mcobject®
precision data management

Will the Real IMDS Please Stand Up?

How to tell the difference between real and imitation
in-memory database systems, and why it matters.

McObject LLC

33309 1st Way South
Suite A-208
Federal Way, WA 98003

Phone: 425-888-8505

E-mail: info@mcobject.com

www.mcobject.com

Introduction

Declining RAM cost, emergence of data-hungry real-time systems, and other factors drove the growth of in-memory database systems (IMDSs) over the past decade. This technology offers the features of traditional (file system-based) database management systems (DBMSs)—including transactions, multi-user concurrency control, and high level data definition and querying languages—but with a key difference: in-memory databases store records in main memory, eliminating disk storage and related overhead. This enables IMDSs to offer faster performance as well as a more streamlined design and smaller code size.

In-memory databases have changed the software landscape in several ways. Whole categories of applications that previously could not benefit from database systems are now able to do so. IMDSs' growing popularity has sparked mergers and acquisitions involving the largest technology companies. However, one troubling recent trend, which should matter to anyone considering using the technology, is a flurry of products falsely claiming to be in-memory database systems. This report examines a handful of these imitators and explains why they fail to deliver on IMDSs' promise. The goal is to better educate potential users.

Why In-Memory Database Systems?

Random-access memory (RAM) has steadily declined in price to the point where desktop and back office systems are customarily configured with amounts of RAM ranging up to hundreds of gigabytes. In memory database storage that would have been prohibitively expensive 20, or even 10, years ago is easily within reach today.

8- and 16-bit embedded systems simply do not have the ability to address sufficient memory to be able to take advantage of an in-memory database. Data management code for these systems is either extremely simple or consists of pushing the data upstream to systems with greater processing power. However, as embedded systems have evolved to 32- and 64-bit processors, more data is processed (stored, sorted, indexed, etc.) at the point of collection. Embedded systems typically require an in-memory database for these tasks, for reasons that include real-time performance demands, harsh operating environments that rule out use of a mechanical hard disk drive, and minimal power resources that preclude motorized "spinning memory" (disk storage).

A wide variety of embedded and real-time applications can benefit from database management features such as concurrent access, fast and flexible indexing, and transactions. But the nature of data in these systems is often transient, making conventional file system-based[1] (i.e. disk-based) database systems an inappropriate choice. A few examples include network routers (which have no spinning memory), set-top box electronic program guides (if lost due to power interruption, the programming guide is simply downloaded from the cable headend or satellite transponder), and cached social network graphs (think LinkedIn, Tagged, etc).

Finally, many embedded systems and real-time enterprise applications (including business intelligence, Web caching and others) impose performance demands that can only be met with an in-memory database system.   A battlefield command and control system, sensor fusion application or other critical real-time system simply can't wait while data is retrieved from disk, and would suffer from disk-based DBMSs' inherent overhead caused by functions including cache management and data transfer.

Innovation and Imitation

Companies and products such as Polyhedra, TimesTen, and *eXtreme*DB® represented the first wave of IMDSs. Their growth occurred both in many application types that clearly could not accommodate conventional DBMSs and in some systems where use of disk-based databases would likely have been used. This drove traditional embedded and enterprise software vendors, including IBM and Oracle, to acquire several of the early IMDS companies.

As might be expected, the success of in-memory database systems motivated many DBMS vendors to try to protect their slice of the database system market. As a result, numerous vendors now market their variations on traditional DBMS software as "in memory." Wikipedia's in-memory database article[2] documents the proliferating claims. In July 2007, the page listed just eight commercial and open source IMDSs (or products claiming to be such). By February 2010, the list had mushroomed to 24 entries. While Wikipedia may not be an authoritative source on market size and growth, this certainly documents that *claiming* to be an IMDS has been a hot technology trend in recent years!

