White Paper

Understanding Microsoft Jet Locking

*Kevin Collins*  
Microsoft Jet Program Management  
9/1/96

Updated for Jet 3.5/DAO 3.5

© 1996-1997 Microsoft Corporation. All rights reserved.



Introduction

Layout of the .Ldb File

Database Header Page

Types of Microsoft Jet Locks

Shared vs. Exclusive Locks

User Locks

Write Locks

Read Locks (Microsoft Jet 2*.x*)

Read Locks (Microsoft Jet 3.0 and 3.5)

Commit Locks

Table-Read Locks

Table-Write Locks

Table Deny-Write Locks

Deciphering Lock Ranges

User-Lock Algorithm

Deciphering Write Locks

Data Page Write Locks

Internal Transactions on SQL DML Statements

Internal Transactions on DAO Code with Microsoft Jet 3.0 and 3.5

Nested Transactions

Index Page Write Locks

Inserting Records

Inserting Records with a Counter

Deleting Records

Updating Records in a Multiple-Table Join

Deciphering Read and Commit Locks (Microsoft Jet 2.x)

Deciphering Read and Commit Locks (Microsoft Jet 3.0 and 3.5)

Deciphering All Types of Table Locks

Multiuser Enhancements Specific to Microsoft Jet 3.5

Removal of Implicit Transactions for SQL DML Statements

New Registry Settings

New DAO Methods and Options

Using the Utilities

LDBView

DBLock

MSLDBUSR.DLL

## Introduction

The purpose of this paper is to introduce you to Microsoft Jet 2*.x* and Microsoft Jet 3.0 and 3.5 locking techniques so that you can apply the information to coding and debugging when developing multiuser applications. The paper has a special section near the end that discusses features particular to Microsoft Jet 3.5. Other sections have also been updated with features particular to Microsoft Jet 3.5. This paper will address the following areas:

* Layout of the .ldb file
* Database header page
* Types of Microsoft Jet locks
* Deciphering Microsoft Jet locks

## Layout of the .Ldb File

The .ldb file plays an important role in the Microsoft Jet multiuser scheme. This file stores the computer and security names and has extended byte range locks placed on it by Microsoft Jet.

Microsoft Jet retrieves the computer name by making a request to the operating system. To modify the computer name in Windows® for Workgroups, Windows 95, or Windows NT®, go to Control Panel and double-click the Network icon. From there you can change the computer name. Windows for Workgroups stores the computer name in the System.ini file, whereas Windows 95 and Windows NT store the computer name in the registry. The security name is determined by passing a value to the **Workspace** object by using Data Access Objects (DAO). The default security name is Admin.

Extended byte range locks are locks placed outside the physical boundaries of a file — no data is ever physically locked. An example of this is placing a lock at 10 million hex for a file that has a physical size of only 64 bytes. In other words, a lock is *virtually* placed at a location that does not exist on the hard disk. This type of locking is used because extended byte range locks are not limited by the size of the physical file, allowing for locking algorithms that would not otherwise be possible. Also, placing locks inside a data file would prevent other users from reading that data. In the early dBASE days, a user could place a lock on a record located in the data file that prevented everyone from reading that data — when printing a report, for example.

One .ldb file is always created for every Microsoft Jet database file that is connected in a shared mode. The .ldb file always retains the same name as the database that was opened (for example, Nwind.mdb always has a Nwind.ldb file), and the .ldb file will always be in the same directory as the database. If an .ldb file does not exist and the database is connected in a shared mode, an .ldb file is created. The physical structure of the .ldb file is best thought of as a one-dimensional array. Each element in the array, up to 255, consists of 64 bytes. The first 32 bytes contain the computer name (such as Machine1), and the second 32 bytes contain the security name (such as Admin). This data is then used to provide information regarding which other users are holding locks. You can view this information by simply looking at the contents of the .ldb file.

The physical size of the .ldb file never exceeds 16,320 bytes, because the maximum number of concurrent users in a Microsoft Jet database is 255 (255 \* 64 = 16320).

The high-order ASCII characters seen in Microsoft Jet 2*.x* .ldb files are meaningless to both the viewer and to Microsoft Jet. Microsoft Jet 3.0 and 3.5 eliminate the high-order ASCII characters from the .ldb file so that just the computer and security names are shown.

Microsoft Jet 3.0 and 3.5 automatically delete the .ldb file when the last user closes the database. This is done to aid in issues with replicated databases and to allow for performance improvements when determining which other users have locks. The exception to this is when a user does not have NOS/OS delete rights to the .ldb file or if the database is in a suspect state. There is no performance or concurrency benefit gained from deleting your .ldb file in a Microsoft Jet 2.*x* environment.

A 32-bit utility (LDBView) is included to enable you to view the status of the commit byte(s) (commit bytes are described in the next section) and what users are currently logged on to the database. This utility helps you determine which user or users left the database in a suspect state.

Another utility (MSLDBUSR.DLL) enables you to programmatically retrieve the following information if you’re running 32-bit VBA applications:

* List of all users in an .ldb file
* List of all connected users in an .ldb file
* List of users that left the database in a suspected state

You can find out how to use this DLL at the end of this paper.

## Database Header Page

The database header page (DBH) is the first page in a database. It is partially used to store the commit byte(s) for each of the 256 possible users connected to a database. A commit byte (or pair of bytes in version 3.0 and 3.5) is a value that is used by Microsoft Jet to determine the state of the database.

Microsoft Jet 2*.x* utilizes 256 bytes that store the commit bytes for each possible user starting at 700 hex and continuing to the end of the page (800 hex). Microsoft Jet 3.0 and 3.5 utilize 512 bytes that use two bytes per user starting at 600 hex. The first byte (Microsoft Jet 2*.x*) or the first two bytes (Microsoft Jet 3.0 and 3.5) are used only when a database is connected in an exclusive mode and the remaining 255 bytes (Microsoft Jet 2*.x*) or 510 bytes (Microsoft Jet 3.0 and 3.5) are used when the database is connected in a shared mode.

Commit bytes in Microsoft Jet 2.0 have only two valid values, FF and 00. A value of 00 represents a neutral state and a value of FF means that Microsoft Jet is in the process of physically writing data to disk. If there is a value of FF and no corresponding user lock, then a user has had an abnormal shutdown of the database file. New users attempting to connect to the database will receive the message “Database is corrupted or is not a database file” and they will be forced to run repair on the database to open it. An anomaly in Microsoft Jet 2.0 allowed instances where the database could be closed with a value of FF, thus preventing other users from connecting to the database until repair could be run. This was corrected in Microsoft Jet 2.5.

Commit bytes in Microsoft Jet 2.5 have five valid values ranging from 00 to 04. This range of values provides more information about what users were doing when the database is left in a suspect state. Similar to Microsoft Jet 2.0, the database is in a suspect state if there is a nonzero value without a corresponding user lock. A value of 00 represents a neutral state. A value of 01 indicates that a user accessed a corrupted page in the database. A value of 02 indicates the database is being created. A value of 03 indicates that the database is being repaired, and a value of 04 indicates that the user is in the process of physically writing data to disk.

Commit bytes in Microsoft Jet 3.0 and 3.5 can have many different values, so they were increased to 2 bytes. A value of 00 00 indicates that the user is in the process of physically writing to disk, and a value of 01 00 indicates that a user has accessed a corrupted page. Therefore, if a value of 00 00 is present without a corresponding user lock, or a value of 01 00 is present, users will not be allowed to connect to the database without first executing the repair utility. If you look at these commit-byte values with the LDBView utility, you will notice that there may be many other values present in the 600~800 hex range. These values are used internally by Microsoft Jet for performance reasons (mainly to determine if other users have written data to the database file). If Microsoft Jet determines that other users have not written information to disk, it will delay refreshing its internal cache, resulting in fewer reads and increasing performance. Therefore, it is possible that a database opened in shared mode may approach the performance levels of a database opened in exclusive mode.

Again, which commit bytes are actually used for a user is determined by the corresponding user lock in the .ldb file. A detailed discussion of how these locations are obtained appears later in this paper. Also, a user can monitor these values for both Microsoft Jet 2*.x* and 3.0 and 3.5 databases by using the 32-bit LDBView utility included on the CD-ROM.

## Types of Microsoft Jet Locks

### Shared vs. Exclusive Locks

These two types of locks will be referred to frequently later in the paper and it is important to understand the concepts behind them. Regardless of the type of lock, each extended byte range lock placed on the .ldb file always represents one page in the .mdb file (except for user locks). No locks are ever placed on the .mdb file.

A shared lock occupies only 1 byte and never conflicts with another shared lock. These types of locks are typically used to allow many people to read information at the same time. An example of when shared locks are used is when many users have the same table open at the same time without conflicting with each other. Another example is when users are reading an index that participates in referential integrity: Many users would be able to read the index at the same time.

