

latest version of mariadb target and visual studio disable disk based storage function but dll plugin which installable

I will research how to port MariaDB's InnoDB storage engine as a custom DLL plugin on Windows using the latest version of MariaDB and Visual Studio, while restricting it to only in-memory table functionality and disabling disk-based storage.

I'll find details on:

- The process of compiling a MariaDB storage engine as a DLL plugin on Windows.
- How to modify InnoDB to disable disk-based storage while keeping memory-only functionality.
- Necessary build configurations and CMake settings for Visual Studio.
- The installation and integration process of the custom DLL plugin into MariaDB.

I'll update you once I have the findings.

Research completed in 5m · 19 sources

---

# Porting MariaDB's InnoDB Engine as an In-Memory Plugin on Windows

**Overview:** *Creating a custom in-memory variant of InnoDB on Windows involves building the InnoDB storage engine as a dynamic plugin (DLL) and modifying its code to eliminate disk I/O. This ensures all data remains in memory (volatile), trading persistence for performance. Below, we outline how to set up the build environment with Visual Studio, compile InnoDB as a DLL plugin, adjust the source to disable disk usage, configure CMake for a Windows build, and deploy the plugin. We also discuss best practices and challenges when working with the latest MariaDB version and Visual Studio.*

## 1. Setting Up the Build Environment (Latest MariaDB & Visual Studio)

To begin, obtain the **latest MariaDB source code** (e.g. MariaDB 10.11 or 11.x). Ensure your Windows development environment meets MariaDB's requirements for building the server:

- **Visual Studio 2019 or 2022:** MariaDB supports the two most recent Visual Studio versions (at present VS2019 and VS2022)  
[MARIADB.COM](#) . Install the **Desktop Development with C++** workload in Visual Studio [MARIADB.COM](#) , and use the latest Windows SDK (Windows 11 SDK is recommended) [MARIADB.COM](#) for compatibility. Community Edition of VS is sufficient.
- **CMake:** Install a recent CMake release (minimum CMake 3.14 for VS2019) [MARIADB.COM](#) . MariaDB uses CMake to configure the build, including plugin build options.
- **Build Tools:** Install **bison** (for the SQL parser) and **perl** (for tests) as required by MariaDB's build process  
[MARIADB.COM](#) . Also ensure **Git** is available if building from the Git repository [MARIADB.COM](#) (it's used to fetch submodules and for version info).

With the environment ready, **download or clone the MariaDB source** corresponding to the latest version. If using Git, run `git clone` for the official repo and check out the desired version tag or branch. Then proceed to generate a Visual Studio solution using CMake.

## 2. CMake Configuration for a Visual Studio Build

MariaDB's build system is CMake-based, which makes it straightforward to produce a Visual Studio solution or MSBuild files. Key points for configuring the build:

- **Generate the VS Solution:** Run CMake from a *Visual Studio Developer Command Prompt* or via CMake GUI. Specify the VS generator and platform, for example:

```
bash
```



Copy



Edit

```
cmake -G "Visual Studio 17 2022" -A x64 -Hpath\to\mariadb\source -  
Bpath\to\build\dir
```

This will create a Visual Studio 2022 solution for a 64-bit build. You can adjust the generator (e.g., "Visual Studio 16 2019" for VS2019) and platform (Win32 vs x64) as needed. If using the CMake GUI, select the MariaDB source directory and a build output directory, then choose the appropriate Visual Studio generator from the dropdown.

- **Select Plugin Build Options:** By default, **MariaDB enables all plugins as dynamic modules** (shared libraries) if supported [RUNEBOOK.DEV](#) . InnoDB is included in the source ( `storage/innobase` ) and supports dynamic builds, so it will be built as a `.dll` by default unless explicitly overridden. You can verify or force this by setting the CMake option `PLUGIN_INNOBASE` to `DYNAMIC` (dynamic plugin build) [RUNEBOOK.DEV](#) . For example, add `-DPLUGIN_INNOBASE=DYNAMIC` to the CMake command to be explicit. (By contrast, setting `PLUGIN_XYZ=STATIC` would compile a plugin statically into the server, and `NO` would omit it [RUNEBOOK.DEV](#) .)
- **Build Type:** If using the multi-configuration Visual Studio generator, you can select **Debug** or **Release** builds in the IDE. For command-line builds with MSBuild, specify `/p:Configuration=Release` (or `Debug`) when building. For most purposes, a `Release` or `RelWithDebInfo` configuration is desired for an optimized plugin build.