The problem is, while many of these products exploit cheap and abundant memory, they are not in-memory database systems. Understanding the distinction between these knockoffs and actual IMDSs is critical for potential users whose problem domain is best served by the technology. The differences can affect the hardware requirements (and therefore total cost of ownership), performance, time-to-revenue, and ultimately the success or failure of a solution. In order to tell the difference between real and fake IMDSs, two key areas to examine are *origins* and *wholeness*.

[1] Throughout this paper we will use the terms "file system-based", "on-disk" and "disk-based" interchangeably to refer to traditional DBMSs that are "hard-wired" to store records to a file system on persistent media. While this media is usually a hard disk, it can also be a flash memory stick, solid state drive (SSD) or some other storage.

[2] http://en.wikipedia.org/wiki/In_memory_database

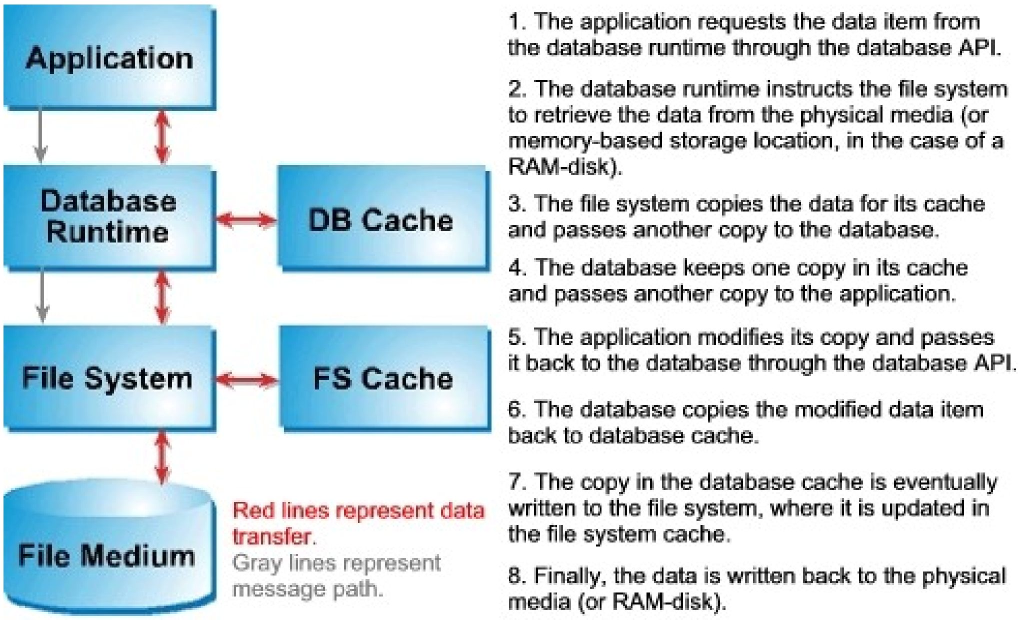
Origins

First, a true IMDS is one that was written, from the start and from the bottom up, with the intent of being an in-memory database system. The design goals and optimization strategies used to develop IMDSs are diametrically opposed to those underpinning traditional, file system-based DBMSs. Traditional databases are created using a strategy of *minimizing file I/O*, even at the expense of consuming more CPU cycles and memory. This design strategy stems from the vendors' recognition of I/O as the single biggest threat to their databases' performance. In contrast, in-memory database systems eliminate disk I/O from the beginning. Their overriding optimization goal is reducing memory and CPU demands. Once these opposing strategies—minimizing I/O vs. minimizing memory and CPU demands—are "baked into" database system code, they can't be undone, short of a rewrite.

Note that having minimized memory and CPU demands from the start, the in-memory database system provider always has the option, later on, to "use up" some of those spare CPU cycles and bytes of memory if the design goals change—for example, to create a hybrid database system that permits the user to selectively add on-disk storage back into an application. Conversely, the creators of on-disk (traditional) DBMSs cannot "un-use" the CPU cycles or memory that are consumed to achieve the fundamental design goal of minimizing I/O in their products.