An exclusive lock spans between 256 and 512 bytes and always conflicts with other shared locks and other exclusive locks. The exclusive lock always locks the first 256 bytes of the range to prevent any shared locks from being set and to determine if there are any existing shared locks set. In addition, it locks enough additional bytes beyond 256 to determine which user is holding the lock. An example of an exclusive lock is when a user opens a table in deny-read mode. This exclusive lock would be prevented if another user had a shared lock on the table; if the exclusive lock were acquired, it would prevent other users from obtaining a shared lock. Another example is when a user is writing information back to an index page that participates in referential integrity. This would produce a write lock, which is always exclusive, and would prevent anyone from reading the index page. The index page requires a read lock and is always shared.

There are seven types of Microsoft Jet locks:

* User locks
* Write locks
* Read locks
* Commit locks (Microsoft Jet 2*.x* only)
* Table-read locks
* Table-write locks
* Table deny-write locks

These extended byte range locks are broken out into six virtual regions off of the .ldb file for Microsoft Jet 2*.x* and five virtual regions for Microsoft Jet 3.0 and 3.5. These areas are where Microsoft Jet places extended byte range locks that range from 10000001 to 6FF800FF hex. These locks are only present when users have the database connected in a shared or read-only mode. Note that the names of these locks were assigned by the Microsoft Jet development team and do not necessarily have the same meaning that other database vendors use. See Table 1 after the descriptions below for a summary of these locking ranges.

### User Locks

User locks determine ownership of a commit byte(s) in the DBH, write the computer and security names in the correct location in the .ldb file, and retrieve the computer and security names of another user that has conflicting locks. A user lock is obtained and is persistent for as long as a user is connected to the database (persistence in this context indicates the duration of a lock or how long a lock is held). Only one user lock exists for each connected user; however, multiple instances of Microsoft Jet on the same computer create an individual user lock for each instance. User locks are always in the range of 10000001 hex through 100000FF hex. They occupy only 1 byte (for example, 10000002 ~ 10000002).

### Write Locks

Write locks prevent other users from changing data while a user is modifying data. A write lock is typically placed on data, index, or long value pages. (Long value pages are a type of data page that contains ANSI SQL data types of CHARACTER VARYING, BIT, or BIT VARYING. These data types are known as Memo or OLE Object in Microsoft Access or Visual Basic®.) Write locks are in the range of 20000000 hex through 2FF800FF hex and always span between 256 and 512 bytes (for example, 2000E600 ~ 2000E700 hex); therefore, they are always exclusive locks. The persistence of write locks is directly related to the duration of a transaction. All SQL DML statements have implicit transactions placed around them. Thus, an UPDATE statement will have persistent write locks until the entire update is committed. The persistence of write locks is also determined by the type of locking that is chosen for recordset navigation and form editing (for example, pessimistic or optimistic) and if explicit transactions are used. For more information about locking semantics, see Chapter 10, “Creating Multiuser Applications,” in *Building Applications with Microsoft Access 97.*

### Read Locks (Microsoft Jet 2*.x*)

Read locks, which are a type of shared lock, are primarily used for the immediate recycling of index pages and for ensuring that index pages in Microsoft Jet’s cache are up to date. This type of lock is placed on long value pages and index pages. It is used to prevent an index page from being recycled while that page is being referenced in Microsoft Jet’s cache, and to provide an integral view of the index. Read locks are placed in the range of 30000000 ~ 3FF800FF hex. They occupy only 1 byte (for example, 30001A01 ~ 30001A01).

Read locks in version 2.*x* are probably the most troublesome types of locks to deal with as they can cause locking conflicts that are not obvious to the developer or user. The persistence of read locks is determined by:

* The LockedPageTimeout setting, specified in tenths of a second, which determines the amount of time the Microsoft Jet database engine retains locks before releasing them. By increasing the LockedPageTimeout setting, read locks are retained for a longer period of time, which reduces the need for Microsoft Jet to reread data pages into cache. The LockedPageTimeout setting can be changed by modifying the ISAM section of the corresponding .ini file (MSACC20.INI for Microsoft Access users) or by modifying the value in the registry for Windows 95 or Windows NT in the Microsoft Jet 2*.x*\ISAM key. The side effects could be reduced concurrency due to commit lock conflicts, which will be talked about next.
* Calling **DBEngine.Idle dbFreeLocks** or **FreeLocks**. However, these commands do not always free all read locks when inside a transaction (see lock log examples later in this paper). Note that the **dbFreeLocks** constant is used in DAO 3.0 and 3.5 and are used when Access 7.0/8.0 or VB 4.0/5.0 opens a Microsoft Jet 2*.x* database. If Access 2.0 or Visual Basic 3.*x* is used, then the constant would be **DB\_FreeLocks**.

Read-locks are typically placed when an index or long value page is read and placed in Microsoft Jet’s cache. An example would be performing an SQL DML UPDATE command on a table that has a primary key.

### Read Locks (Microsoft Jet 3.0 and 3.5)

The main purpose of read locks in Microsoft Jet 3.0 and 3.5 is to allow multiple users to read long value data, but prevent users from writing to it while others are reading.

Microsoft Jet 3.0 and 3.5 have reduced the amount of read locks that are placed on index pages, which directly results in greater concurrency and performance. Read locks on index pages are now only placed when referential integrity is being enforced. Unlike Microsoft Jet 2*.x*, the persistence of read locks is determined by the currency of a record. Thus a read lock on a long value page will remain until the user leaves that record. An exception to this is when a long value page contains data from more than one record, at which point Microsoft Jet will release the read lock on that page. An example of this is when several records of Memo data are placed on one long value page.

Because read locks are persistent until the user loses currency on the record, **DBEngine.Idle** **dbFreeLocks** and **FreeLocks** no longer provide any functionality (**FreeLocks** is no longer supported in Visual Basic 4.0/5.0 or Microsoft Access 7.0/8.0, unless the DAO 2.5/3.0 or 2.5/3.5 compatibility layer is enabled). Microsoft Jet 3.0 and 3.5 read locks are also placed in the same range as write locks; the only difference is that read locks are shared locks and they occupy only 1 byte (for example 20001A01 ~ 20001A01).

### Commit Locks

Commit locks are only present in Microsoft Jet 2.*x* and are very similar to read locks, except that they are always exclusive locks. These locks are also placed in the 30001A01 ~ 3FF800FF hex range. They are placed when either index pages or long value pages are being written back to the database, and they only conflict with read locks. Read locks are typically placed on index pages whenever an index page is placed in the Microsoft Jet cache. An index page references many data pages, therefore it becomes apparent why many users can experience locking conflicts when they know other users are not editing data on the same data page.

One of the best mechanisms to prevent these conflicts is to judiciously use **DBEngine.Idle DB\_FreeLocks** or **FreeLocks** in the application code. The use of these two commands is the most effective way to free up read locks so that commit locks will not conflict with them.

Commit locks do not exist in Microsoft Jet 3.0 and 3.5 because the range for read locks has been moved into the same range as write locks, thus eliminating the need for this type of lock.

### Table-Read Locks

Table-read locks are used to control placing a table in a deny-read mode, which uses an exclusive lock and prevents other users from reading data from the table. Unlike the previous types of locks, table-read locks and the rest of the table-type locks are placed only on a special type of page called a table header page (TBH). There is one TBH page for each table, and every TBH contains statistics about the table (such as record count, next counter value, field data types and index types).

Table-read locks are placed in the 40000000 ~ 4FF800FF hex range and can be placed as shared locks or exclusive locks.

When a table is opened, a shared table-read lock is placed. Deny-read mode is set when the default locking on a database is set to All Records (through the Microsoft Access user interface) or **dbDenyRead** is issued through the DAO **OpenRecordset** method. If the exclusive lock can be obtained, then there are no other users that have the table open. The exclusive lock prevents other users from acquiring shared locks when they try to open a table. These locks are persistent until the user closes the table.

### Table-Write Locks

Table-write locks are used in conjunction with table deny-write locks and are placed in the 50000000 ~ 5FF800FF hex range. These shared locks are persistent whenever a table is opened in a state that allows writing.