After configuring, CMake will generate a Visual Studio solution (`.sln`) with projects for the server and all plugins. Open the solution in Visual Studio or use MSBuild to compile. Make sure the **InnoDB engine plugin project** is included – it may be named something like `ha_innodb` in the solution. By default, building the solution will produce the server executable and all plugin DLLs under the build directory (e.g., in `plugins` or similar output folder).

### 3. Compiling the InnoDB Storage Engine as a DLL Plugin

With the solution generated, **compile MariaDB**. This will build the `mysqld` server and all plugins, including InnoDB. Since InnoDB is the default storage engine, ensure it indeed builds as a separate plugin library ( `ha_innodb.dll` ). The MariaDB Knowledge Base notes that *“by default all plugins are built as dynamic `.so` (or `.dll`) modules”*

[RUNEBOOK.DEV](#) , so you should get a `ha_innodb.dll` after a successful build.

#### Steps to compile:

1. **Open/Build in Visual Studio:** Open the `.sln` file in Visual Studio. Select the desired configuration (e.g. `Release x64`) and build the solution (`Build > Build All`). This will produce the server and plugin binaries.
2. **Or Build via MSBuild:** From a Developer Command Prompt, run:

```
bash
```



Copy



Edit

```
msbuild MariaDB.sln /p:Configuration=Release /p:Platform=x64
```

This compiles the project without the Visual Studio GUI.

3. **Locate the Plugin DLL:** After building, find `ha_innodb.dll` in the output. Typically, it will be in a `lib/plugin` subdirectory of the build output. (If not immediately obvious, search the build directory for "ha\_innodb.dll"). The presence of this file confirms that the InnoDB engine was built as a plugin library.

**Note:** If you plan to create a *separate custom engine* (with a different name to avoid confusion with the built-in InnoDB), you could copy the `storage/innobase` source to a new directory (e.g. `storage/innodb_mem`) and adjust names/IDs. However, an easier approach is to modify the existing InnoDB code in place (see next section) and compile it as the plugin. You can later choose whether to replace the default InnoDB or register it as a separate engine.

At this stage, you have a build environment ready and the InnoDB plugin compiled as a DLL. Next, we'll modify the InnoDB source to strip out disk-based storage and enable memory-only operation.

## 4. Modifying InnoDB Source for In-Memory Only Operation

By default, InnoDB is a disk-based transactional engine – it writes data and logs to files for durability. Since our goal is an **in-memory only** engine, we must **disable or remove all disk I/O** from InnoDB while preserving its in-memory data handling (indexes, transactions, etc.). This requires changes in the InnoDB source code. Key modifications include:

- **Stub out File Operations:** InnoDB's low-level file I/O is handled in modules like `os0file.cc` and `fil0fil.cc`. These manage tablespace files (`ibdata*`, `.ibd` files) and log files (`ib_logfile*`). To prevent any actual disk activity, alter these routines to use memory-backed storage or do nothing. For example:
  - Modify file-create and file-open functions to **not create physical files**. One approach is to allocate a chunk of memory to represent the data file. For instance, when InnoDB tries to create `ibdata1` or the log file, allocate a large memory buffer and store it in a structure instead of creating a file on disk. On Windows, you could use memory-mapped files backed by the pagefile (e.g. via `CreateFileMapping` with an invalid handle to allocate memory) to simulate a file in RAM.

