About Locking in Xbase++

This article explains what happens when Xbase++ uses a file on a shared drive and how it handles file locking. You will learn about the internal locking protocols of file-based database applications and what the Extended Locking scheme introduced with Xbase++ 1.82 is all about.

Opening a file

When you open a database file with the USE command, the Xbase++ database engine uses the CreateFile() API to open the file. During this operation, Xbase++ sets the shared mode based on whether you request the file for SHARED or EXCLUSIVE usage. Xbase++ expects the operating system to take care of communicating file access rules to other processes. If you open the file exclusively, Xbase++ does not need to lock the file for update operations or to re-read the header prior to any append operation.

If a file is opened in shared mode, Xbase++ has to re-read records after any successful record lock. Also, the file header must be re-read prior to any append operation. In addition, the file header must be updated after any successful append. In exclusive mode, this is only done when the file is closed or when a COMMIT command is issued. Therefore, exclusive access is most of the time faster than shared access, even if no other user in a network tries to access the same file.

Work Areas

As part of opening a DBF file, Xbase++ reserves a section of memory for a work area containing various information about the data in use. The work area includes information such as the file's open mode, what records are locked, relations, related orders and so forth.

DBF Locking and Unlocking

If a DBF file is opened in shared mode, Xbase++ relies on the operating system to communicate lock information between processes. It uses the LOCK/UNLOCK APIs to handle locks. These APIs expect a byte offset and the number of bytes to lock when it is called. When you issue a record or a file lock, Xbase++ will call this API. Unfortunately, due to backward compatibility to Clipper/FoxPro, Xbase++ is only allowed to use the exclusive locking type. When it locks a portion of the file, it does not allow any other process to read or write that portion of the file.

If you would lock the middle of a DBF file, for example, then some other processes would not be able to read the middle of the DBF file until you release the lock. This is the semantics of an exclusive lock which is the only type of lock known by Clipper. Therefore, Xbase++ has to use the same type of locks to support concurrent file access. However, when Clipper's networking was designed, it was decided to allow read access to locked files and records. In order to allow read access, Clipper does not actually lock the physical file, it locks a location outside the file instead. Since it is not possible to read outside the file, the locked portion

will never be accessed for read or write purposes. This way Clipper and other Xbase systems have worked around the limitation of DOS to support only exclusive locking. They emulate so called Shared Locks.

Due to its compatibility, Xbase^{++} also has to use standard exclusive locks and emulate shared locks the same way as Clipper. This is a serious limitation for Xbase^{++} and causes a significant performance decrease, because Xbase^{++} is not allowed to take advantage of modern 32 bit operating-system capabilities such as shared locks and asynchronous write. This is the price we pay for compatibility.

Clipper compatible locking

The Xbase^{++} DBFDBE DatabaseEngine which is 100% Clipper 5.x compatible uses the following offset and bytes when you issue a record lock or a file lock:

```
Record Lock:
1,000,000,000 + record number    for one byte

File Lock:
 1,000,000,000    for the length of the file
```

When Xbase^{++} issues a record lock, it attempts to lock the location at one billion bytes for one byte. If another program has that byte locked, Xbase^{++}'s attempt to lock the record fails. Since this locking takes place beyond the end of the file, the DBF file can be read regardless of any file or record locks.

The default 1 gigabyte locking offset is the reason why Clipper 5.01 DBF files cannot be larger than one gigabyte in size. If a file grows larger, the locking scheme could lock the file physically and that portion of the file would become unaccessible for all other workstations.

This limitation has changed with Clipper 5.2 where the locking offset can be set to 2 gigabytes by simply linking the Ntxlock2.obj file to the Clipper application. In Xbase^{++} you can configure the lock offset for database engines using DbeInfo().

When Xbase^{++} unlocks a record, it merely calls the unlock API with the same offset and size.

