait themselves.

So there’s an API design choice here:

1. An API in which the lock held by the caller remains unknown to the container’s primitive methods, but, in the process the container must throw or otherwise signal the need to wait for an entry-level lock release in order to succeed with the operation, or
2. An API in which the caller must use a scoped lock to get a lock on the container, and must then pass that lock to any method which might be required to do a cond\_wait because of the entry-level locking implementation.

I’m going with #2 for two reasons:

1. It is easier on the caller than a solution where they must constantly remember to handle a particular exception or indicative return value.
2. It clearly identifies (by implication) which APIs of the container could block due to the implementation of entry-level locking.
3. It reinforces the need to have a lock when calling those methods.

## Transaction and Scoped Locks

Idea: integrating the process of acquiring container locks into the transaction class somehow. Perhaps having the transaction acquire and track the locks against the container by using the container\_proxy class’es lock methods.

Concern: The transaction would end up hiding the Boost.interprocess scoped locking classes, eliminating the advantage of that approach.

# txn\_id vs Log\_seq\_id:

txn\_id:

When a transaction first begins, it is assigned a txn\_id (alias lock\_id) which it can use to mark in progress changes on containers as belonging to that txn\_id. However, transactions occur in parallel, so while the assignment of txn\_id values to new transactions may be ascending, when comparing two values for a container entry (a row of data) you can’t tell which container entry is current based on the txn\_id. Txn\_id 100 did not necessarily happen after txn\_id 99. It all depends on when those transactions actually committed.

When a nearly completed transaction is enqued for its commit buffer write, the transaction is assigned a log\_seq at that time. This is a separate ascending sequential value. Unlike txn\_id, it is purely sequential. Commit buffers in the logs are written in ascending order of log\_seq\_id. It is possible, at any given point in time, for the database to know:

* The last (highest) enqueued log\_seq\_id
* The last (highest) written log\_seq\_id
* The last (highest) written log\_seq\_id for which fsync() can returned.

In light of these two values, how does checkpointing, and incremental checkpointing work?

# Full Checkpointing:

When the checkpoint algorithm starts, it can get the database’s current highest sync’ed log\_seq\_id. This is then used to mark the point in time of the checkpoint. It is incorporated into the filename. The algorithm then walks over the contents of the container, one chunk at a time, and writes out all committed data seen.

For map, this means:

* If the row has Insert\_op, skip it, and don’t write it out.
* If the row has any other value on it, write it out. So we write only the

## Recovering a Container:

1. Search all checkpoints for the latest successfully written checkpoint of the container.
2. Load that checkpoint.
3. Start reading log records starting with the one that is just after the log\_seq that the checkpoint was created for. Continue applying them until EOF.