- Modify read/write functions to operate on that memory buffer. Essentially, implement a **fake file system in memory**: reads fetch from the buffer, writes go to the buffer. This way, all table and index data stays in process memory.
- **Disable Flush and Sync Operations:** InnoDB normally flushes dirty pages from the buffer pool to disk and fsyncs files for durability. In an in-memory engine, these are unnecessary (and we have no disk to flush to). Identify background threads and functions responsible for flushing (e.g., `buf_flush_page()` in `buf0flu.cc`, the master thread in `srv0srv.cc` that does periodic flush, and the log flushing in `log0log.cc`). Alter these so they either **do nothing or return success immediately**. Essentially, **skip writing any data to disk**. For example, you might short-circuit the flush function to just mark pages as “not dirty” without writing, or simply disable the flush thread loop. Similarly, stub out calls that fsync log or data files (so they don’t block on I/O).
- **Bypass Redo Logging and Recovery:** InnoDB’s redo log (`ib_logfiles`) exists to recover after crash by replaying transactions. If we never write logs to disk, we should also disable the expectation of recovery. You can **turn off redo logging** by making log write functions no-ops. Also ensure that on startup the engine doesn’t attempt to recover a previous state. One strategy is to **treat each engine start as a fresh instance** – always initialize the storage as if it’s a new database. This means if the engine finds no files (which in our case it won’t persist any), it should create the necessary in-memory structures without error. In code, this might involve forcing the **creation of a new system tablespace** in memory on each startup and skipping any recovery routine (`recv0recover.cc`).
- **Constrain Table Persistence:** InnoDB normally stores metadata in the system tablespace and expects tables to persist across restarts. With no disk storage, any data will vanish on shutdown, but MariaDB’s higher-level data dictionary might still have table definitions. It’s best to document that **tables using this engine are ephemeral**. For safety, you might mark the engine as not supporting persistent storage. For instance, in the handler (engine descriptor), set flags indicating it’s a *volatile or memory-only engine*. There isn’t a built-in flag specifically for “memory only” in MySQL/MariaDB API, but you could treat it similarly to how the MEMORY engine behaves (data not persisted). Users should be aware that after restarting MariaDB, InnoDB-memory tables will be empty or may need to be recreated.
- **Memory Management:** Ensure that any memory allocated for the faux files or buffer pool is freed on plugin shutdown to avoid leaks if the plugin is unloaded. InnoDB will already free buffers on a clean shutdown; just verify that your added allocations (if any) are freed. If the engine is not unloaded until server exit, OS will reclaim memory, but it’s good practice to clean up.

These changes essentially transform InnoDB into a **RAM-only engine**. All storage resides in the buffer pool or allocated memory areas, and no writes hit the disk. Keep in mind that InnoDB was not originally designed for this mode, so careful testing is needed.

**Important:** There is no built-in configuration option to achieve pure in-memory mode for InnoDB – normally, you *cannot make an InnoDB table behave like a MEMORY table without modifications*

[STACKOVERFLOW.COM](https://stackoverflow.com) . (Using TEMPORARY tables or setting `innodb_flush_log_at_trx_commit=0`, etc., still involves some disk I/O.) Therefore, code modifications are the only route to get true memory-only behavior.

After making these code changes, rebuild the MariaDB solution to produce an updated `ha_innodb.dll` reflecting the in-memory logic.

## 5. Building and Packaging the Custom Plugin

Once the code is modified and compiled, you will have a **custom InnoDB plugin DLL**. The next steps are packaging it for use and installing it into MariaDB:

- **Isolate the Plugin Binary:** The `ha_innodb.dll` (or whatever name you used if renamed) is the plugin library. If you built it as part of the whole MariaDB build, you can extract just this DLL (and possibly any associated files, though typically a storage engine plugin is a single binary).
- **Place in MariaDB Plugin Directory:** Copy the DLL into MariaDB's plugin directory. By default on Windows, this is the `lib\plugin` folder under the MariaDB installation path [DBA.STACKEXCHANGE.COM](https://dba.stackexchange.com) (you can confirm the path with `SHOW VARIABLES LIKE 'plugin_dir'`; in MariaDB). For example, if MariaDB is installed in `C:\Program Files\MariaDB 10.11\`, place the DLL in `C:\Program Files\MariaDB 10.11\lib\plugin\`. This makes it accessible for the server to load.
- **Update Configuration to Load the Plugin:** If you want your custom engine to replace the built-in InnoDB, update the server config to **load it at startup**. You should also disable the normal InnoDB to avoid conflicts. In `my.ini` (under the `[mysqld]` section), add lines such as:

```
ini
```