### Table Deny-Write Locks

These locks are used in conjunction with table-write locks and are explicitly set when opening a table in deny-write mode. These locks are placed in the 60000000 ~ 6FF800FF hex range and have a persistent shared lock while the table is opened in deny-write mode. An exclusive lock is placed, but not held, to determine which other users have the table open in deny-write mode.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Lock name** | **What sets the lock** | **What is locked** | **What the lock prevents** | **Persistence of the lock** | **Shared or exclusive available** | **Byte range examples** |
| User | User opens an .mdb | N/A | Nothing | Until user exits .mdb | N/A | 10000001-  10000001 |
| Write | Insert, update or delete | All available page types | Updates or deletes to data and sometimes inserts to a table | Controlled by default locking (optimistic or pessimistic) and the duration of a transaction | Exclusive | 2000A601-  2000A701 |
| Read (version 2.*x*) | Reads on a long value or an index page | Long value, index, or TBH | Updates or deletes to long value pages or index | Controlled by LockedPageTimeout in the .ini file or by issuing a **FreeLock** call | Shared | 30001E01-  30001E01 |
| Read (version 3.0 and 3.5) | Reads on certain long values or index pages when referential integrity is being enforced | Long value, index, or directory pages | Updates or deletes to long value or index pages | Held until a read or a transaction is complete or the user moves to a new record | Shared | 20063801-  20063801 |
| Commit (dropped in version 3.0 and 3.5) | Writes to long value or index pages | Long value or index pages | Prevents reads when data is being written to disk | Until data is finished being written to disk | Exclusive | 30001A01-  30001B01 |
| Table read | Shared lock is obtain when a table is opened; exclusive lock is obtained when default locking is set to All Records through Microsoft Access or Set **dbDenyRead** has been issued through DAO **OpenRecordset** | TBH | Prevents exclusive read lock from being set | As long as the table is open | Both | 4000C801-  4000C801 |
| Table write | Shared lock is obtained when a table is opened; exclusive lock is obtained when default locking is set to All Records through Microsoft Access or **dbDenyRead** has been issued through DAO **OpenRecordset** | TBH | Prevents exclusive write lock from being set, does not prevent Table deny-write lock | As long as the table is open | Both | 5000C801-  5000C801 |
| Table-deny-write | Can be set only through DAO | TBH | Prevents all writes to the table | As long as the table is open | Both | 6000C801-  6000C801 |

**Table 1**

## Deciphering Lock Ranges

To utilize the information previously discussed in a way that can help you resolve locking issues with your application, you need to be able to view, understand, and decipher the locks that Microsoft Jet places.

To view the locks being placed, you need to have either a Novell NetWare environment, Microsoft SMS Network Monitor, or some other “network sniffing” tool. By having one of these utilities, particularly the real-time NetWare Monitor program, an administrator can view the locks being placed by Microsoft Jet on the .ldb file. Developers can also use this information to see how their code places locks on the .ldb file and what effects it might have in a multiuser environment.

## User-Lock Algorithm

All Microsoft Jet multiuser locking schemes revolve around the placement of a user lock. As stated previously, user locks are placed in the 10 million hex range and occupy only 1 byte. When Microsoft Jet connects to a database in a shared or read-only mode, the following activities occur before a user lock is actually placed.

The first activity is to determine if the Microsoft Jet database engine is in a suspect state. This is done by examining the DBH page and seeing what bytes have a nonzero value. If the first byte (Microsoft Jet 2*.x*) or the first 2 bytes (Microsoft Jet 3.0 and 3.5) contain a commit-in-progress value, or the remaining 255 bytes (Microsoft Jet 2.*x*) or 510 bytes (Microsoft Jet 3.0 and 3.5) have a commit-in-progress value and do not have a corresponding user lock, then Microsoft Jet will force the user to repair the database. When opening a database in shared mode there is never a need to check for a user lock on the first byte (Microsoft Jet 2.*x*) or the first two bytes (Microsoft Jet 3.0 and 3.5) because it is only used when a database is connected exclusively. A corresponding user lock would be a lock that shares the same offset in the ten million hex range as the offset from the first byte in the DBH page. Thus, a non-zero commit byte at 701 hex (Microsoft Jet 2*.x*) or 602~603 hex (Microsoft Jet 3.0 and 3.5) would need to have a user lock at 10000001 in order for the database to be opened without a corrupt database message.

The Microsoft Jet database engine then opens the .ldb file (or creates one if one does not exist) and tries to place a lock at 10000001 hex. If Microsoft Jet is successful in obtaining this lock, it will write the computer and security name to the first 64 bytes of the file. If Microsoft Jet cannot acquire this lock, it will continue moving one byte further until a lock can be successfully obtained. After the user lock is acquired, the Microsoft Jet database engine will then write the computer and security name at the corresponding location in the .ldb file. For example, a user lock at 10000040 hex would write an entry starting at 4096 bytes in the physical part of the .ldb file.

Some users have experienced delays when trying to connect to a database that already has many users connected. This is mainly due to non-optimized network drivers. An example of this is using a Windows NT 3.5 client accessing a NetWare server. If a user were trying to connect to the database and 30 other users were currently connected, it could take upwards of 30 seconds to perform the user lock algorithm. Windows NT 3.51 has a modified NetWare requester driver that brings the above scenario down to approximately two seconds.

## Deciphering Write Locks

As stated previously, write locks will always occur in the 20 million hex range and will always have a starting and ending lock range between 256 and 512 bytes. The code below places write locks on the .ldb file that relates to pages in the Customers table. All future references to locks on pages are really referring to locks placed on the .ldb file that relates to pages in the database file.

**NOTE:** The following code examples assume that the user is using Visual Basic 4.0/5.0 or Microsoft Access 7.0/8.0 with either an original Microsoft Jet 2*.x* version of the NWIND database that shipped with Microsoft Access 2.0, or a NWIND database that was compacted to Microsoft Jet 3.0 and 3.5 from the original Microsoft Access 2.0 NWIND database (using the DAO **CompactDatabase** method in Visual Basic 4.0/5.0 32-bit, or Microsoft Access 7.0/8.0). If Microsoft Access 7.0/8.0 is used to convert the Microsoft Access 2.0 NWIND database via the Convert option from the Tools/Database Utilities menu, then the logs will differ from the Microsoft Jet 3.0 and 3.5 logs below. The logs also assume that the user is the first user logged on to the database, thus implying a user lock of 10000001 hex.

### Data Page Write Locks

Here is sample code and a lock log of write locks placed on data pages:

Sub WriteLocksDataPages ()

' The example is intended to be run from Access 7.0/8.0 or VB 4.0/5.0.

Dim db As Database

Dim rs As Recordset

Dim ws As Workspace

Dim ContactName As String

Set db = OpenDatabase("NWind.mdb", False, False)

Set rs = db.OpenRecordset("SELECT \* FROM Customers")

rs.LockEdits = False

Set ws = Workspaces(0)

ws.BeginTrans

While Not rs.EOF

rs.Edit

ContactName = rs![Contact Name]

rs![Contact Name] = ContactName

rs.Update

rs.MoveNext

Wend

ws.CommitTrans

End Sub

When you set a breakpoint on the recordset **MoveNext** method, the Microsoft Jet database engine will leave a write lock on the first data page of the Customers table. Assuming that only one user is connected to the database, the write lock will have a starting and ending lock address of 2000AC00 ~ 2000AD00 (Microsoft Jet 2*.x*). To determine what page is being locked, the following steps need to occur:

1. Remove the first digit. (2000AC00 à AC00)
2. Convert to decimal (AC00 à 44032)
3. Determine the page number by dividing by 512 (44032 / 512 = 86). Thus, page 86 is being locked via the .ldb file by rs.Edit. Page 86 would represent the page in the database starting at 176,128 bytes and extending to 178,176 bytes.

To determine the user number that is locking the page, simply take the last two digits of the ending address, convert it to decimal and add one digit if the database is Microsoft Jet 2*.x* (2000AD00 à 00 + 1 = 1). If the database is a Microsoft Jet 3.0 or 3.5 database, simply converting the last two digits to decimal will give the user number (2000A701 à 01 = 1). Thus, in the Microsoft Jet 2*.x* example above, user 1 would be holding the lock (last two digits are zero, plus 1).

To find the computer name of the user, simply open the corresponding .ldb file and move to the offset of the user number \* 64. Using the previous example, the computer name would be in the first 32 bytes of the .ldb file (1 \* 64). Alternatively, you can use the 32-bit LDBView utility included on the CD-ROM to quickly find the computer name of the user who is placing the lock.

In addition to knowing which user has a page locked, it is usually more useful to know which table is associated with the write lock. You can find this information by using the 32-bit Visual Basic 4.0/5.0 utility that is included on the CD-ROM (DBLock).

