FairCom White Paper

Memory Files

Audience

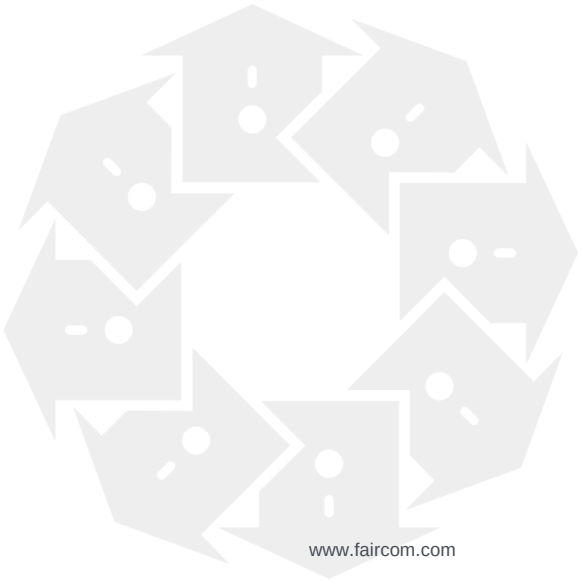
Developers

Subject

FairCom's high-performance NAV and SQL database technology.

FairCom®

Contents



1. Memory Files Overview

The FairCom DB Server supports pure memory-resident data and index files. The distinguishing characteristic of memory files is that they exist solely in memory, occupying their own memory space separate from the data and index caches.

Memory files satisfy the need for the creation and manipulation of temporary data or index files that are always memory resident and never touch disk. Contrast this with non-memory files, whose contents may be paged out of the database cache and written to disk. Memory files are ideal for true temporary files: they can exist in memory as long as the FairCom DB Server or c-tree application process is running.

Some applications of memory files include:

- **Temporary files**: Temporary files can be created as memory data or index files. The file contents are always memory resident, and when the memory file is finally closed, it ceases to exist.
- **Storing read-only file contents**: At server or application startup, the contents of a disk file can be read into a memory file and subsequent read requests can be satisfied from the memory file.
- **In-memory list management**: Applications frequently need to maintain in-memory lists. A memory index provides easy creation and manipulation of items in a B+-tree-indexed list using the c-tree API, rather than requiring the developer to implement custom list management routines.

Using Memory Files

The main operational differences between memory files and non-memory files involve creating and closing the files. After memory files are created, they are accessed using the standard c-tree API functions, just as non-memory files.

Published 4/15/2021

1.1 HUGE File Support

As memory files use the native memory address size associated with their underlying platform architecture, they must match the attributes of that platform. Importantly, a memory file on a 64-bit OS requires 64-bit offset support. (Contrast that to filesystem support which is separate and independent of memory architecture support.)

Memory files on 32-bit OS platforms must use 32-bit file offsets (and *cannot* be HUGE).

Memory files on 64-bit OS platforms must use 64-bit file offsets (and *must* have **HUGE** file support).

When using memory files on a 64-bit server: If you create any indexes that allow duplicates, be sure to include 8 additional bytes in the total key length (rather than the typical 4 bytes) for the record offset indicator. This will avoid potentially triggering a **582 error**. **115** errors can also be observed as key segment lengths do not match the assigned key length.

1.2 Creating Memory Files Using Server Configuration Keyword

The FairCom DB Server supports creating memory files using a server configuration keyword. This feature allows developers to create memory files using their existing application code, provided that the file is created using an *Xtd8* create function such as **CreateIFileXtd8()**, see *Xtd8 File Creation Functions* (*http://docs.faircom.com/doc/ctreeplus/30545.htm*). To create a memory file using the server configuration keyword, specify one or more entries of the form:

```
MEMORY_FILE <file name>#<max size>
```

where the file name may include wild card characters, and the maximum size is optional. If no maximum size is specified, then 4GB is used. If a file is being created and matches one of the `MEMORY_FILE` file name entries, then it will be created as a memory file unless it is a superfile host, superfile member, mirrored, segmented or partitioned file.

To cause all possible files to be created as memory files, add the following configuration entry:

```
MEMORY_FILE *
```

The `MEMORY_FILE` keyword is useful to quickly test how a file or set of files will behave as memory files.

1.3 Creating Memory Files Programmatically

To create a memory file programmatically, include the *ctMEMFILE* attribute in the *x8mode* member of the file's *XCREblk* structure (see *Extended File Creation Block Structure* (*http://docs.faircom.com/doc/ctreeplus/30571.htm*)), and specify the maximum file size using the *mxfilzhw* and *mxfilzlw* members of the *XCREblk*.

A temporary, memory-resident index file can be created independent of any data file. Simply call **CreateIndexFileXtd8()** (**https://docs.faircom.com/doc/ctreeplus/createindexfilextd8.htm**) with an *XCREblk* set as described above.

See the pseudo-code below to create a memory file:

```
XCREblk xcreblk[] = {
      {ctMEMFILE, 0, 0, 104857600, 0,0,0,0,0,0, ctKEEPOPEN, 0,0,0,0,0},
      {ctMEMFILE, 0, 0, 104857600, 0,0,0,0,0,0, ctKEEPOPEN, 0,0,0,0,0}
   };
...
   if (CreateIFileXtd8(&vcustomer,NULL,NULL,0,NULL,NULL,xcreblk))
   {
      ctrt_printf("\nCould not create file %d with error %d.\n",isam_fil,isam_err);
   }
```

Note: Memory files can not have *ctTRNLOG* mode applied or **CreateIFileXtd8()** will fail with error 749 (bad parameter). Be sure to adjust any *IFIL* definitions.