Copy



Edit

```
ignore_builtin_innodb plugin_load_add = innodb=ha_innodb.dll
```

The first line tells MariaDB to skip the built-in InnoDB engine, and the second explicitly loads your plugin, naming it "InnoDB" engine and pointing to the DLL

[DBA.STACKEXCHANGE.COM](http://DBA.STACKEXCHANGE.COM) . (These lines reflect the case where you kept the plugin name as "InnoDB". If you gave your engine a new name, use that name accordingly in `plugin_load`.) On startup, the server will load your DLL and register it as the InnoDB storage engine.

Alternatively, if you chose a new engine name (say "INNODB\_MEM" ), you would use `plugin_load_add=innodb_mem=ha_innodb.dll` and possibly `set default_storage_engine=innodb_mem` for convenience. You could also **install the plugin at runtime** with SQL: for example, `INSTALL PLUGIN innodb_mem SONAME 'ha_innodb.dll'`; which registers the engine plugin with the given name.

- **Verify Plugin Installation:** Start MariaDB and run `SHOW PLUGINS;` or `SHOW ENGINES;` to ensure your plugin is listed as active. You should see "InnoDB" (or your engine name) with a **YES** in the "Support" column and possibly "Active" status. If you replaced the built-in, `SHOW ENGINES` should show InnoDB as the plugin (and possibly XtraDB disabled, depending on version). If any errors occur (e.g., undefined symbols or init function errors), double-check that the plugin DLL is built for the exact MariaDB version of the server – mismatches can cause load failure.
- **Usage:** Create a table using the engine to test it. For example: `CREATE TABLE t1 (id INT PRIMARY KEY) ENGINE=InnoDB;` . Inserts to this table should not produce any disk files. You can monitor the data directory; no `.ibd` files for `t1` should appear (since our engine does not write them). The table's data exists only in memory. Performance for memory-only workloads should be high, but remember the data will not survive a restart.

## 6. Best Practices and Potential Challenges

Creating a custom in-memory InnoDB plugin is an advanced task. Keep these best practices and caveats in mind:

- **Stay Up-to-Date:** Align your work with the *latest MariaDB source* to maintain compatibility. MariaDB's internal APIs (for storage engines) can change over versions. Using the exact version source to build the plugin that matches your MariaDB server version is crucial – a plugin built for MariaDB X may not load or function correctly on MariaDB Y if there are differences in the handler interface or internal structures.

- **Testing and Stability:** Rigorously **test the engine** under various scenarios. Without disk backing, certain InnoDB features may behave differently. For example, transactions will not survive a crash, and the crash-recovery code is effectively disabled. Test create/drop tables, large data loads (to see how memory usage grows), and multi-user concurrency. Pay attention to error handling – if the engine runs out of its pre-allocated memory (since no auto-extending file), it should gracefully throw an “out of space” error rather than crash.
- **Memory Management:** Since everything is in memory, the size of your InnoDB buffer pool or the allocated “tablespace” buffer effectively limits your database size. It might be wise to configure a large buffer pool (e.g., 70-80% of system RAM) because the buffer pool now **is** the primary storage  
[STACKOVERFLOW.COM](https://stackoverflow.com) [DBA.STACKEXCHANGE.COM](https://dba.stackexchange.com) . Also consider using **Memory**-optimized data structures where possible in the code (though InnoDB already uses in-memory b-trees and hash indexes once data is loaded).
- **Transactional Behavior:** InnoDB’s hallmark features (transactions, foreign keys, row-level locking) will still function in memory. However, durability (the “D” of ACID) is sacrificed. You may want to set transaction isolation and flush-related system variables to values that make sense for volatile storage. For instance, `innodb_flush_log_at_trx_commit` can be left at 1 (default) even though log writes do nothing – commits will appear fast due to no I/O. Just be aware that a crash means all data is lost; design your usage of this engine accordingly (e.g., for ephemeral workloads, caches, or testing where persistence isn’t required).
- **Metadata and Restart:** After a restart, MariaDB’s information schema or data dictionary may still have records of tables that were created with your engine, but the actual data is gone. The engine should handle this gracefully – typically, an InnoDB engine encountering a table that’s listed but whose tablespace is missing would try to report an error. You might need to ensure the plugin can **recreate or accept empty tables** on startup. One approach is to have the plugin always initialize a fresh tablespace in memory and ignore any table it cannot find (maybe treating it as empty). Document this behavior for users: they should not expect data to persist, but the table definitions remain (similar to how MEMORY engine tables exist after restart but empty). In practice, some may choose to drop and recreate tables each run.