Following is a log of the majority of locks placed from the previous code example with descriptions for each lock. By using DBLock, you can enter the beginning lock range shown and get a description of the type of lock and which table it is locking. DBLock cannot get table names for long value pages and other internal types of pages used by Microsoft Jet.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **DAO command** | **Microsoft Jet 2*.x* lock hex** | **Description** | **Microsoft Jet 3.0 and 3.5 lock hex** | **Description** |
| OpenDatabase | 10000001~10000001 | User lock | 10000001~10000001 | User lock |
|  | 30001A01~30001A01 | Read lock on index page for MSysObjects |  |  |
|  | 30001E01~30001E01 | Read lock on index page for MSysObjects |  |  |
|  | 30002001~30002001 | Read lock on index page for MSysIndexes |  |  |
|  | 30002201~30002201 | Read lock on index page for MSysACEs |  |  |
|  | 30003001~30003001 | Read lock on index page for MSysObjects |  |  |
|  | 3000AA01~3000AA01 | Read lock on index page for MSysColumns |  |  |
|  | 30040C01~30000C01 | Read lock on index page for MSysColumns |  |  |
|  | 30065201~30065201 | Read lock on index page for MSysACEs |  |  |
| OpenRecordset | 4000A601~4000A601 | Shared table-read lock on Customers | 4000A201~400A201 | Shared table-read lock on Customers |
|  | 5000A601~5000A601 | Shared table-write lock on Customers |  |  |
| rs.Edit |  | No lock placed due to optimistic locking |  |  |
| rs.Update | 2000AC00~2000AD00 | Write lock on table customers | 2000A601~2000A701 | Write lock on table customers |
| Repeat loop until EOF. | 2000AE00~2000AF00 | 15 records have looped through before the next data page is write locked. Even though pessimistic locking is enabled, the transaction holds the write locks until ws.CommitTrans | 2000A801~2200A901 | 16 records have looped through before the next data page is write locked. |
|  | 2000B000~2000B100 |  | 2000AA01~2000AB01 |  |
|  | 2000B200~2000B300 |  | 2000AC01~2000AD01 |  |
|  | 2000B400~2000B500 |  | 2000AE01~2000AF01 |  |
|  | 2000B600~2000B700 |  | 2000B001~2000B101 |  |
|  | 2000B800~2000B900 |  | 2000B201~2000B301 |  |

There is some interesting information in this trace log. Notice that the Microsoft Jet 3.0 or 3.5 log has no read locks placed and that the last two digits of the write locks represent the true user number, whereas the last two digits for the Microsoft Jet 2*.x* write locks require the user to add 1 to determine the true user number. Also, when optimistic locking is used, the explicit transaction will hold write locks for the duration of the transaction, thus emulating pessimistic locking. Many users mistakenly believe that optimistic locking will always be in effect regardless of the transaction mechanism. Another example of this would be to issue the following code to mimic the DAO code example above.

### Internal Transactions on SQL DML Statements

Sub WriteLocksDML ()

' This code example is intended to be run from Access 7.0/8.0 or VB

' 4.0/5.0.

Dim db As Database

Set db = OpenDatabase("NWind.mdb", False, False)

DBEngine.Execute \_

"UPDATE Customers SET ContactName = ContactName", dbFailOnError

End Sub

Because all SQL DML commands are implicitly wrapped in a transaction, the write locks in the example above will be held until the UPDATE statement successfully completes. **NOTE**: Microsoft Jet 3.5 no longer places implicit transactions around SQL DML statements. Internal transactions are now used with SQL DML statements, thus allowing for greater performance and increased concurrency. How long the internal transaction will persist and hold locks is determined by the following Microsoft Jet 3.5 registry settings: FlushTransactionTimeout and MaxBufferSize or SharedAsyncDelay, MaxBufferSize, and ImplicitCommitSync or UserCommitSync (more detail on these settings can be found in the “Microsoft Jet 3.0 Performance Overview” and “Microsoft Jet 3.5 Performance Overview and Optimization Techniques” white papers).

### Internal Transactions on DAO Code with Microsoft Jet 3.0 and 3.5

By removing the explicit calls to ws.BeginTrans and ws.CommitTrans from the code above (WriteLocksDatapage), Microsoft Jet 3.0 and 3.5 will utilize internal transactions. Based on settings (SharedAsyncDelay and MaxBufferSize) in the system registry, Microsoft Jet places a series of commands in an internal transaction (more detail on these settings can be found in the “Microsoft Jet 3.0 Performance Overview” and in the “Microsoft Jet 3.5 Performance Overview and Optimization Techniques” white papers). While this does speed up performance, it may also create concurrency issues where they did not exist in Microsoft Jet 2*.x*.

### Nested Transactions

The following code example and lock log demonstrate that locks are held in nested transactions until the outermost transaction is committed:

Sub NestedTransactions()

' This code example is intended to be run from Access 7.0/8.0 or VB

' 4.0/5.0.

Dim db As Database

Dim ws As Workspace

Dim ws1 As Workspace

Set ws = Workspaces(0)

Set ws1 = Workspaces(0)

ws.BeginTrans

db.Execute \_

"UPDATE Suppliers SET [Contact Name] = [Contact Name]", dbFailOnError

ws1.BeginTrans

db.Execute \_

"UPDATE Employees SET [Last Name] = [Last Name]", dbFailOnError

ws1.CommitTrans

ws.CommitTrans

End Sub

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **DAO command** | **Microsoft Jet 2*.x* lock hex** | **Description** | **Microsoft Jet 3.0 and 3.5 lock hex** | **Description** |
| ws.BeginTrans |  | Start outer level transaction. |  | Start outer level transaction. |
| UPDATE Suppliers | 40042401~40042401 | Shared table-read lock on Suppliers table. | 400C7E01-400C7E01 | Shared table-read lock on Suppliers table. |
|  | 50042401~50042401 | Shared table-write lock on Suppliers table. | 200C5001-200C5101 | Write lock on data page for MSysObjects. This is for compilation of the temporary query and is not placed every time. |
|  | 20042600~20042700 | Write lock on data page for Suppliers table. | 200C8401-200C8501 | Write lock on data page for Suppliers table. |
|  | 20042800~20042900 | Write lock on data page for Suppliers table. | 200C8601-200C8701 | Write lock on data page for Suppliers table. |
|  | 20042A00~20042B00 | Write lock on data page for Suppliers table. |  |  |
| ws1.BeginTrans |  | Start inner level transaction. |  | Start inner level transaction |
| UPDATE Employees | 4000CC01~4000CC01 | Shared table-read lock on Employees table. | 4000C801-4000C801 | Shared table-read lock on Employees table. |
|  | 5000CC01~5000CC01 | Shared table-write lock on Employee table. |  |  |
|  | 2000E600~2000E700 | Write lock on data page for Employee table. | 2000E401-2000E501 | Write lock on data page for MSysObjects. This is for compiling the temp query. |
|  | 20020400~20020500 | Write lock on data page for Employee table. | 20020801-20020901 | Write lock on data page for Employee table. |
| ws1.CommitTrans |  | Write locks stay on. |  | Write locks stay on. |
| ws1.Rollback |  | If a rollback was issued here, the write locks on the Employees table would be removed. |  | If a rollback was issued here, the write locks on the Employees table would be removed. |
| ws.CommitTrans |  | Write locks are removed. |  | Write locks are removed. |

The previous example illustrates a couple of key points:

* If a **CommitTrans** or **Rollback** is not issued or is not balanced with corresponding **BeginTrans** commands, write locks will be held until the database is closed. This can cause serious concurrency issues. If a user does not explicitly check the error status of a **Rollback** or **CommitTrans** statement, the command could fail, leaving locks on the .ldb file.
* Simply executing a nested **CommitTrans** statement does not remove write locks nor is the modified data actually flushed to disk until the outermost transaction is committed. DAO 3.5 exposes a new option to the **CommitTrans** method that guarantees that data will be written to disk for Windows 95 and Windows NT operating systems. The new option to **CommitTrans** is called **dbForceOSFlush**, and this option will force the Windows 95 or Windows NT cache to issue synchronous Microsoft Jet writes to disk. NetWare does not honor these calls and will issue Microsoft Jet writes asynchronously to disk.

### Index Page Write Locks

Below is sample code and a lock log of the majority of write locks placed on both data and index pages.

Sub WriteLocksIndexPages ()

' This code example is intended to be run from Access 7.0/8.0 or VB

' 4.0/5.0.

Dim db As Database

Dim rs As Recordset

Dim ws As Workspace

Dim CompanyName As String

Set db = OpenDatabase("NWind.mdb", False, False)

Set rs = db.OpenRecordset("SELECT \* FROM Customers", dbOpenDynaset)

' The constant dbFreeLocks is DAO v3 specific and is used when

' Jet 3.0 or 3.5 (Access 7.0/8.0 or VB 4.0/5.0) is opening a Jet

' 2*.*x database. If Access 2.0 or VB 3.x is used, the constant would

' be db\_FreeLocks.