When a DBMS designed for disk storage is recast as an IMDS—often by simply redeploying the original database system with memory-based analogs of a file system—artifacts of its origins remain. These can inhibit performance and waste system resources. For example, traditional DBMSs store redundant data (that is, data that is already stored in tables) in their indexes. This is useful for on-disk databases: if sought-after data resides in the index, there is no need to retrieve it from the data file, and I/O is prevented. But when the vendor later deploys this database in RAM and re-christens it as an IMDS, the redundant data is typically still present in the indexes, consuming storage space even though the entire table is now in memory. There is no longer any performance advantage from the redundant data—it just wastes memory. In contrast, a database designed from the ground up as an IMDS does not store redundant data in its indexes.

Maintaining a cache is another artifact. Traditional databases keep recently-used records in cache, so they can be accessed without I/O. But managing the cache is itself a process that requires substantial memory and CPU cycles, so even a "cache hit" underperforms an in-memory database. This is significant given that some vendors simply add a feature that causes 100% of a database to be cached, to justify slapping an IMDS label on their product.



Red lines represent data transfer.
Gray lines represent message path.

1. The application requests the data item from the database runtime through the database API.

2. The database runtime instructs the file system to retrieve the data from the physical media (or memory-based storage location, in the case of a RAM-disk).

3. The file system copies the data for its cache and passes another copy to the database.

4. The database keeps one copy in its cache and passes another copy to the application.

5. The application modifies its copy and passes it back to the database through the database API.

6. The database copies the modified data item back to database cache.

7. The copy in the database cache is eventually written to the file system, where it is updated in the file system cache.

8. Finally, the data is written back to the physical media (or RAM-disk).

Figure 1.

Yet another artifact is on-disk database system architectures' requirement that data be transferred numerous times as it is used. Figure 1 shows the handoffs required for an application to read a piece of data from an on-disk database, modify it and write that record back to the database. These steps, which require time and CPU cycles, are still present in nearly all the DBMSs that have re-christened themselves as IMDSs. In contrast, an IMDS has just a single data transfer (in each direction). Data is copied directly from the IMDS to the application, and back from the application to the database, as shown in Figure 2. There are no intermediate copies in a database cache or file system cache.

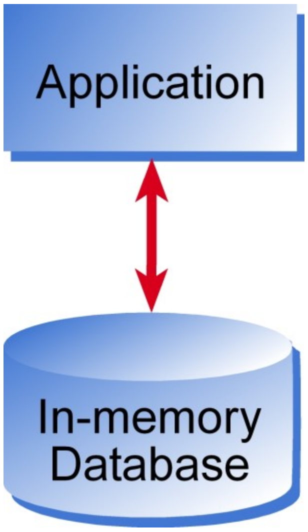


Figure 2.

Do such artifacts really hinder performance? One benchmark test[3] addressed the question, measuring performance of an application using an embedded IMDS alongside the same application with an embedded disk-based DBMS. The application with the traditional database was benchmarked in normal mode, and also when deployed on a RAM-disk. The latter scenario is analogous to many of today's imitation IMDSs: they merely eliminate physical disk I/O while retaining caching, data transfer and other traits.

In the benchmark, moving the on-disk database to a RAM drive quadrupled read performance, and tripled database write (update) performance. But the same application running on a true in-memory database system delivered much more dramatic performance gains: the IMDS outperformed the RAM-disk database by 4x for database reads and by a startling 420x for database writes.

Wholeness

An IMDS is a type of database management system, and it should be a *complete* DBMS. It should not sacrifice functionality, or impose superfluous functionality, just to accommodate in-memory storage of tables or of the entire DBMS. Sacrifices like inability to support concurrent access, or non-working remnants of functions involving disk-access (such as useless transaction logging files in some of the products discussed below), are giveaways that an old-style DBMS is being shoehorned into the role of an IMDS.

With that backdrop, let's examine a few specific database systems.

Oracle's BerkeleyDB

Berkeley DB is probably the best known of the class of DBMS products that is sometimes mistakenly classified as an in-memory database system. A quick examination of the product's in-memory offering reveals that, in addition to some pretty serious limitations, it really just substitutes RAM for file storage. All the other assumptions about how to build and maintain the database are unchanged (such as cache, indexes, and so on).

