FairCom White Paper

FairCom DB Data Compression

Audience

Developers

Subject

Understanding FairCom's Data Compression Technology

Contents

1. Data Compression

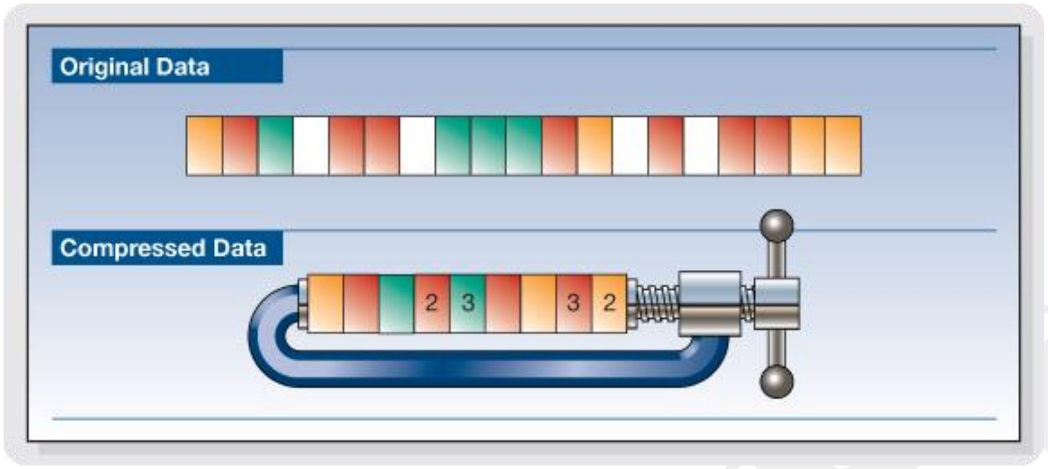
Data volumes are exploding, and, as a result, database file sizes are proportionately increasing. Managing large files remains an important parameter in sizing disk arrays, which can be quite expensive for high availability systems. Simply moving large data sets for archiving and backup purposes becomes a challenging task. Compressing data is a valuable technique to reduce this data storage challenge. By directly reducing the data record size, file sizes can be substantially reduced.

To enable support for additional file modes FairCom DB has implemented augmented variable-length records. The first supported augmented feature being data record compression.

Published 4/15/2021

1.1 Compressed Files

FairCom DB now supports Data Compression. Recent challenges of larger HUGE files, downsizing needs, and migration compatibility from other systems that support data compression, as well as specific customer requirements resulted in this core database necessity.



As low-level data records are written to and read from disk, FairCom DB now intervenes just before they are passed to the operating system's file system, and will “compress” before writing and “un-compress” after reading each data record.

The default compression algorithm comes from the standard zlib library, written by Jean-Loup Gailly and Mark Adler and is an abstraction of the DEFLATE compression algorithm used in their gzip file compression program.

FairCom DB also supports a proprietary run-length encoding (*RLE*) option; yet perhaps even more important is the ability for the user to implement a Call-Back where you can define your own compression technique.

1.2 Data Record Compression

As different files and applications may require various types of compression, a FairCom DB resource (*CMPRECRES*) has been defined and is embedded into files supporting data record compression. This resource contains the compression type, version, custom parameters, and the DLL name (if built-in compression is not used). The calls to the compression routines are handled by function pointers that are automatically initialized for the built-in routines and DLLs.

The compression resource structure supports optional parameters to fine tune the compression operations. For example, zlib has five parameters that control internal processing. The compression structure uses a *pVOID* pointer and a length parameter to permit different compression DLLs to support different parameter sets. **ctSETCOMPRESS()** allows an application to set these parameters.

See Also

- **ctSETCOMPRESS()** in the *FairCom DB Pro Developer's Guide*

1.3 Server Configuration

Default FairCom DB compression can be specified with the following configuration keywords:

```
CMPREC_TYPE          < "ZLIB" | "USER" >
CMPREC_VERSION                <a number >= 1>
CMPREC_DLL           <name of DLL>
```

These keywords should be entered in the configuration file in the order shown, and a DLL name is required for `CMPREC_TYPE` of USER.

A new COMPRESS_FILE keyword can be used to enable compression in files whose names match specified file names (including wildcards). See **COMPRESS_FILE**.

1.4 Compressed Files in c-treeDB

A new c-treeDB file create mode, *CTCREATE_COMPRESS*, has been introduced to enable compressed record support in c-treeDB. When this mode is used, c-treeDB automatically creates the file as variable length.

Notes

- Some files will compress better than others (text data vs. binary data for instance). Initial testing revealed up to an 80:1 compression ratio in some cases.
- At this time, only zlib file compression is available, however, support has been included for multiple compression routines in the future.

- This feature is available only for ISAM files fully maintained by ISAM updates. Low-level FairCom DB functions will return an error when used on files with the compression attribute enabled.

User Defined Compression Dynamic Shared Library

The FairCom DB data record compression feature supports user definable compression modules. To include a custom compression module, the following information must be made available from within the shared library.