DBEngine.Idle dbFreeLocks 'Works only in Jet 2*.*x*.*

Set ws = Workspaces(0)

ws.BeginTrans

While Not rs.EOF

rs.Edit

CompanyName = rs![Company Name]

rs![Company Name] = CompanyName

DBEngine.Idle dbFreeLocks ' Works only in Jet 2*.*x.

rs.Update

rs.MoveNext

Wend

ws.CommitTrans

' The following command can only be used with DAO 3.5 and Jet 3.5.

' ws.CommitTrans dbForceOSFlush

rs.Close

End Sub

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **DAO Command** | **Microsoft Jet 2*.x* lock hex** | **Description** | **Microsoft Jet 3.0 and 3.5 lock hex** | **Description** |
| OpenDatabase | 10000001 | User lock | 10000001 | User lock |
|  | 30001A01~30001A01 | Read lock on index page for MSysObjects |  |  |
|  | 30001E01~30001E01 | Read lock on index page for MSysObjects |  |  |
|  | 30002001~30002001 | Read lock on index page for MSysIndexes |  |  |
|  | 30002201~30002201 | Read lock on index page for MSysACEs |  |  |
|  | 30003001~30003001 | Read lock on index page for MSysObjects |  |  |
|  | 3000AA01~3000AA01 | Read lock on index page for MSysColumns |  |  |
|  | 30040C01~30000C01 | Read lock on index page for MSysColumns |  |  |
|  | 30065201~30065201 | Read lock on index page for MSysACEs |  |  |
| OpenRecordset | 4000A601~4000A601 | Shared table-read lock on Customers table | 4000A201~4000A201 | Shared table-read lock on Customers table |
|  | 5000A601~5000A601 | Shared table-write lock on Customers table |  |  |
| dbFreeLocks |  | All 30 million hex locks removed |  | Does nothing |
| rs.Edit | 2000AC00~2000AD00 |  | 2000A601~2000A701 |  |
| rs.Update | 3000BC01~3000BC01 | Read lock on index page for Customers |  |  |
|  | 3000BE01~3000BE01 | Read lock on index page for Customers |  |  |
| dbFreeLocks |  | 3000BE01 is removed, 3000BC01 remains and will not be freed by **dbFreelocks** |  | Does nothing |
| rs.Update | 2000AE00~2000AF00 | Write lock on data page for Customers | 2000A801~2200A901 | Write lock on data page for Customers |
|  | 2000BE00~2000BF00 | Write lock on index page for Customers | 2000B601~2000B701 | Write lock on index page for Customers |
| Continue looping |  |  |  |  |
|  | 2000B000~2000B100 | Write lock on data page for Customers | 2000AA01~2000AB01 | Write lock on data page for Customers |
|  | 2000B200~2000B300 | Write lock on data page for Customers | 2000AC01~2000AD01 | Write lock on data page for Customers |
|  | 2000B400~2000B500 | Write lock on data page for Customers | 2000AE01~2000AF01 | Write lock on data page for Customers |
|  | 2000B600~2000B700 | Write lock on data page for Customers | 2000BA01~2000BB01 | Write lock on index page for Customers |
|  | 2000B800~2000B900 | Write lock on data page for Customers | 2000B001~2000B101 | Write lock on data page for Customers |
|  | 3000C001~3000C001 | Read lock on index page for Customers | 2000B201~2000B301 | Write lock on data page for Customers |
|  | 2000C000~2000C100 | Write lock on index page for Customers |  |  |

The above log illustrates the reduction in read locks from Microsoft Jet 2*.x* to Microsoft Jet 3.0 and 3.5. Most importantly, the lock log demonstrates how indexes can greatly affect concurrency. In all three versions of Microsoft Jet, there are three index pages that were locked in the Customers table that referenced eight data pages. Thus, by updating one index column, a user could be locking out many other users who are trying to update values on different data pages. **Remember, while indexes can increase retrieval performance, it is very important to weigh the cost of indexes when you are using a multiuser system. By following the motto “If in doubt, index,” you may experience substantial performance degradation and concurrency conflicts**.

Coming up with the best balance of indexed and non-indexed columns is an art and is different for every application and database. One rule of thumb: Don’t index columns that have a high duplication factor. An example of this is indexing a Customer Type field on a Customers table where there are only four unique customer types in the Customers table, which has 100,000 records. Indexing this field would typically not increase performance on SQL SELECT statements and would cause performance (maintaining the index) and concurrency issues (each highly duplicated index page would reference many data pages) on SQL DML statements. The developer and system administrator need to weigh the importance of retrieval time (SQL SELECT statements) versus online transaction processing (OLTP) time (SQL DML statements) and come up with a proper balance of indexes that provides the fastest retrieval times with the fastest OLTP times.

### Inserting Records

The following code example and lock log illustrate the drastic improvement in the locking algorithms used in Microsoft Jet 3.0 and 3.5. While Microsoft Jet 2*.x* would require a lock on the last data page and the TBH during a record insert, Microsoft Jet 3.0 and 3.5 only require a lock on the last data page. Also, if Microsoft Jet 3.0 or 3.5 can’t acquire a lock on the last data page, it will continue seeking available data pages allocated to that table until it can successfully acquire a lock. This eliminates locking contentions when multiple users insert records into the same table at the same time. The exception to this is when indexes are present on the table, because the user might experience some conflicts while modifying the index page.

Also note the keyword **dbFailOnError** after the SQL string. This is absolutely necessary if a user wants to have an error returned. This was not supported in Visual Basic 3.*x*, and it prevented many users from using SQL in multiuser environments because locking conflicts could not be trapped.

Sub InsertRows ()

' This code example is intended to be run from Access 7.0/8.0 or VB

' 4.0/5.0.

Dim db As Database

db.Execute \_

"INSERT INTO Shippers ([Company Name]) VALUES ('Test')", \_

dbFailOnError

End Sub

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **DAO command** | **Microsoft Jet 2*.x* lock hex** | **Description** | **Microsoft Jet 3.0 and 3.5 lock hex** | **Description** |
| INSERT | 20041A00~20041B00 | Write lock on table header page. | 2003BE01~2003BF01 | Write lock on last data page. |
|  | 20041C00~20041D00 | Write lock on last data page of Shippers table. | 2003C401~2003C501 | Write lock on PK index page for counter column. |
|  | 20041E00~20041F00 | Write lock on PK index page for counter column in Shippers table. | 4003BA01~4003BA01 | Table-read lock on Shippers table. |
|  | 30041E00~30041E00 | Read lock on PK index page for counter column in Shippers table. | 5003BA01~5003BA01 | Table-write lock on Shippers table. |
|  | 40041A01~40041A01 | Table-read lock on Shippers table. |  |  |
|  | 50041A01~50041A01 | Table-write lock on Shippers table. |  |  |

### Inserting Records with a Counter

The following code example and lock log illustrate some of the types of locks placed when inserting records with a **Counter** data type.

Sub InsertCounter()

' This code example is intended to be run from Access 7.0/8.0 or VB

' 4.0/5.0.

Dim db As Database

Dim ws As Workspace

Dim SQLStr As String

Set ws = Workspaces(0)

SQLStr = "INSERT INTO Categories ([Category Name], Description, "

SQLStr = SQLStr & "Picture) VALUES ('CounterLock', 'CounterLock', NULL)"

ws.BeginTrans

db.Execute SQLStr, dbFailOnError

ws.CommitTrans

End Sub

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **DAO command** | **Microsoft Jet 2*.x* lock hex** | **Description** | **Microsoft Jet 3.0 and 3.5 lock hex** | **Description** |
| INSERT | 20003C00~20003D00 | Write lock on TBH page for Categories table. | 20004601~20004701 | Write lock on data page for Categories table. |
|  | 20004A00~20004B00 | Write lock on data page for Categories table. | 20009C01~20009D01 | Write lock on index page for Categories table. |
|  | 2000A000~2000A100 | Write lock on index page for Categories table. | 20009E01~20009F01 | Write lock on index page for Categories table. |
|  | 2000A200~2000A300 | Write lock on index page for Categories table. |  |  |
|  | 40003C01~40003C01 | Shared read lock on Categories table. | 40003401~40003401 | Shared read lock on Categories table. |
|  | 50003C01~50003C01 | Shared write lock on Categories table. | 50003401~50003401 | Shared write lock on Categories table. |

### Deleting Records

The following code example and lock log demonstrate the types of locks placed when doing a delete.

Sub DeleteRow()

' This code example is intended to be run from Access 7.0/8.0 or VB

' 4.0/5.0.