1.4 Sharing Memory Files Created Programmatically

There is a subtle issue about how to get a memory-resident file to be a shared file. At create, it is exclusive. If it is created and immediately closed, to be reopened for shared access, it would disappear on the close, and the reopen would fail. You can use the **UpdateFileMode()** c-tree API function to change the file mode from exclusive to shared:

```
CreateIFileXtd8(...);
UpdateFileMode(datno,ctSHARED);
UpdateFileMode(keyno_1,ctSHARED);
...
UpdateFileMode(keyno_n,ctSHARED);
```

The **UpdateFileMode()** call will fail if there are transaction-dependent actions pending. For example, if the **CreateIFileXtd8()** call in the above pseudo code were for a transaction-dependent (*ctTRANDEP*) file, the **UpdateFileMode()** call could not be called until the create is committed.

As noted above, the final close of a memory file causes all the existing memory file contents (data records or index nodes) to be lost. A final close refers to a file close that causes the user count of the file to drop to zero. It is possible to make the data file persist even after the final close by including *ctKEEPOPEN* bit in the *splval* member of the data file's *XCREblk*. Then the final close leaves the file open (but without any user attached to the file.) It can then be opened again, and the data still exists. The file will be closed when the server terminates, or when a call is made to the **CloseCtFileByName()** c-tree API function:

```
CloseCtFileByName(pTEXT filnam,pTEXT fileword)
```

It is possible to use the *ctKEEPOPEN* flag on the create, and then close and reopen the file shared (without calling **UpdateFileMode()**), but only if the file's creation is not pending commit. In this case, the sequence would be like:

```
XCREblk xcreblk[] = {
      {ctMEMFILE, 0, 0, 104857600, 0,0,0,0,0,0, ctKEEPOPEN, 0,0,0,0,0},
      {ctMEMFILE, 0, 0, 104857600, 0,0,0,0,0,0, ctKEEPOPEN, 0,0,0,0,0}
   };
...
   if (CreateIFileXtd8(&vcustomer,NULL,NULL,0,NULL,NULL,xcreblk))
   {
      ctrt_printf("\nCould not create file %d with error %d.\n",
                  isam_fil,isam_err);
   }
   else
   {
      CloseIFile(&vcustomer);
      if ((eRet = OpenIFileXtd(&vcustomer,NULL,NULL,NULL)) != 0)
      {
         ctrt_printf("\nUNEXPECTED ERROR %d ON REOPEN.\n",eRet);
      }
      else
      {
         ctrt_printf("\nFile created in memory.");
      }
   }
```

1.5 Collecting Memory File Statistics

Memory file statistics are available through the FairCom DB command-line utility ctstat using the *-m* switch, discussed in the topic titled *ctstat - Statistics Utility* (*http://docs.faircom.com/doc/ctreeplus/52856.htm*).

Memory file statistics can also be collected using the **SnapShot()** function call discussed here in the *SnapShot* (*http://docs.faircom.com/doc/ctreeplus/snapshot.htm*) topic. See the details for the *ctGFMS* structure where the following memory file values are maintained:

- *memcnt*: current number of memory records
- *hghcnt*: highest number of memory records
- *phyrec*: current total memory allocated for memory records
- *mhghbyt*: highest amount of memory allocated for memory records

1.6 Memory File Limitations

Memory files are subject to the following limitations:

- A memory file cannot be mirrored, partitioned, or segmented.
- A memory file cannot be backed up by a dynamic dump.
- The ISAM rebuild function **RebuildIFile()** does not support rebuilding indexes associated with a memory data file; instead use the low-level **RebuildIIndex()** function to rebuild individual indexes.

- The restorable delete (*ctRSTRDEL*) attribute is not supported for memory files.
- The atomic, recoverable transaction control attribute (*ctTRNLOG*) is not supported for memory files. However, the atomic, non-recoverable transaction control (*ctPREIMG*) and the transaction-dependent (*ctTRANDEP*) attributes are supported for memory files.
- When traversing memory files in “physical order” there is no guarantee of any particular ordering. The sequence of **FirstRecord()**, **NextRecord()** operations will traverse records ordered by their internal hash table positioning, which can be quite different from a typical physical ordering on disk.
- Be sure to see *HUGE File Support (page 2) for additional requirements.*

1.7 Tip: Faster Server Shutdown with Memory Files

When the FairCom DB Server shuts down, it frees individual memory records for memory files that are still open. If large memory files are open, the freeing of the memory records can prolong the server shutdown time. Adding the `COMPATIBILITY MEMORY_FILE_SKIP_FREE` keyword to the server configuration file causes the server to skip the freeing of individual memory records. When the server process terminates, the operating system will automatically reclaim the memory allocated to memory file records.

2. Document History

03/20/2019	Added note about HUGE files. Updated for readability.

Last published Thursday, April 15, 2021.

Copyright Notice

Trademarks

Open Source Components

Like most software development companies, FairCom uses third-party components to provide some functionality within our technology. Often those third-party components are selected because they are a standard in the industry, they offer specific functionality that is easier to license than to develop and maintain in the long run, or they provide a proven and inexpensive solution to a particular business need. Examples of third-party software FairCom uses are the OpenSSL toolkit that provides Transport Layer Security (TLS) for secure communications and the ICU Unicode libraries to provide wide character support (think international characters and emojis).

Some of these third-party components are the subject to commercial licenses and others are subject to open source licenses. For open source solutions that we incorporate into our technology, we include the package name and associated license in a notice.txt file found in the same directory as the server.

The notice.txt file should always stay in the same directory as the server. This is particularly important in instances where your company has redistribution rights, such as an ISV who duplicates server binaries and (re)distributes those to an eventual end-user at a third-party company. Ensuring that the notice.txt file "travels with" the server binary is important to maintain third-party and FairCom license compliance.

4/15/2021

3. Index