FoxPro compatible locking

The use of different offsets and lock bytes is why concurrency between Clipper and FoxPro is broken. It also is the reason why Clipper can update a record locked by Foxpro and vice-versa. However with Xbase^{++}'s FOXDBE DatabaseEngine 100% interoperability and concurrency is guaranteed with FoxPro 2.x and Visual FoxPro.

Xbase++ FOXDBE compatibility to other DBF drivers

By default the FOXDBE provides 100% compatibility and concurrency with Visual FoxPro 3.x and higher. In addition, the FOXDBE provides a set of configuration options to handle compatibility to older versions of FoxPro, Comix and SIX drivers. To enable to FOXDBE to automatically detect the file type - such as Visual FoxPro, FoxPro, Comix and SIX - use the following setting:

```
DbeInfo( COMPONENT_STORAGE, FOXDBE_LOCKMODE, FOXDBE_LOCKMODE_AUTO )
```

With this setup the DBE is able to detect the file type of the DBF table you are trying to open. However, when you create a new DBF table, the DBE uses Visual FoxPro style locking schemes.

To configure the DBE to work properly with Fox 2.x, Comix and SIX DBF tables you have to use the following configuration:

```
DbeInfo( COMPONENT_STORAGE, FOXDBE_LOCKMODE, FOXDBE_LOCKMODE_2X )
DbeInfo( COMPONENT_STORAGE, FOXDBE_CREATE_2X, .T.)
```

This configuration causes the FOXDBE to use the Fox 2.x, Comix and SIX locking shemes for open DBF tables. It also creates tables in Fox 2.x, Comix and SIX format when creating a new table using DbCreate().

To force the DBE to always treat files in Visual FoxPro 3.x storage format and to perform VFP compatible locking the following setup is required:

```
DbeInfo( COMPONENT_STORAGE, FOXDBE_LOCKMODE, FOXDBE_LOCKMODE_VISUAL )
DbeInfo( COMPONENT_STORAGE, FOXDBE_CREATE_2X, .F. )
```

DBF File locking

Locking a record or the whole table using the DbRLock(), DbRUnlock(), DbLock() and Unlock() functions is called explicit locking. This is because the developer decides when to request and release the lock. Whenever an explicit lock fails, the DatabaseEngine specific locking implementations return immediatly to allow the developer to handle that kind of situation on the language level.

Whenever a record lock is successful, the DatabaseEngine automatically reloads the record from disk. This is important in shared environments to guarantee the correctness of data for the client that wants to modify data. The opposite is true for releasing the lock: whenever a lock is released and data is modified, the record is written back to disk prior to releasing the lock.

If a new record gets appended, the header of the DBF table must be updated which requires the DBE to obtain a special "header-lock". This lock is set implicitly whenever an append operation takes place. A header-lock may fail under rare circumstances, when multiple workstations perform append operations against the same table with a high frequency.

If a header-lock fails, the DBE raises a "lock failed error 32" at the DbAppend() operation. However, all storage DBEs such as FOXDBE and DBFDBE have a configuration option to increase the retry-count for header-lock attempts to avoid this special error condition.

The default retry-count is 100.000. To increase the retry-count you can use the following statements:

```
For the FOXDBE:
   DbeInfo( COMPONENT_STORAGE, FOXDBE_LOCK_RETRY, 1000000 )

For the DBFDBE:
   DbeInfo( COMPONENT_STORAGE, DBFDBE_LOCK_RETRY, 1000000 )
```

Index File locking

Index files are locked implicitly. You don't need to care about locking open index files. This is the good news. The bad news is that locking must take place in two situations: when an index is read and when it is updated. An index must be locked when the file is read or written to.

When a record is read from the DBF file it is transferred into an internal buffer. When a skip occurs, the next record must be navigated to. It is determined by the logical order of the index stored in the index file.

When a record is modified and is about to be unlocked, the following pattern occurs which is typical for all xbase systems:

1. The record is written back to disk
2. The old key-value is deleted from the index
3. The new key-value is written to the index
4. The record lock is released.