Dim db As Database

Dim ws As Workspace

Set ws = Workspaces(0)

ws.BeginTrans

db.Execute \_

"DELETE ROWS FROM Customers WHERE [Customer Id] = 'Paris'", \_

dbFailOnError

DBEngine.Idle dbFreeLocks ' Works only for Jet 2.x.

ws.CommitTrans

End Sub

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **DAO command** | **Microsoft Jet 2*.x* lock hex** | **Description** | **Microsoft Jet 3.0 and 3.5 lock hex** | **Description** |
| DELETE | 20003400~20003500 | Write lock on data page for table MSysObjects. | 20002E01~20002F01 | Write lock on data page for table MSysObjects. |
|  | 2000A600~2000A700 | Write lock on TBH page for Customers table. | 2000AC01~2000AD01 | Write lock on data page for table Customers. |
|  | 2000B400~2000B500 | Write lock on data page for Customers table. | 2000B401~2000B501 | Write lock on index page for table Customers. |
|  | 2000BA00~2000BB00 | Write lock on index page for Customers table. | 2000B601~2000B701 | Write lock on index page for table Customers. |
|  | 2000BE00~2000BF00 | Write lock on index page for Customers table. | 2000B801~2000B901 | Write lock on index page for table Customers. |
|  | 2000C200~2000C300 | Write lock on index page for Customers table. | 2000BC01~2000BD01 | Write lock on index page for table Customers. |
|  | 2000C400~2000C500 | Write lock on index page for Customers table. | 2000BE01~2000BF01 | Write lock on index page for table Customers. |
|  |  |  | 200CE201~200CE301 | Write lock on data page for table MSysObjects. |
|  |  |  | 200CE401~200CE501 | Write lock on long value page for temporary query. |
|  | 3000BA01~3000BA01 | Read lock on index page for table Customers. |  |  |
|  | 3000BE01~3000BE01 | Read lock on index page for table Customers. |  |  |
|  | 3000C201~3000C201 | Read lock on index page for table Customers. |  |  |
|  | 3000C401~3000C401 | Read lock on index page for table Customers. |  |  |
|  | 4000A601~4000A601 | Shared read lock on Customers TBH. | 4000A201~4000A201 | Shared read lock on Customers TBH. |
|  | 5000A601~5000A601 | Shared write lock on Customers TBH. | 5000A201~5000A201 | Shared write lock on Customers TBH. |
|  |  |  | 400B8A01~400B8A01 | Shared read lock on Orders TBH. |

### Updating Records in a Multiple-Table Join

The following code example and lock log illustrate that write locks are placed on all tables involved in the join, even when only one field from one table is being modified:

Sub UpdateJoin ()

Dim db As Database

Dim rs As Recordset

Dim SQLStr As String

Dim ContactName As String

SQLStr = "SELECT \* FROM Customers AS C INNER JOIN "

SQLStr = SQLStr & "(Orders AS O INNER JOIN "

SQLStr = SQLStr & "[Order Details] AS OD ON "

SQLStr = SQLStr & "OD.[Order Id] = O.[Order Id]) ON "

SQLStr = SQLStr & "C.[Customer Id] = O.[Customer Id] "

Set rs = db.OpenRecordset(SQLStr, dbOpenDynaset)

rs.Edit

ContactName = rs![Contact Name]

rs![Contact Name] = ContactName

rs.Update

rs.Close

End Sub

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **DAO command** | **Microsoft Jet 2*.x* lock hex** | **Description** | **Microsoft Jet 3.0 and 3.5 lock hex** | **Description** |
| OpenRecordset | 4000A601~4000A601 | Shared table-read lock on Customers table. | 4000A201~4000A201 | Shared table-read lock on Customers table. |
|  | 40026201~40026201 | Shared table-read lock on Order Details table. | 40024801~40024801 | Shared table-read lock on Order Details table. |
|  | 40031C01~40031C01 | Shared table-read lock on Orders table. | 40020C01~40020C01 | Shared table-read lock on Orders table. |
|  | 5000A601~5000A601 | Shared table-write lock on Customers table. |  |  |
|  | 50026201~50026201 | Shared table-write lock on Order Details table. |  |  |
|  | 50031C01~50031C01 | Shared table-write lock on Orders table. |  |  |
| rs.Edit | 2000AC00~2000AD00 | Write lock on data page for Customers table. | 2000A601~2000A701 | Write lock on data page for Customers table. |
|  | 2002A000~2002A100 | Write lock on data page for Orders Details table. | 20029801~20029901 | Write lock on data page for Orders Details table. |
|  | 2037C00~2037D00 | Write lock on data page for Order table. | 200365401~20035501 | Write lock on data page for Order table. |
| rs.Update |  | All write locks removed. |  | All write locks removed. |

## Deciphering Read and Commit Locks (Microsoft Jet 2.x)

As stated previously, read and commit locks only occur in the 30 million hex range. Read locks always start and end on the same byte and commit locks start and end between 256 and 512 bytes. To determine what page is being locked, use the 32-bit VB 4.0/5.0 DBLock program included on the CD-ROM. To determine the user number, simply take the last two digits of the ending lock range and convert them to decimal.

## Deciphering Read and Commit Locks (Microsoft Jet 3.0 and 3.5)

To determine which page is being locked, use the DBLock program. To determine the user number, simply take the last two digits of the ending lock range and convert them to decimal.

Following is a code sample and lock log file of read locks that were placed while reading a picture from the NWIND database:

Sub ReadLocks ()

'This example is intended to be run from Access 7.0/8.0 or VB 4.0/5.0.

Dim Photo As Variant

Dim db As Database

Dim rs As Recordset

Set db = OpenDatabase("NWIND.MDB", False, True)

Set rs = db.OpenRecordset \_

("SELECT \* FROM Employees", dbOpenDynaset)

While Not rs.EOF

Photo = rs!Photo

'Only use the DBFreeLocks call below when using Jet 2.x.

DBEngine.Idle DBFreeLocks

rs.MoveNext

Wend

rs.Close

End Sub

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **DAO command** | **Microsoft Jet 2*.x* lock hex** | **Description** | **Microsoft Jet 3.0 and 3.5 lock hex** | **Description** |
| OpenDatabase | 10000001~10000001 | User lock | 10000001~10000001 | User Lock |
| OpenRecordset | 4000CC01~4000CC01 | Shared table-read lock. | 4000C801~4000C801 | Shared table-read lock. |
|  | 5000CC01~5000CC02 | Shared table-write lock. |  |  |
| Photo = rs!Photo | 3000D001~3000D001 | Read lock on long value. | 20063801~20063801 | Read lock on long value. |
| DBEngine.Idle dbFreeLocks |  | Lock is removed. |  | Lock is persistent until rs.MoveNext. dbFreeLocks is an obsolete command in Microsoft Jet 3.0 and 3.5. |
| Repeat loop until EOF. | 3000E801~3000E801 | Read lock on next long value. | 2000E601~2000E601 | Read lock on next long value. |
|  | 30010001~30010001 | “” | 2000FC01~2000FC01 | “” |
|  | 30011801~30011801 | “” | 20011201~20011201 | “” |
|  | 30013001~30013001 | “” | 20012801~20012801 | “” |
|  | 30014801~30014801 | “” | 20013E01~20013E01 | “” |
|  | 30016001~30016001 | “” | 20015401~20015401 | “” |
|  | 30017801~30017801 | “” | 20016301~20016301 | “” |
|  | 30018E01~30018E01 | “” | 20018201~20018201 | “” |
|  | 3001A401~3001A401 | “” | 20019801~20019801 | “” |
|  | 3001BC01~3001BC01 | “” | 2001AE01~2001AE01 | “” |
|  | 3001D401~3001D401 | “” | 2001C401~2001C401 | “” |
|  | 3001EC01~3001EC01 | “” | 2001DA01~2001DA01 | “” |
|  | 30020601~30020601 | “” | 2001F201~2001F201 | “” |
|  | 30021E01~30021E01 | “” | 20020A01~20020A01 | “” |

## Deciphering All Types of Table Locks

As stated previously, table locks occur in three ranges: 40, 50, and 60 million hex. Using the Customers table as an example, and assuming that the user below is the first user to connect to the database in shared mode, the following steps determine which table is being locked:

1. Open the Customers table with the **OpenRecordset** method, using the shared table-read lock at 4000A601 as a reference.
2. Drop the first digit (4000A601 à A601).
3. Drop the last two digits (A601 à A6).
4. Divide by two (A6 hex / 2 hex = 53 hex) and convert to decimal (83) to get the table ID.
5. Browse MSysObjects.Id for value 83 and then look at MSysObjects.Name in the records identified to determine the table name.

The same information can be obtained by using the 32-bit Visual Basic 4.0/5.0 utility included on the CD-ROM.

Below are the lock logs for the following commands:

Sub TableDenyRead ()

'This example is intended to be run from Access 7.0/8.0 or VB 4.0/5.0.