The following passages from the BerkeleyDB documentation illustrate the point:

- *"The DB cache must be configured large enough to hold all your data in memory. If you do not size your cache large enough, then DB will attempt to write pages to disk. In a disk-less system, this will result in an abnormal termination of your program."*

[3] See "Examining Main Memory Databases", *iApplianceWeb*, January 4, 2002, available at http://www.iapplianceweb.com/story/OEG20020104S0070.htm.

This indicates that the in-memory capability of BerkeleyDB really just means creating a cache large enough to hold 100% of the database. And since this "IMDS" capability amounts to large scale caching, all of the overheads associated with caching in traditional DBMSs are still present. There is no point having a cache for an in-memory database – why cache what is already in memory?

- *"…logs are still required if you want transactional benefits … (such as isolation and atomicity)… you must enable logs but configure them to reside only within memory."*

On-disk database systems invariably use transaction log files that require disk writes (and their associated I/O), in order to provide the ability to roll back an aborted transaction, or perform roll forward recovery after a crash. With BerkeleyDB, the log file is apparently inextricably tied into the transaction properties of isolation and atomicity, as well. In contrast, IMDSs are designed to provide logless transactions. A transaction log may be presented as an option, often with tuning parameters that allow the user to trade off durability and performance. The entire notion of BerkeleyDB's mandatory "transaction log," as presented above, is a by-product of how an on-disk database operates. Since its transaction log now "reside[s] only within memory," it can no longer serve the purpose of providing recovery. The log itself is a useless appendage consuming valuable memory, and the process of writing to it is a waste of CPU cycles.

- *"…your in-memory only application must be a single-process, although it can be multi-threaded."*

This passage reveals an onerous restriction on the in-memory BerkeleyDB technology: the inability to support more than one process. There is no reason an in-memory database should not be able to support multiple processes simply by placing the database into shared memory. BerkeleyDB lacks this ability to use shared memory.

- *"…make sure the in-memory log buffer is large enough that no transaction will ever span the entire buffer…"*

The content above from the BerkeleyDB documentation further highlights the fact that support for true IMDS capabilities is incomplete. Why should the technology limit the maximum size of a transaction? After all, when the transaction buffer is a disk file, there is no such limit. And there is no inherent reason why the maximum size of a transaction in an in-memory database should be predefined. But since BerkeleyDB's implementation of an in-memory database amounts to in-memory analogs for file-based artifacts like transaction logs and buffers, limitations like this have to be imposed.

MySQL

MySQL, an open source DBMS, offers something called “memory tables.” This feature promotes a way to keep the database in memory, in order to boost performance. Unfortunately, the capability comes with serious limitations. First, memory tables must use fixed length slots and cannot contain BLOB or TEXT columns. (Database files are usually segmented into pages, and each page is further segmented into a number of slots. To accommodate variable length data, a DBMS can either spread a single row across multiple slots or can allow the size of the slots themselves to vary. MySQL allows in-memory table rows to do neither.)

Also, a MySQL memory table’s maximum size is 4 GB and its maximum key length is 500 bytes. The key length restriction indicates that the physical structure of MySQL’s b-tree index nodes is the same in both memory tables and (traditional) file tables. When the key values are *not* stored in the key slot (as in an in-memory database) they will not affect the width of the slot and therefore do not impose artificial limits on the length of a key. In short, this limitation shows that MySQL’s memory tables implementation suffers from the “artifact” of extra copies of data stored within its indexes. As discussed above, this redundant data serves the useful purpose of minimizing I/O in disk-based databases, but simply wastes memory in an in-memory database system.

In addition, space that is freed by deleting rows in a MySQL memory table can only be reused for new rows of that same table. In contrast, when something is deleted from an IMDS, the free space goes back into the general database memory pool and can be reused for any subsequent need, whether it is for a row of a different table, or a page for a tree node, or anything else.