- **Compatibility:** Since you are effectively forking the InnoDB code, keep track of your changes for forward-porting. When MariaDB releases a new version, you'll need to merge your in-memory modifications into the updated source to build a new plugin. Aim to minimize invasive changes by **using conditional compilation** (e.g., a compile-time flag like `INNODB_MEM_ONLY`) or clearly commenting your changes. This will ease maintenance.
- **Performance Tuning:** You might remove or adjust certain InnoDB settings that no longer apply (file I/O threads, read-ahead, etc.). Simplifying these can reduce overhead. In-memory operation might also let you reduce mutex contention on buffer pool flush lists, etc., since flushing is off. Monitor the engine's CPU usage; you might find new bottlenecks once disk latency is out of the picture.
- **Plugin Integration:** Use MariaDB's plugin system to your advantage. The server treats your DLL as a plugin – you can load and unload it (though unloading a storage engine at runtime is tricky if tables are open). If you encounter startup issues, start `mysqld` with `--skip-innodb` (to not even initialize the built-in one) and then use `INSTALL PLUGIN SQL` to load yours, catching any error messages. This incremental approach can help isolate problems.

In summary, **porting InnoDB to an in-memory engine on Windows** involves building the engine as a standalone module and carefully removing its reliance on filesystem storage. The compilation is done via Visual Studio with CMake (MariaDB supports this out-of-the-box for plugins), and deployment is handled like any other MariaDB plugin. While not a trivial project, it leverages MariaDB's pluggable architecture to achieve a fast, ephemeral storage engine. By following the steps above and paying attention to potential pitfalls (like recovery and metadata handling), you can create a functional in-memory InnoDB plugin for the latest MariaDB on Windows, suitable for use cases where high-performance transient storage is needed.

### Sources:

- MariaDB Knowledge Base – *Specifying Which Plugins to Build: dynamic vs static plugin build options* [RUNEBOOK.DEV](#)
- MariaDB Knowledge Base – *Building MariaDB on Windows: Visual Studio version support and build requirements* [MARIADB.COM](#) [MARIADB.COM](#)
- Stack Overflow – Discussion on pure in-memory InnoDB (necessity of code changes) [STACKOVERFLOW.COM](#)
- DBA StackExchange – Example of loading InnoDB as a plugin on Windows (plugin-dir and config) [DBA.STACKEXCHANGE.COM](#)

show me the source code for the changes

Reasoned about changes for in-memory InnoDB for 4 seconds 

Below is an **illustrative** example of how you might stub out InnoDB's file I/O calls and flush routines in the MariaDB source code to force an **in-memory** only mode.

### Disclaimer

- The actual InnoDB code base is large and spread across multiple files (e.g., `os0file.cc`, `fil0fil.cc`, `buf0flu.cc`, `log0log.cc`, `srv0srv.cc`, `recv0recv.cc`, etc.).
- The snippets below are **simplified** and **incomplete**—they will not give you a fully functional engine if pasted verbatim. Rather, they demonstrate **typical changes** you must make. You will need to adapt them to the **exact MariaDB version** you are using, rename variables/functions as needed, and carefully test them.
- This approach uses conditional compilation (via a custom macro `INNODB_MEM_ONLY`) to preserve normal code paths when building standard InnoDB while disabling disk I/O for your memory-only build.