Dim db As Database

Dim rs As Recordset

Set db = OpenDatabase("NWind.mdb", False, False)

Set rs = db.OpenRecordset("Customers", dbOpenTable, dbDenyRead)

End Sub

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **DAO command** | **Microsoft Jet 2*.x* lock hex** | **Description** | **Microsoft Jet 3.0 and 3.5 lock hex** | **Description** |
| OpenDatabase | 10000001~10000001 | User lock | 10000001~10000001 | User lock |
| OpenRecordset | 4000A600~4000A700 | Exclusive table-read lock | 4000A201~4000A301 | Exclusive table-read lock |

Sub TableDenyWrite ()

'This example is intended to be run from Access 7.0/8.0 or VB 4.0/5.0.

Dim db As Database

Dim rs As Recordset

Set db = OpenDatabase("NWind.mdb", False, False)

Set rs = db.OpenRecordset("Customers", dbOpenTable, dbDenyWrite)

End Sub

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **DAO command** | **Microsoft Jet 2*.x* lock hex** | **Description** | **Microsoft Jet 3.0 and 3.5 lock hex** | **Description** |
| OpenDatabase | 10000001~10000001 | User lock | 10000001~10000001 | User lock |
| OpenRecordset | 4000A601~4000A601 | Shared table-read lock | 4000A201~4000A201 | Shared table-read lock |
|  | 5000A601~5000A601 | Shared table-write lock | 5000A201~5000A201 | Shared table-write lock |
|  | 6000A601~6000A601 | Table deny-write lock | 6000A201~6000A201 | Table deny-write lock |

## Multiuser Enhancements Specific to Microsoft Jet 3.5

Many new features were added to Microsoft Jet 3.5 (more detailed information on these topics can be found in the “Microsoft Jet 3.5 Performance Overview and Optimization Techniques” white paper).

### Removal of Implicit Transactions for SQL DML Statements

Microsoft Jet 2*.x* and 3.0 place implicit transactions around all SQL DML statements. When issuing DML statements that affected thousands of records, the following issues could arise:

* If the Microsoft Jet database resided on a NetWare server, the SQL DML statement could fail due to Microsoft Jet exceeding the maximum number of locks. Depending on the workstation OS, this could result in the application appearing to hang as Microsoft Jet would constantly retry to acquire locks that the NetWare server would not allow.
* As Microsoft Jet’s cache was exceeded, the transaction would start writing the modified pages to Microsoft Jet’s temporary database. This would cause a performance hit due to the extra I/O and would result in the locks being held on pages in the database for a longer period of time. This ultimately reduces concurrency for the application.
* Microsoft Jet’s implicit transactions would hold locks for the duration of the transaction (identical to using an explicit transaction via DAO’s **BeginTrans** and **CommitTrans** methods) thus reducing concurrency for other users.

The above issues are resolved by only using internal transactions for SQL DML statements. If an explicit transaction is still desired, the application will have to be modified to use the DAO’s **BeginTrans** and **CommitTrans** methods. The new implementation significantly increases performance and concurrency.

### New Registry Settings

The new settings discussed below are pertinent to multiuser applications.

#### MaxLocksPerFile

The setting is designed to limit the number of locks a Microsoft Jet transaction can place before splitting and committing the transaction. The main reason for implementing this setting was to alleviate issues with users running against NetWare servers where there was a 10,000 lock per workstation limit. The default for this setting is 9500 and may be needed to be adjusted down to accommodate transactions that update long value (Memo and OLE Object data types) and indexed fields. This setting will now allow users to complete transactions that place locks exceeding the NetWare limit without resorting to looping DAO code.

#### FlushTransactionTimeout

This new registry setting overrides the previously used SharedAsyncDelay registry setting. It was designed to increase performance for Microsoft Jet’s internal transactions by significantly reducing I/O. This is accomplished by fully utilizing Microsoft Jet’s cache, but never writing to Microsoft Jet’s temporary database. While this design, in most instances, significantly increase performance, it will also increases the persistence of locks. This may result in decreased concurrency. If your application consists of multiple workstation that utilize DAO code to rapidly update one record at a time, then it may be advisable to disable this functionality. This can be done by changing the default value of 500 to 0. This will set Microsoft Jet’s internal transaction behavior to be dependent on the SharedAsyncDelay registry setting.

### New DAO Methods and Options

While there are many other new DAO methods and options, the ones discussed below are pertinent to multiuser applications.

#### CommitTrans dbForceOSFlush

When Microsoft Jet moved to a 32-bit DLL, a severe performance issue arose when running on Windows 95 and Windows NT due to the fact that they were honoring Microsoft Jet’s call to the operating system to issue synchronous writes. With the 16-bit versions of Microsoft Jet, this did not prove to be a performance issue because both operating systems would ignore the call from 16-bit applications and issue asynchronous writes to disk. However, both operating systems would acknowledge the call for 32-bit applications and the performance of Microsoft Jet suffered. In order to regain performance, Microsoft Jet had to utilize a different operating system call that did not force synchronous writes. The downside of this is that users implementing explicit transactions could not be guaranteed that information was truly written to disk when they issued the **CommitTrans** method.

Microsoft Jet 3.5 has exposed this functionality again and DAO 3.5 has added an option to the **CommitTrans** method. By using the option **dbForceOSFlush**, Microsoft Jet will issue the call to the operating system, forcing synchronous writes to disk. This gives developers the option of guaranteeing that information has been written to disk before moving to the next line of code. This option has been tested on Windows NT, Windows 95, and NetWare, but is not honored by NetWare.

This new option should be used judiciously because calling it too frequently (for example, in a looping construct) will cause a severe performance degradation.

#### DBEngine.Idle dbRefreshCache

Microsoft Jet 3.0 would only refresh a workstation’s cache when the PageTimeOut registry setting expired. By default, this would occur every five seconds. By using a five second refresh time, I/O was significantly reduced and performance was increased. However, this caused issues for developers that needed their application to see the absolute latest version of the data. The workaround in Microsoft Jet 3.0 was to reduce the PageTimeOut registry setting, but this resulted in a performance hit.

Microsoft Jet 3.5 has resolved this situation by allowing developers to call the **dbRefreshCache** option from the **Idle** method. This will force Microsoft Jet to refresh its cache and display that latest information in the database. Now developers can return the PageTimeOut registry setting to its default to regain the performance gains and refresh the cache on demand when appropriate.

#### DBEngine.SetOption

Microsoft Jet 2*.x* and Microsoft Jet 3.0 relied on either .ini file or registry settings to change the behavior of the engine. This made it difficult to change settings for an application or have user-specific settings and impossible to change settings that were dependent on the type of code being executed. To resolve this problem, Microsoft Jet 3.5 has been enhanced to change registry settings during run time via DAO’s new **SetOption** method. This allows developers to programmatically change the registry settings in their applications and not worry about using customized registry trees or .ini files. This also allows developers to customize and change their applications behavior to take advantage of Microsoft Jet settings that are particular to a specified task.

The registry settings below are useful for controlling multiuser specific situations.

|  |  |
| --- | --- |
| **Registry Name** | **DAO Constant** |
| PageTimeout | **dbPageTimeout** |
| SharedAsyncDelay | **dbSharedAsyncDelay** |
| LockRetry | **dbLockRetry** |
| UserCommitSync | **dbUserCommitSync** |
| ImplicitCommitSync | **dbImplicitCommitSync** |
| MaxBufferSize | **dbMaxBufferSize** |
| MaxLocksPerFile | **dbMaxLocksPerFile** |
| LockDelay | **dbLockDelay** |
| FlushTransactionTimeout | **dbFlushTransactionTimeout** |

## Using the Utilities

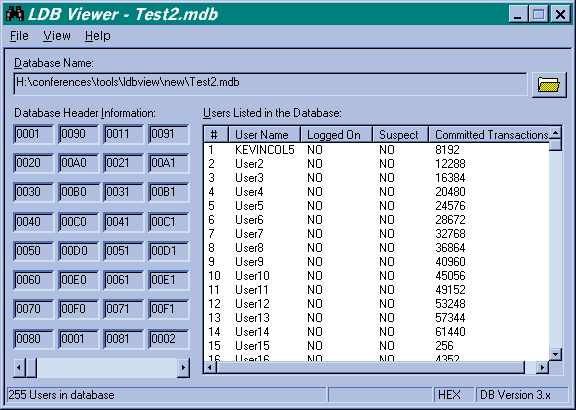
All the utilities referenced in this paper are available on the companion CD-ROM in the Utilities folder.

### LDBView

This utility has been enhanced from the original utility. The utility now displays more information on the screen and also works with low-resolution screen displays.

