NOTE 256 –Casacore Table Locking

Ger van Diepen, ASTRON Dwingeloo

2020 March 3

Abstract

The Casacore Table Data System (CTDS) supports single writer and multiple readers by means of table locking (which uses the OS file locking). However, if the OS has no or limited support of file locking, it is possible to disable table locking.

1.0	2003 Feb 10	Original version
2.0	2020 Mar 3	Describe nolocking modes

1 Introduction

The Casacore Table Data System (CTDS) is used to store all data in the Casacore environment. CTDS supports concurrent access by means of table locking. The user can control if and how the table locking is used which may have impact on the performance of the Table System. This note describes the various ways in which a table can be locked and the best ways to use them.

2 Locking and Synchronization

Before a table can be read or written, an appropriate lock has to be acquired. As in most systems there are two types of locks:

- A read lock (also called shared lock) is used to grant a process read-access to a table. At a given time multiple processes can have a read lock on the same table (hence shared lock).
- A write lock (also called exclusive lock) is used to grant a process write-access to a table. At a given time only one process can have a write lock on a table (hence exclusive). If a write lock is acquired, no

read lock on that table can exist at the same time. Thus at a single time there can be one writer or multiple readers.

The locking assures that the internal buffers of CTDS are synchronized and kept consistent.

Note, however, that CTDS also supports modes to skip table locking which can be useful for systems with no or limited support of file locking (such as Lustre).

- CTDS can be built with -DAIPS_TABLE_NOLOCKING which disables table locking entirely.
- Similarly, setting the *aipsrc* variable `table.nolocking` to True disables table locking.
- Table lock option NoLocking can be used to disable locking for the table opened with that option.
- Table lock option NoReadLocking can be used to read a table without locking. That implies that no synchronization is done before the read, unless it is explicitly done. It is described in a later section.

Of course, disabling locking should be used with great care. One has to be sure that no race conditions can occur.

CTDS only supports locks on the entire table; there is no fine-grained page or row locking. Thus if a process is updating a table, no other process can read or update any part of that table. Of course, different tables can be accessed simultaneously.

Basically there are two functions in CTDS to handle locking.

- `Table::lock` can be used to acquire a read or write lock. Note that a write lock also grants read-access. Internal data buffers will be refreshed as needed.
- `Table::unlock` can be used to release a lock. It will flush the internal data buffers to disk if they are changed.

2.1 Synchronization

CTDS uses a special lock file (`table.lock`) in the table directory to hold information about the storage managers containing data changed in a lock/unlock cycle. That information is used to achieve that a process acquiring a lock only refreshes the buffers really needed. This saves a lot of needless buffer refreshing.

Apart from changing data, a table can also change by adding or removing rows, keywords, or columns. These changes are also synchronized with the exception of added or removed columns. Such changes mean that the layout of the table is changed. Alas the Table System cannot cope with such changes yet (it will in a future version). If it detects that a column is added or removed, the synchronization function throws an exception.

CTDS uses the file locking functions provided by the operating system to do the actual locking. For UNIX systems it means that the `fcntl` function is used. This also works fine for NFS files provided that the `lockd` and `statd` deamons are running (which is usually the case).

The first few bytes of the lock file form the data part that is being used for the locking.
Furthermore an extra lock is used to keep track of the tables that are open in any process. This is used by CTDS to prevent a table from being deleted while another process is still accessing it.

Note that on a UNIX system file locks are handled per inode. That means that if the same file is opened twice an unlock on one the the file descriptors also unlocks the other file descriptor. CTDS usually opens a table only once. However, if opened with a different name, CTDS might not recognize that the file bas been opened already and open it again in the same process. However, in practice it never happens.

3 Locking Modes

Data base systems usually hold locks only for a short period of time. It means that buffers have to be flushed often because another process needs up-to-date data. A flush can be expensive as it involves I/O.
It was believed that this would be too limiting for CTDS, because many processes prefer to lock a table for as long as possible to avoid too many flushes. Therefore three locking modes are supported by Casacore. They can be given when a `Table` object is constructed. In order high to low they are:

1. `PermanentLocking` locks the table for the full session, so no other process can use it. It is always locked for read. It is locked for write if the table is not opened as readonly.