This update pattern has always the potential to bring the tree of the index-file temporarily into an inconsistent state, i.e. no link between nodes or wrong links between nodes exist until the update is complete. If another workstation would try to read/traverse the tree while an update takes place, the reading workstation could find dangling references between nodes of the tree.

For this reason, all xbase systems lock the entire index file exclusive while an update takes place. This means in turn, that an index file can only be read when no lock is set. On the other hand, an index update can only occur while no other read operation takes place. That's why read and write operations have to lock the entire index file.

The mutual exclusive nature of index file sharing is the critical bottleneck in terms of performance. All index operations, no matter if it is an update or read operation, are mutually exclusive and therefore not allowed to happen at the same point in time. We can always read all records of a table independently of active record locks. This is not the case with indexes. Here we can have only one workstation at the any point in time that accesses an index. All operations against an index file are therefore serialized by exclusively locking the entire index file.

When more and more workstations try to lock the index file, the timeslot to obtain a lock shrinks. When a lock fails, the DatabaseEngine has to retry to obtain a lock. The amount of retries can be configured for the DatabaseEngines using DbeInfo()

```
For the CDX DatabaseEngine:
   DbeInfo( COMPONENT_ORDER, CDXDBE_LOCKRETRY, 1000000 )

For the NTX DatabaseEngine:
   DbeInfo( COMPONENT_ORDER, NTXDBE_LOCKRETRY, 1000000 )
```

These function calls configure a retry-count of 1.000.000, the default value is 100.000.

As a conclusion, whenever you update a record, all open index files and contained orders/tags must also be updated. Fortunately, Xbase^{++} does a good job with determining which indexes/orders are affected by your changes, so only the minimum number of index files will be locked. However, for each index file that has to be changed, Xbase^{++} must lock the entire index file and not only parts of it - like it is the case with the table and its records. This is because an index is a tree of information where removing and inserting a key in many cases means changing links between nodes of the tree.

Deadlock situations in Index File locking

Whenever a record update occurs, the related index files must be updated as well. The update operations are performed automatically in the same sequence the index files were opened. That means if multiple processes operate on the same set of index files but opened them in different sequences, there is a good possbility of a deadlock. For example: station 1 opens index A, B and station 2 opens index B, A. When both stations update the indexes simultaneously, station 1 locks index A and station 2 index B first. After this, both stations wait forever to obtain a lock for the second index file. As a result, the applications hang and consume 100% CPU resources.

To avoid possible deadlock situations in concurrent data access, all applications operating on the same index files must open them in the same sequence.

Implicit locking and performance implications

Since index files are locked implicitly during read operations, it is simple to design a system that will spend more time on locking the index file than reading/writing the actual data. Imagine the following simple query:

```
USE "U_FILES.DBF" INDEX "U_FILES.CDX" SHARED
SET ORDER TO 1
GO TOP
nTotalSize := 0
DO WHILE !EOF()
   nTotalSize += FIELD->SIZE
   SKIP 1
ENDDO
```

This loop computes the sum of numeric data stored in a table. Our test dbf file has 75000 records and the following table lists the results of this simple query when running concurrently at multiple workstations (WKS):

Concurrent index file access

Table Scan w / wo Index	1-WKS	2-WKS	3-WKS	4-WKS	5-WKS
Absolute metrics in seconds					
Xbase^{++} w/o Index	0.03	0.06	0.09	0.12	0.15
Xbase^{++} w/Index	0.50	0.80	1.46	1.92	2.34
Clipper w/o Index	0.12	0.27	0.44	0.54	0.70
Clipper w/Index	0.50	1.60	2.97	4.67	9.72
Relative metrics					
Xbase^{++} w/o Index	100%	200%	300%	400%	500%
Xbase^{++} w/Index	100%	160%	292%	384%	468%
Clipper w/o Index	100%	225%	367%	450%	583%
Clipper w/Index	100%	320%	594%	934%	1944%

It is obvious from these figures, that data-access performance decreases with a growing number of workstations accessing the data simultaneously. This is true for Clipper and Xbase^{++}. The performance is radically decreased if navigation is performed via logical order (index) compared to natural order (table only). Note that the index file was open in both cases *w/Index* and *w/o Index*, only SET ORDER TO was different.