The utility LDBView (Figure 1) is used to accomplish the following tasks:

* Discover which users have been connected to the database and what users are currently connected to the database
* Discover the values of the commit bytes in the DBH (database header page)
* Determine which user or users have left the database in a suspect state



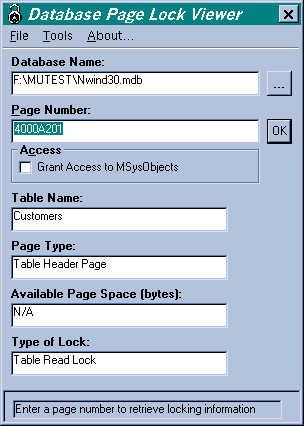
**Figure 1**

LDBView is a 32-bit single .exe utility that can run under either Windows 95 or Windows NT, and can view both Microsoft Jet 2*.x* and Microsoft Jet 3.0 and 3.5 databases. In order for the utility to run, a corresponding .ldb file must be present.

To use the utility, simply use the File menu to select the database that you want to view. After selecting a database, a screen similar to Figure 1 is displayed. At this point, the refresh interval can be modified by using the View menu. Changing these values will determine how often LDBView looks at reads and displays information from the selected database and corresponding .ldb file.

### DBLock

The utility DBLock (Figure 2) determines which types of locks Microsoft Jet is placing, which page types are being locked and which tables are associated with those pages. By using a utility, such as NetWare’s Monitor utility, you can immediately enter a beginning lock range and determine what effect a particular command would have on other users. This utility can also be used to help determine what bottlenecks exist when users experience locking conflicts. This information could then be used to modify the application or database design (by removing unnecessary indexes, for example).



**Figure 2**

DBLock is a Visual Basic 4.0/5.0 32-bit utility that runs under either Windows 95 or Windows NT and can work against both Microsoft Jet 2*.x* and Microsoft Jet 3.0 and 3.5 databases. The Setup program installs the Microsoft Jet DLLs necessary to run this program. By default, DBLock installs a default System.mda (security file for Microsoft Jet 2*.x*) and a default System.mdw (security file for Microsoft Jet 3.0 and 3.5). If you have an unsecured database, the utility will grant rights to MSysObjects for user Admin. If you have a secured database, you will need to use the Tools menu to point to the path of your security database.

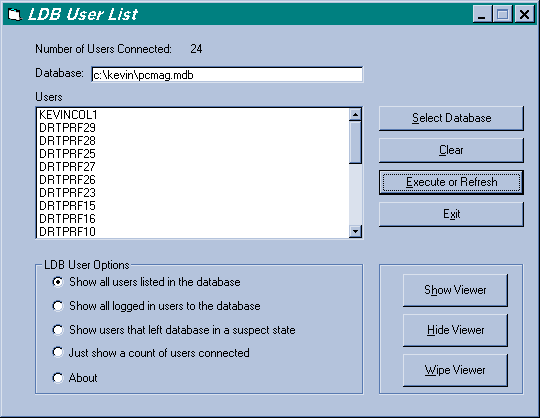
After a database has been selected, simply enter in the hexadecimal values that are displayed (on the NetWare Monitor utility, for example) and the information for the table name, page type, available page space, and type of lock will be populated.

### MSLDBUSR.DLL

This 32-bit DLL enables you to retrieve one of the most sought-after pieces of information: a list of users connected to the database. This was previously impossible to do programmatically unless a programmer understood the user lock algorithm and understood how to place extended byte range locks. Since this is a 32-bit DLL, it will only work with 32-bit products (Microsoft Access 7.0/8.0, Visual Basic 4.0/5.0 32-bit, Microsoft Excel 7.0/8.0, Visual C ++ 32-bit) running on Windows 95 or Windows NT. However, it will retrieve information for both Microsoft Jet 2*.x* and Microsoft Jet 3.0 and 3.5 databases. Now your application can display a list of users who are preventing you from opening the database exclusively or running a repair or compact.

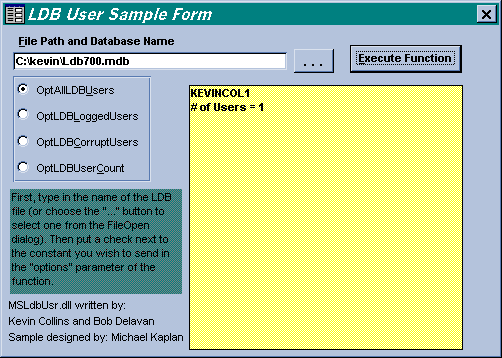
Because the DLL places real-time extended byte range locks, the information that it retrieves can be out of date as soon as another user logs out, but recalling the DLL will always provide current information.

Figure 3 is a screen shot of a sample Visual Basic 4.0/5.0 utility (LDBUSRDLL.VBP) that includes source code on how to use the DLL. Figure 4 shows a screen shot of a sample Microsoft Access 7.0/8.0 database (provided by Michael Kaplan).



**Figure 3**

To use the sample utility above, simply select a database file, choose an option under LDB User Options and click Execute or Refresh, which calls the function LDBUser\_GetUsers. To view error codes and descriptions, click the Show Viewer button, which calls the LDBUser\_GetError function. This utility will not work against library or wizard databases because those are opened exclusively and the Show Users That Left Database In A Suspect State option needs to open the database in a shared mode. For the utility to work properly, it is necessary to have the MSLDBUSR.DLL in the path or the System folder.



**Figure 4**

The above screen shot uses the LDB700.MDB database from Microsoft Access 7.0/8.0. For the utility to work properly, it is necessary to have the MSLDBUSR.DLL in the path or the System folder.

**To use the DLL you must make the following two function declarations. These must go in your module (NOTE: This cannot be a form module) and the DLL must be located in the System directory.**

Declare Function LDBUser\_GetUsers Lib "MSLDBUSR.DLL" \_

(lpszUserBuffer() As String, ByVal lpszFilename As String, \_

ByVal nOptions As Long) As Integer

Declare Function LDBUser\_GetError Lib "MSLDBUSR.DLL" \_

(ByVal nErrorNo As Long) As String

The following options can be called from the function LDBUser\_GetUsers:

|  |  |  |
| --- | --- | --- |
| Public Const OptAllLDBUsers | &H1 | This option returns a list of users in the .ldb file. This option opens the .ldb file and returns each user one by one and then adds them to the array list. This option does not check to see if the users currently have the database open. It essentially represents users who at one time or another had the database open. It also does not represent all users who have ever had the database open. |
| Public Const OptLDBLoggedUsers | &H2 | Shows all .ldb users who currently have the database open. This is similar to OptAllLDBUsers, except it checks to make sure that the users listed in the .ldb file have the database open. Often, OptAllLDBUsers and OptLDBLoggedUsers show the same list. |
| Public Const OptLDBCorruptUsers | &H4 | Shows all users who left the database in a suspect state. |
| Public Const OptLDBUserCount | &H8 | Returns a list of users connected to the database. (NOTE: You still must include an array and .ldb file name.) |
| Public Const OptLDBUserAuthor | &HB0B | Invokes the credits screen when using -Z as the .ldb file name. |

Here is some sample VBA code that will get the first user in the .ldb file:

' The DLL must be in the System directory or have a path referencing it.

Declare Function LDBUser\_GetUsers Lib "MSLDBUSR.DLL" \_

(lpszUserBuffer() As String, ByVal lpszFilename As String, \_

ByVal nOptions As Long) As Integer

Declare Function LDBUser\_GetError Lib "MSLDBUSR.DLL" \_

(ByVal nErrorNo As Long) As String

Public Const OptAllLDBUsers = &H1

Public Const OptLDBLoggedUsers = &H2

Public Const OptLDBCorruptUsers = &H4

Public Const OptLDBUserCount = &H8

Public Const OptLDBUserAuthor = &HB0B

Sub Main()

' It is important that ReDim be used to define the array as the DLL,

' because the DLL depends on being able to redimension the array.

ReDim msString(1) As String

' The array is 1-based rather than 0-based, regardless if Option Base 1

' is specified in the declarations section.

Dim miLoop As Integer

Dim LDBReturn As String

Dim LDBName As String

LDBName = InputBox("Enter an LDB name", " ", "c:nwind.mdb")

miLoop = LDBUser\_GetUsers(msString, LDBName, OptAllLDBUsers)

' The function calls cannot be combined and must be used individually.

' Get the first user in the selected .LDB file.

LDBReturn = msString(1)

MsgBox ("Total number of users in .LDB file is: " & miLoop & \_

". The first computer name in .LDB file is " \_

& LDBReturn & ".")

End Sub

 1997 Microsoft Corporation. All rights reserved.

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, this paper should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This document is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS OR IMPLIED, IN THIS DOCUMENT.

Microsoft, Windows, NT, Visual Basic, and Visual C++ are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.