**Memory-Mapped I/O**

The default mechanism by which SQLite accesses and updates database disk files is the xRead() and xWrite() methods of the [sqlite3\_io\_methods](https://www.sqlite.org/c3ref/io_methods.html) VFS object. These methods are typically implemented as "read()" and "write()" system calls which cause the operating system to copy disk content between the kernel buffer cache and user space.

Beginning with [version 3.7.17](https://www.sqlite.org/releaselog/3_7_17.html) (2013-05-20), SQLite has the option of accessing disk content directly using memory-mapped I/O and the new xFetch() and xUnfetch() methods on [sqlite3\_io\_methods](https://www.sqlite.org/c3ref/io_methods.html).

There are advantages and disadvantages to using memory-mapped I/O. Advantages include:

1. Many operations, especially I/O intensive operations, can be faster since content need not be copied between kernel space and user space.
2. The SQLite library may need less RAM since it shares pages with the operating-system page cache and does not always need its own copy of working pages.

But there are also disadvantages:

1. An I/O error on a memory-mapped file cannot be caught and dealt with by SQLite. Instead, the I/O error causes a signal which, if not caught by the application, results in a program crash.
2. The operating system must have a unified buffer cache in order for the memory-mapped I/O extension to work correctly, especially in situations where two processes are accessing the same database file and one process is using memory-mapped I/O while the other is not. Not all operating systems have a unified buffer cache. In some operating systems that claim to have a unified buffer cache, the implementation is buggy and can lead to corrupt databases.
3. Performance does not always increase with memory-mapped I/O. In fact, it is possible to construct test cases where performance is reduced by the use of memory-mapped I/O.
4. Windows is unable to truncate a memory-mapped file. Hence, on Windows, if an operation such as [VACUUM](https://www.sqlite.org/lang_vacuum.html) or [auto\_vacuum](https://www.sqlite.org/pragma.html" \l "pragma_auto_vacuum) tries to reduce the size of a memory-mapped database file, the size reduction attempt will silently fail, leaving unused space at the end of the database file. No data is lost due to this problem, and the unused space will be reused again the next time the database grows. However if a version of SQLite prior to 3.7.0 runs [PRAGMA integrity\_check](https://www.sqlite.org/pragma.html#pragma_integrity_check) on such a database, it will (incorrectly) report database corruption due to the unused space at the end. Or if a version of SQLite prior to 3.7.0 writes to the database while it still has unused space at the end, it may make that unused space inaccessible and unavailable for reuse until after the next [VACUUM](https://www.sqlite.org/lang_vacuum.html).

Because of the potential disadvantages, memory-mapped I/O is disabled by default. To activate memory-mapped I/O, use the [mmap\_size pragma](https://www.sqlite.org/pragma.html" \l "pragma_mmap_size) and set the mmap\_size to some large number, usually 256MB or larger, depending on how much address space your application can spare. The rest is automatic. The [PRAGMA mmap\_size](https://www.sqlite.org/pragma.html#pragma_mmap_size) statement will be a silent no-op on systems that do not support memory-mapped I/O.

**How Memory-Mapped I/O Works**

To read a page of database content using the legacy xRead() method, SQLite first allocates a page-size chunk of heap memory then invokes the xRead() method which causes the database page content to be copied into the newly allocated heap memory. This involves (at a minimum) a copy of the entire page.

But if SQLite wants to access a page of the database file and memory mapped I/O is enabled, it first calls the xFetch() method. The xFetch() method asks the operating system to return a pointer to the requested page, if possible. If the requested page has been or can be mapped into the application address space, then xFetch returns a pointer to that page for SQLite to use without having to copy anything. Skipping the copy step is what makes memory mapped I/O faster.

SQLite does not assume that the xFetch() method will work. If a call to xFetch() returns a NULL pointer (indicating that the requested page is not currently mapped into the applications address space) then SQLite silently falls back to using xRead(). An error is only reported if xRead() also fails.

When updating the database file, SQLite always makes a copy of the page content into heap memory before modifying the page. This is necessary for two reasons. First, changes to the database are not supposed to be visible to other processes until after the transaction commits and so the changes must occur in private memory. Second, SQLite uses a read-only memory map to prevent stray pointers in the application from overwriting and corrupting the database file.

After all needed changes are completed, xWrite() is used to move the content back into the database file. Hence the use of memory mapped I/O does not significantly change the performance of database changes. Memory mapped I/O is mostly a benefit for queries.

**Configuring Memory-Mapped I/O**

The "mmap\_size" is the maximum number of bytes of the database file that SQLite will try to map into the process address space at one time. The mmap\_size applies separately to each database file, so the total amount of process address space that could potentially be used is the mmap\_size times the number of open database files.

To activate memory-mapped I/O, an application can set the mmap\_size to some large value. For example:

PRAGMA mmap\_size=268435456;

To disable memory-mapped I/O, simply set the mmap\_size to zero:

PRAGMA mmap\_size=0;

If mmap\_size is set to N then all current implementations map the first N bytes of the database file and use legacy xRead() calls for any content beyond N bytes. If the database file is smaller than N bytes, then the entire file is mapped. In the future, new OS interfaces could, in theory, map regions of the file other than the first N bytes, but no such implementation currently exists.

The mmap\_size is set separately for each database file using the "[PRAGMA mmap\_size](https://www.sqlite.org/pragma.html#pragma_mmap_size)" statement. The usual default mmap\_size is zero, meaning that memory mapped I/O is disabled by default. However, the default mmap\_size can be increased either at compile-time using the [SQLITE\_DEFAULT\_MMAP\_SIZE](https://www.sqlite.org/compile.html#default_mmap_size) macro or at start-time using the [sqlite3\_config](https://www.sqlite.org/c3ref/config.html)([SQLITE\_CONFIG\_MMAP\_SIZE](https://www.sqlite.org/c3ref/c_config_covering_index_scan.html#sqliteconfigmmapsize),...) interface.

SQLite also maintains a hard upper bound on the mmap\_size. Attempts to increase the mmap\_size above this hard upper bound (using [PRAGMA mmap\_size](https://www.sqlite.org/pragma.html#pragma_mmap_size)) will automatically cap the mmap\_size at the hard upper bound. If the hard upper bound is zero, then memory mapped I/O is impossible. The hard upper bound can be set at compile-time using the [SQLITE\_MAX\_MMAP\_SIZE](https://www.sqlite.org/compile.html#max_mmap_size) macro. If [SQLITE\_MAX\_MMAP\_SIZE](https://www.sqlite.org/compile.html#max_mmap_size) is set to zero, then the code used to implement memory mapped I/O is omitted from the build. The hard upper bound is automatically set to zero on certain platforms (ex: OpenBSD) where memory mapped I/O does not work due to the lack of a unified buffer cache.

If the hard upper bound on mmap\_size is non-zero at compilation time, it may still be reduced or zeroed at start-time using the [sqlite3\_config](https://www.sqlite.org/c3ref/config.html)([SQLITE\_CONFIG\_MMAP\_SIZE](https://www.sqlite.org/c3ref/c_config_covering_index_scan.html#sqliteconfigmmapsize),X,Y) interface. The X and Y parameters must both be 64-bit signed integers. The X parameter is the default mmap\_size of the process and the Y is the new hard upper bound. The hard upper bound cannot be increased above its compile-time setting using [SQLITE\_CONFIG\_MMAP\_SIZE](https://www.sqlite.org/c3ref/c_config_covering_index_scan.html#sqliteconfigmmapsize) but it can be reduced or zeroed.