---

## 1. Adding a Custom Macro

One straightforward way is to define a macro in the CMake build for the InnoDB source. For example, in your `CMakeLists.txt` (or when invoking CMake) do something like:

```
cmake
```

 Copy  Edit

```
# In your top-level or plugin-level CMakeLists.txt # Enable a custom define for  
your memory-only InnoDB build add_definitions(-DINNODB_MEM_ONLY)
```

Alternatively, you could wrap it in an `option` and pass `-DINNODB_MEM_ONLY=ON` at the command line, etc.

---

## 2. Stubbing Out File Operations (`os0file.cc`)

InnoDB's low-level file I/O is handled in `storage/innobase/os/os0file.cc`. The code below shows a **rough** idea of how you might disable file creation/opening and read/write calls:

cpp



Copy



Edit

```

/***** File I/O
primitives. */ #include "os0file.h" #include "my_sys.h" #include "srv0srv.h"
#include "fil0fil.h" #include "os0errno.h" #include "os0thread.h" #include
<string.h> #ifdef INNODB_MEM_ONLY #include <vector> // A simple static buffer to
simulate file data in memory static std::vector<byte> g_innodb_mem_data; static
bool init_innodb_mem_data(size_t size) { try { g_innodb_mem_data.resize(size, 0); }
catch(...) { return false; } return true; } #endif /* INNODB_MEM_ONLY */
/***** Creates or
opens a file. */ os_file_t os_file_create_func( const char* name,
os_file_create_mode_t create_mode, os_file_perm_t create_perm, ulint purpose, ulint
type, bool* success) { #ifndef INNODB_MEM_ONLY // ----- Normal InnoDB code path
----- // This is the original code that opens/creates real files on disk. //
... // (Original InnoDB Logic) // ... #else // ----- Memory-Only Stub -----
// Do NOT create any physical file. Instead, ensure our memory buffer is "ready".
static bool initialized = false; if (!initialized) { // Example: allocate 1GB for
the entire InnoDB "tablespace" in memory // Adjust size as needed size_t
memory_size = 1024ULL * 1024ULL * 1024ULL; if (!init_innodb_mem_data(memory_size))
{ *success = false; return OS_FILE_CLOSED; } initialized = true; } // For now, just
return a dummy handle (non-null). *success = true; // Return something that isn't
OS_FILE_CLOSED, e.g. a cast pointer or dummy struct return (os_file_t)0x1; #endif }
/***** Writes data
to a file. */ bool os_file_write( os_file_t file, const void* buf, ulint offset,
ulint bytes_to_write) { #ifndef INNODB_MEM_ONLY // ----- Normal InnoDB code path
----- // ... #else // ----- Memory-Only Stub ----- // Write into
g_innodb_mem_data at position 'offset'. // In real usage, you'd need to handle
bounds checking, concurrency, etc. if (file == OS_FILE_CLOSED || offset +
bytes_to_write > g_innodb_mem_data.size()) { return false; }
memcpy(&g_innodb_mem_data[offset], buf, bytes_to_write); return true; #endif }
/***** Reads data
from a file. */ bool os_file_read( os_file_t file, void* buf, ulint offset, ulint
bytes_to_read) { #ifndef INNODB_MEM_ONLY // ----- Normal InnoDB code path -----
---- // ... #else // ----- Memory-Only Stub ----- if (file ==
OS_FILE_CLOSED || offset + bytes_to_read > g_innodb_mem_data.size()) { return
false; } memcpy(buf, &g_innodb_mem_data[offset], bytes_to_read); return true;
#endif } /*****
Flushes the writes to disk. */ bool os_file_flush(os_file_t file) { #ifndef
INNODB_MEM_ONLY // ----- Normal InnoDB code path ----- // ... #else // ----
---- Memory-Only Stub ----- // No-op, nothing to do in memory-only mode.
return true; #endif }

```

In the snippet above:

1. `INNODB_MEM_ONLY` gates the disk-based logic.
2. We maintain a **static memory buffer** (`g_innodb_mem_data`), which stands in for what would otherwise be the data file(s).