Finally, when using MySQL’s replication, “slave” replica databases do not recognize when a “master” (main) database ceases to exist. If the master node goes down, the memory table is wiped out on that node, but remains present on the slave(s). When the master database comes back up and the replica(s) re-attach to it, all on-disk tables are automatically re-synchronized, but memory table are treated differently: they remain present on the slave nodes only, but are absent in the master database until some explicit action is taken to re-create them there. So, for memory tables, one of the key notions of database replication – that master and slave nodes must be kept synchronized, without explicit action being taken by the application – is broken in this circumstance. In contrast, an IMDS will re-synchronize the database, even in the case of a hybrid (in-memory and on-disk) database.

SQLite

The open source SQLite embedded database also offers memory tables. In its implementation, memory tables cannot be shared by other users and this relegates the memory tables’ usefulness to that of mere user-defined temporary tables.

When SQLite’s memory tables feature is used, all of the tables in a given database must be of that type. As a technique to gain greater concurrency (since SQLite relies on database locking), SQLite allows an application to open and/or attach to more than one database, with transactions that span database boundaries. However, according to the SQLite documentation, if the main database is in-memory, then SQLite no longer enforces the ACID (Atomic, Consistent, Isolated and Durable) properties of transactions. In other words, if the system fails during a transaction that spans database “A” (in-memory) and database “B” (on-disk), SQLite will not guarantee the integrity of the “B” database.

In-memory databases cannot be saved to persistent storage with the SQLite product, though you can find one or more third party patches that will apparently supplement SQLite with this ability. In contrast, while in-memory databases eliminate the disk writes that are mandatory at various points in traditional databases’ processing, IMDSs typically provide features that allow optional, periodic streaming of data to persistent storage.

SQLite’s memory tables feature lacks the design criteria—and the capabilities—of a true in-memory database system. The constraints indicate memory tables’ intended use was more as temporary tables—for example, to support fast lookup of coded values—before some promoters of SQLite decided to use this feature to attempt to jump on the IMDS bandwagon.

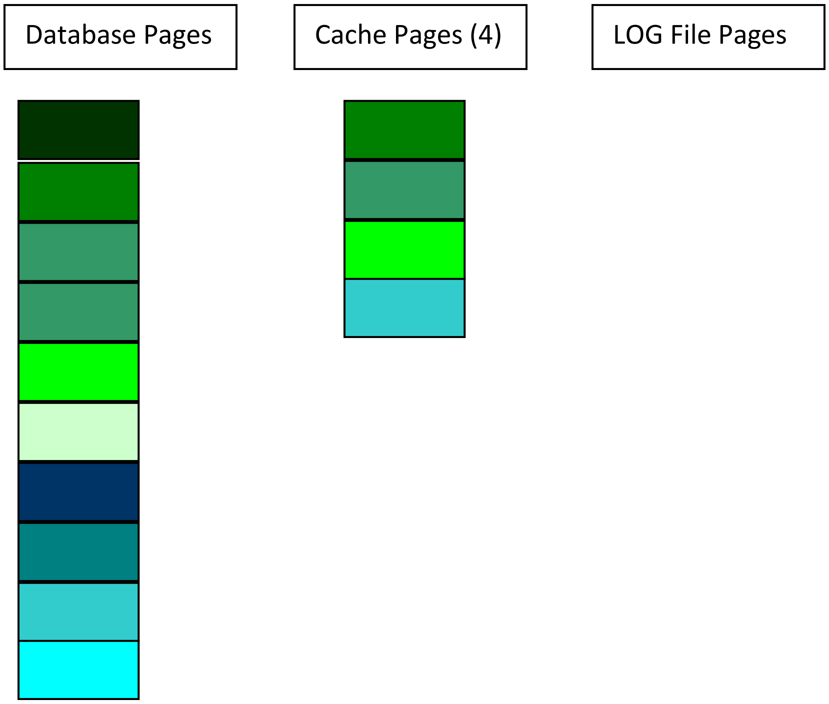
In some cases, SQlite’s integration of memory tables and file tables is incomplete. In the case described above, where support for ACID-compliant transactions can lapse, this flaw can threaten to compromise data integrity. In contrast, a true in-memory database system (like any DBMS worthy of the name) regards integrity of the entire database as a fundamental covenant between the system and users.