This mode is rarely used, because it makes concurrent access impossible (unless another process uses NoReadLocking). However, in some circumstances it might be useful. Note that in this mode lock and unlock calls can still be done, but they are not doing anything.

2. `AutoLocking` locks automatically when a read or write needs to be done and the table does not have an appropriate lock yet. Unlocking is done automatically at an appropriate time.
   This mode makes concurrent access possible and tries to hold the lock as long as possible. Furthermore the programmer does not have to worry about (un)locking, although it is still possible to do explicit locking and unlocking.
   The problem with this mode is that automatic unlocking can only be done when possible. Therefore it could happen that a table is locked for a longer period than expected. It is discussed in more detail in a later section.

3. `AutoNoReadLocking` is the same as AutoLocking, but no locks are needed when reading the table. NoReadLocking is discussed in another section.

4. `UserLocking` means that locking and unlocking have to be done explicitly.
   This mode gives the finest control, but it it may be hard for the programmer to decide how fine-grained it should be used. On one hand not too fine, because it involves a lot of flushing. On the other hand fine enough to give other processes the opportunity to grab a lock. An exception is thrown if a read or write is done before an appropriate lock is acquired.

5. `UserNoReadLocking` is the same as `UserLocking`, but no locks are needed when reading the table. NoReadLocking is discussed in another section.

6. `DefaultLocking` is the same as `AutoLocking`. It is lower in the hierarchy when merging locking modes (see next section).

7. `NoLocking` disables read and write locking for the given table. in the hierarchy when merging locking modes (see next section).

The default locking mode is `DefaultLocking`.

3.1 Locking Mode Merge

It is possible that in a process multiple `Table` objects are created for the same table. These `Table` objects share the same underlying `BaseTable` object doing the actual locking. Since each `Table` object can be created with its own locking mode, those locking modes have to be merged in the `BaseTable` object. The merge result is the locking mode with the highest position in the locking mode table.

3.2 Locking Mode for a Subtable

`Table` objects for subtables are created automatically when the table is retrieved from a keyword set like:

```
Table tab(``test.ms'');
Table anttab(tab.keywordSet().asTable(``ANTENNA''));
```

In this way it inherits the locking mode of its parent `(Base)Table` object. However, there is a second `asTable` function accepting a locking mode. In that way a subtable can be opened with a given locking mode.

4 How to lock and unlock

As explained above locking and unlocking has to be done explicitly for User-Locking and can optionally be done for AutoLocking.

Function `Table::lock` is used to acquire a lock. It can be specified if a read or write lock is needed. If the lock cannot be acquired immediately, the process will be added to a list (in the lock file) to indicate that it needs a lock. The list is used by AutoLocking (see below). Thereafter it will try again to acquire the lock until `nattempts` is reached. It sleeps a little while between each attempt. After 30 attempts a message is sent to the logger telling that the process is waiting for a table lock. If the lock could be acquired, the process is removed from the list.
Having acquired a lock means that the internal table and storage manager buffers are refreshed as needed.

Function `Table::unlock` releases a lock. If table data have been changed since the lock was acquired, it will flush the changed table and storage manager buffers and indicate in the lock file which storage managers were changed.

4.1 class TableLocker

It should be clear that when using lock/unlock explicitly, care should be taken that the unlock is also done in case of exceptions. This can be quite cumbersome. Therefore the class `TableLocker` has been created. Its constructor acquires a lock, while the destructor releases it. C++ scoping can be used to invoke the destructor automatically. E.g.

```
Table tab(``test.ms'', Table::UserLocking);
{
  TableLocker locker(tab, FileLocker::Read);  // acquire read lock
  ... write data into the table ...
}  // end of scope, so TableLocker destructor is called
```

The nicest thing of using `TableLocker` in this way is that in case of an exception its destructor is called, thus the lock is always released.

5 AutoLocking working

AutoLocking is a handy mode because it frees the user from having to do explicit locking and unlocking. Furthermore it has the advantage that it does not release a lock before another process needs it. This advantage is at the same time the weakness of AutoLocking, because it may take a while before a process holding a lock recognizes that another process needs a lock. For this to understand it is explained how AutoLocking works.