3. The real InnoDB logic that opens, reads, and writes real files on disk is bypassed.
4. File flush ( `os_file_flush` ) is a no-op, since there's nowhere to flush to.

You'd apply a similar approach in other file/space-management modules like `fil0fil.cc` , especially where `fil_node_t` structures are created, to ensure no actual `.ibd` or `ibdata*` files are created on disk.

---

### 3. Disabling Flushing and Checkpointing ( `buf0flu.cc` )

InnoDB's buffer pool flush logic lives in `storage/innobase/buf/buf0flu.cc` . In normal operation, a background thread flushes dirty pages to disk. In a memory-only engine, **we do not want** to write pages anywhere, so we can stub out or remove that logic:

cpp

 Copy  Edit

```
/****** Buffer flushing
routines. */ #include "buf0flu.h" #include "srv0srv.h" #ifdef INNODB_MEM_ONLY // If
memory-only, skip flush logic #define SKIP_FLUSHING 1 #endif
/****** Flushes a page
from the buffer pool to disk. */ bool buf_flush_page( buf_page_t* bpage) { #ifndef
SKIP_FLUSHING // ----- Normal code that writes page to disk ----- // ...
#else // ----- Memory-Only Stub ----- // Mark page as clean without writing
to disk bpage->flush_type = BUF_FLUSH_NONE; bpage->is_dirty = false; return true;
#endif } /******
Background flush thread. */ void buf_flush_page_cleaner_thread( void* arg) {
#ifndef SKIP_FLUSHING // ----- Normal background flush thread loop ----- //
... #else // ----- Memory-Only Stub ----- // Do nothing or exit while
(!srv_shutdown_state) { os_thread_sleep(1000000UL); // Sleep 1 second, do nothing }
#endif }
```

Here we:

- Use a macro ( `SKIP_FLUSHING` ) to short-circuit flush logic.
- Mark pages as clean in `buf_flush_page()` but **don't** actually write them anywhere.
- Make the flush thread effectively do nothing (or exit gracefully), avoiding overhead.

---

### 4. Disabling Redo Logging and Crash Recovery ( `log0log.cc` , `recv0recv.cc` )

In `storage/innobase/log/log0log.cc`, InnoDB maintains a redo log for crash recovery. If your engine is truly ephemeral, logs become meaningless. You can stub them out:

cpp



Copy



Edit

```
#include "log0log.h" #ifdef INNODB_MEM_ONLY #define NO_REDO_LOG 1 #endif
/***** Writes a Log
record to the redo Log. */ void log_write_up_to( lsn_t lsn, bool flush) { #ifndef
NO_REDO_LOG // ----- Normal log write to disk or Log buffers ----- // ...
#else // ----- Memory-Only Stub ----- // Do nothing ut_ad(lsn >= 0);
return; #endif }
```

And in `storage/innobase/log/log0recv.cc` (`recv0recv.cc` in older versions) you have crash recovery code. Stub it out so it does not attempt to read a redo log or recover data:

cpp



Copy



Edit

```
#ifdef INNODB_MEM_ONLY #define DISABLE_RECOVERY #endif void
recv_recovery_from_checkpoint_start( ulint scan_len) { #ifndef DISABLE_RECOVERY //
Normal InnoDB recovery code, scanning redo logs, etc. // ... #else // Memory-Only
Stub // Skip everything; treat as fresh start return; #endif }
```

As a result, every startup is effectively a “fresh” InnoDB system tablespace in memory. No disk-based logs or data exist to recover from.