The Extended Locking scheme of Xbase++ 1.82

The complexity of data stored in index files is the reason why they must be locked for read and write operations. To guarantee consistency, Clipper, Comix, Six, Visual FoxPro, Visual dBase, ADS Server and Xbase++ applications have to place exclusive locks whenever they access the index file.

Since exclusive locks are expensive and can lead to drastic performance degradation in concurrency, we have equipped the CDXDBE of Xbase++ 1.82 with a new locking model called "Extended Locking". This new locking approach distinguishes between read and write access and places read and write locks depending on the action a process wants to perform. Using the extended locking model, read operations are no longer mutually exclusive for multiple processes, or workstations. Instead, all processes that read index files only can do this simultaneously. Because more than 90% of the average data-access in typical business solutions is reading data and not writing/modifying data, this extended locking approch increases performance dramatically.

The following table shows a measurement of execution time of our original sample query and compares the standard with the new extended locking scheme:

Concurrent index file access (Read Only)

Table scan RO	**1-WKS**	**2-WKS**	**3-WKS**	**4-WKS**	**5-WKS**
Absolute metrics in seconds					
Xbase++ Ext. Locking	0.45	0.47	0.49	0.51	0.53
Xbase++ Std. Locking	0.45	0.80	1.70	2.10	2.20
Clipper 5.2	0.33	0.66	0.99	2.05	2.52
Relative metrics					
Xbase++ Ext. Locking	100%	104%	109%	113%	118%
Xbase++ Std. Locking	100%	178%	378%	467%	489%
Clipper 5.2	100%	200%	300%	621%	764%

As you can see, the new extended locking model resolves some of the problems related to simultaneous index access. The amount of time needed by a single workstation is almost constant and the processing time is homogeneously spread across the workstations.

In the following table we see how the extended locking approach works in situations where 1% of operations is write and 99% is read access. There is again a significant performance improvement compared to Clipper and an even better improvement compared to Xbase++ using standard locking.

Concurrent index file access (1% Write)

Table scan R/W	1-WKS	2-WKS	3-WKS	4-WKS	5-WKS
Absolute metrics in seconds					
Xbase++ Ext. Locking	0.41	0.61	1.13	1.35	1.73
Xbase++ Std. Locking	0.40	1.10	2.28	2.95	3.56
Clipper 5.2	0.12	0.48	0.82	1.60	2.25
Relative metrics					
Xbase++ Ext. Locking	100%	149%	276%	329%	422%
Xbase++ Std. Locking	100%	275%	570%	738%	890%
Clipper 5.2	100%	400%	683%	1333%	1875%

The extended locking scheme can be enabled for the CDX Database Engine using DbeInfo():

```
DbeInfo( COMPONENT_ORDER, DBE_LOCKMODE, LOCKING_EXTENDED )
```

Conclusions

To maintain data consistency in concurrent data access, processes must exclude other processes from data access in certain situations. This is done by locking files explicitly and implicitly. Explicit locks are set on the Xbase++ language level via function calls while implicit locks are set automatically by Database Engines.

Implicit index file locks lead to a performance degradation in concurrency when the file is exclusively locked. This is the case with most xbase systems, including Xbase++ 1.80. The new extended locking scheme of Xbase++ 1.82 removes the bottleneck of mutually exclusive read operations on index files by allowing simultaneous read-access for multiple processes. Most business database solutions are going to benefit from this locking scheme, since most database-access operations are Read and not Write.