1. When table I/O is done, it is checked if the table is appropriately locked. If not, it is tried to acquire a lock with `TableLock::maxWait` as the maximum number of attempts (default is trying forever).

2. Unlocking is also done automatically. After some I/O-s are done, CTDS inspects the list in the lock file to see if another process needs the lock. If so, it releases the lock. The inspection interval can be defined in the `TableLock` constructor and defaults to 5 seconds.

In general this scheme works fine, but the problem is that if no I/O is done, CTDS does not check if another process needs a lock. This is especially a problem for glish clients as they can be idle for some time waiting for a command to be given. To circumvent this problem the function `Table::relinquishAutoLocks` can be used to release locks on tables using AutoLocking. Either all such tables are unlocked or only the tables needed by another process.

5.1 Releasing AutoLocks in Glish Clients

Care has been taken that glish clients do not hold AutoLocks too long. Glish clients time out after some period and call `Table::relinquishAutoLocks` to release AutoLocks at regular intervals. The time out period is controlled by two aipsrc variables. They are:

- `table.relinquish.reqautolocks.interval` defines the number of seconds to wait before relinquishing autolocks requested in another process. The default is 5 seconds.
- `table.relinquish.allautolocks.interval` defines the number of seconds to wait before relinquishing all autolocks. The default is 60 seconds.

The user can define these variables at will, but usually the defaults suffice.

6 NoReadLocking

Normally locking should be used to read data from a table because in that way it is assured that the data is always consistent. However, in some cases it might be useful to be able to read data from a table without having to acquire a lock. That could, for instance, be the case for an online process filling a MeasurementSet. It should not happen that such a process has to wait for a write lock because some other process is holding a read lock. NoReadLocking is possible with AutoLocking and UserLocking modes.

The NoReadLocking option makes it possible to read table data without acquiring a read lock. It means that the internal buffers are not automatically synchronized with the data on disk. It is possible though to do that explicitly using the function `Table::resync`. For this to work well, the writer should flush its data regularly, otherwise it may take a long while before data appears on disk.

Often a NoReadLocking reader is coupled to the writer by means of interprocess communication. In such cases it is best that the writer tells the reader when it should resync and read.

7 Lock Information

Sometimes it is not clear which process is holding a lock. A glish function in `os.g` has been made to tell the user if a table has been opened in some process and if and how it has been locked.

```
  dos.showtableuse ('test.ms')
```

prints this information for the given table. The function `dos.lockinfo` returns this information in a glish record.

The information contains the PID of a process holding the lock or having opened the table. Note that in case of a read lock multiple processes may hold a lock. The PID of only one process is given in the information though.

8 Overview of classes/functions related to locking

A brief overview is given. For detailed information the relevant documentation should be examined.

8.1 class TableLock

This class defines the locking mode which can be given to the `Table` constructor. For AutoLocking mode a few parameters can be set.

8.2 class TableLocker

This class can be used to acquire and release a lock, especially in UserLocking mode. As described above, it is particularly useful to ensure that a lock is released in case of an exception.

8.3 Table functions

- `Table` constructor accepts a `TableLock` argument.
- `lock` tries to acquire a read or write lock and resync-s..
- `unlock` releases the lock and flushes the table buffers.
- `hasLock` tests if the table holds a read or write lock.
- `canLock` tests if the table can acquire a read or write lock.
- `flush` flushes the table buffers.
- `resync` resync-s the table buffers with the data on disk.
- `hasDataChanged` tests if the table has changed since the last time this function was called.
- `lockOptions` gets the current locking mode.

- `isMultiUsed` tests if the table is used in another process.
- `nAutoLocks` is a static function returning number of tables using AutoLocking.
- `relinquishAutoLocks` is a static function releasing some or all AutoLocks.

8.4 python and glish

pyrap contains the python interface `pyrap.tables` to CTDS. It has lock functions similar to the ones in class Table.

The same is true for the old glish interface `table.g`
In `os.g` the functions `showtableuse` and `lockinfo` give information about table locking.