---

## 5. Bypassing the Master Thread Disk Activities (`srv0srv.cc`)

The InnoDB “master thread” manages background activities like checkpointing and purge. In memory-only mode, you still need to handle purge (removing outdated row versions for MVCC) if you want transactions to work normally. However, checkpoint writes to disk or log flushes are not necessary. In `storage/innobase/srv/srv0srv.cc`, you might do:

cpp



Copy



Edit

```
#ifdef INNODB_MEM_ONLY #define NO_CHECKPOINT 1 #endif /* The master thread main
function. */ void srv_master_thread( void* arg) { // ... for (;;) { if
(srv_shutdown_state) { break; } // Normal checkpoint logic #ifndef NO_CHECKPOINT
srv_master_do_flush(); // flush dirty pages to disk srv_master_do_checkpoint();
#endif // Purge can remain if you want normal transaction MVCC
srv_master_do_purge(); os_thread_sleep(1000000UL); // Sleep for 1 second } }
```

## 6. Handling Tablespace Creation ( fil0fil.cc )

In `fil0fil.cc`, InnoDB typically expects to create or open `.ibd` or shared `.ibdata` files. With an in-memory approach, you can either:

1. **Stub out** the creation logic, same as in `os0file.cc`.
2. **Store** necessary metadata in memory only.

For a minimal stub, you might do something like:

cpp

 Copy  Edit

```
#ifndef INNODB_MEM_ONLY #define NO_FIL_DISK 1 #endif fil_space_t* fil_space_create(
const char* name, space_id_t id, fil_space_t::Flags flags, fil_type_t type) {
#ifdef NO_FIL_DISK // Normal InnoDB code that registers a disk-based tablespace //
... #else // Return a fil_space_t struct that is used in memory only fil_space_t*
space = fil_space_create_internal(name, id, flags, type); if (space == nullptr) {
return nullptr; } space->is_in_memory = true; // Custom flag if you wish return
space; #endif }
```

And then you must ensure anywhere that tries to physically open files for that tablespace is also stubbed out.

---

## 7. Building and Testing

1. **Rebuild** MariaDB with your changes:
  - Confirm `INNODB_MEM_ONLY` is defined (e.g., via your CMake config).
  - Compile the entire solution; you should get a `ha_innodb.dll` where the file I/O references have changed.
2. **Install** or **copy** the resulting DLL into MariaDB's plugin directory.
3. **Disable** the built-in InnoDB and **load** your custom plugin:

ini

 Copy  Edit

```
[mysqld] ignore_builtin_innodb plugin_load_add = innodb=ha_innodb.dll
```

4. **Start** MariaDB and run `SHOW ENGINES;` to confirm your engine is active as "InnoDB."
5. **Test** by creating a table:

sql

 Copy  Edit

```
CREATE TABLE test_mem (id INT PRIMARY KEY) ENGINE=InnoDB; INSERT INTO test_mem
VALUES (1), (2), (3);
```

Verify that **no** `.ibd` or `ibdata` **files** appear on disk, and that after restart, data is gone (since it's memory-only).

---

## 8. Summarizing the Key Changes

- `os0file.cc` / `fil0fil.cc` : Stub out file creation/open/read/write so data goes to memory rather than disk.
- `buf0flu.cc` / `srv0srv.cc` : Disable buffer pool flushing, checkpointing, and I/O threads.
- `log0log.cc` / `recv0recv.cc` : Stub out redo log writes and crash recovery so logs are never written to disk.
- Possibly handle foreign keys, double-write buffer, or other disk-bound features similarly.

These changes ensure that **no disk I/O** occurs and that all data resides purely in memory.

**Warning:** This is a **highly simplified** illustration. Real InnoDB code has many interdependent modules. You will likely need to chase references to flush, recovery, and file I/O across multiple files. Testing thoroughly is crucial.

---

## Final Thoughts

- A real in-memory InnoDB fork can be **quite involved**. Officially, MariaDB (or MySQL) does **not** offer a simple toggle for "In-Memory InnoDB."
- You must maintain these patches whenever you update to a new MariaDB version.
- This approach **breaks transaction durability**—data is lost on crash or restart.
- If you want a simpler ephemeral engine, consider **the built-in MEMORY engine** (though it lacks certain InnoDB features like row-level locking and foreign key constraints).

Nonetheless, with the above **example code stubs**, you can see the *pattern* of modifications required to achieve a diskless InnoDB plugin DLL.