Function Arguments

`pCMPRECFNC pfnc;` input pointer to function pointer/DLL structure defined in *ctstrc.h.*

`ppVOID pattrbuf;` on input, a pointer to optional, fine-tuning parameter buffer. **pattrbuf*==NULL implies no fine-tuning parameters specified. On output, **pattrbuf* points to a parameter buffer. **pattrbuf* is not guaranteed to be NULL on output if it is NULL on input, and vice versa.

`pVRLEN pattrlen;` on input, pointer to length of fine-tuning parameter buffer. If **pattrbuf* is NULL on input, **pattrlen* must be zero. On output **pattrlen* contains the length of the parameter buffer.

`pCMPRECRES pres;` input pointer to compression resource structure defined in *ctstrc.h.*

`pVOID context;` input pointer to optional, compression/decompression support structure. If the compression (*CmpActn*) and/or decompression (*ExpActn*) routines do not use a support structure, then context will be NULL or *((pVOID) 1)*.

`pVOID source;` input pointer to byte stream to be compressed or decompressed.

`VRLEN slen;` input length of source buffer.

`pVOID destination;` output pointer to buffer that will contain the result of compression/decompression.

`pVRLEN pdlen;` on input, **pdlen* is length of destination buffer. On output, **pdlen* is the length of the byte stream result from the compression/decompression.

Function Prototypes

The module *ctzlib.c* has implementations of these functions for the built-in zlib support.

FncInit

```
NINT FncInit(pCMPRECFNC pfnc,pTEXT attrstr,VRLEN attrlen)
```

This routine performs a one-time setup on each physical open of a file after the DLL has been loaded. *attrstr* and *attrlen* are the same as those in **ctSETCOMPRESS()**. In addition to any DLL specific requirements, **FncInit()** should verify that the compression type and version (*pfnc->comptype* and *pfnc->compvrsn*) are compatible with the DLL, and that the fine-tuning parameters, if any, are valid. If no fine-tuning parameters are passed in, then if the routines require a parameter set, **FncInit()** should allocate and initialize such a parameter buffer. If it does so, then it should set *pfnc->allocbuf* to the address of the allocated buffer.

FncInit() returns NO_ERROR or **CMPR_ERR** (). In the case of **CMPR_ERR** **FncInit()** should set the *sysiocod* (using the routine *usys(code)*) as listed in *cterrc.h* for **CMPR_ERR**. **FncInit()** does not actually set *uerr_cod*.

FncExit

```
NINT  FncExit(pCMPRECFNC pfnc )
```

FncExit() is called just before a file is closed and its associated DLL is unloaded. In addition to DLL specific requirements, **FncExit()** should free memory allocated for a parameter buffer. **FncExit()** can test *pfnc->allocbuf* to find the allocated buffer address, if any.

FncExit() should return NO_ERROR.

CmpInit

```
pVOID CmpInit(pCMPRECFNC pfnc)
```

CmpInit() is called each time a user opens a compressed data file. **CmpInit()** should allocate a context buffer if the DLL requires one, and initialize the context as needed. The context buffer would typically be used to initialize and maintain the compression state. If no such context buffer is needed, then return *((pVOID) 1)*.

CmpInit() returns the address of the context buffer, or *(pVOID) 1* if no context is used, or NULL on error. In case of error call *usys(35)* to indicate the **CmpInit()** failure.

CmpActn

```
NINT  CmpActn(pVOID context, pCMPRECFNC pfnc, pVOID source, VRLEN slen, pVOID destination, pVRLEN pdlen)
```

CmpActn() performs the actual compression of the source into the destination. *pdlen holds the length of the destination buffer on input, and is set to the length of the compressed output on return. CmpActn should be aware that an input value of *((pVOID) 1)* means that no context structure has been provided.

CmpActn() returns **CMPREC_OK** () on success, **CMPREC_BUF** () if the destination buffer is too small, or **CMPR_ERR** for an unexpected error. In case of an unexpected error, *usys(1000 + err)* should be called to set *sysiocod* where *err* is the internal error code used by the compression routines. (See **zlib1_CmpActn()** for an example.)

CmpExit

```
NINT  CmpExit(pVOID context, pCMPRECFNC pfnc)
```

CmpExit() is called each time a user closes a compressed data file. **CmpExit()** should de-initialize a context buffer, if one has been allocated in **CmpInit()**, and free the associated memory.

CmpExit() typically returns NO_ERROR unless the DLL specific code to de-initialize the context buffer returns an error code in which case return this error code.

The following three functions behave just as their compression counterparts except that they apply to decompression. **ExpActn()** has the same return values as **CmpExp()** except that attempting to decompress corrupted data results in a return value of *CMPREC_DATA* and *usys(2000 + err)* is called where *err* is the DLL specific internal code signifying the corruption.

```
pVOID ExpInit(pCMPRECFNC pfnc)
NINT  ExpActn(pVOID context, pCMPRECFNC pfnc, pVOID source, VRLEN slen, pVOID destination, pVRLEN pdlen)
NINT  ExpExit(pVOID context, pCMPRECFNC pfnc)
```

Copyright Notice

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

2. Index