RDM Embedded

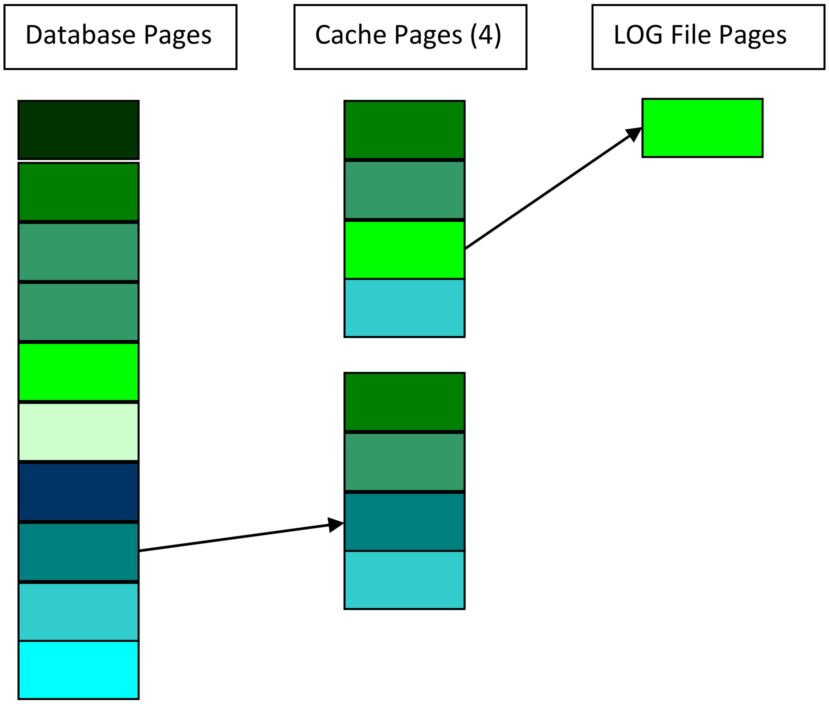
RDM Embedded is perhaps the most recent of the in-memory database knock-offs. As with BerkeleyDB, the RDM Embedded response to the IMDS challenge was simply to replace disk-based files with shared memory and call this an in-memory database. All the inner workings of the file system-based database run-time are unchanged. Indexes require as much space as if they were created on disk. The run-time maintains a transaction activity file (TAF), and spawns transaction log files for each user/process that has opened the database, to support recovery in case one of the processes crashes (even though these log files will now reside entirely in shared memory, making them useless in the event of system failure). This “in-memory database” also still maintains and manages a cache, exercising its lookup and other cache-related logic for every database access—even though the cache now means nothing, since the entire database is in memory.

The existence of a cache is particularly challenging, for reasons inherent in RDM Embedded’s architecture. The product requires a minimum cache size (specified in number of pages) as well as “transaction overflow” pages. Database caches use a least-recently-used (LRU) algorithm to determine what page to remove when the cache is full and a new page needs to be brought in to satisfy a read request. When the cache is extremely small, the likelihood that any given page will be in cache is also quite small, causing the database system to thrash on the cache (constantly flushing pages in order to bring in new pages). This wastes CPU cycles. The application faces a less than ideal situation. It must either permit thrashing (and suffer the performance consequences), or create a larger cache, which is redundant because it is already guaranteed to be in memory.

In RDM Embedded, the transaction log file is used as overflow when a transaction modifies more database pages than can be cached. For example, if 1) the cache is 8 pages, 2) every page has been modified, and 3) the transaction needs to read data that is not in cache, then one of the modified 8 pages has to be removed from the cache to make room for the incoming page. But the modified page cannot just be discarded or overwritten. It has to be kept somewhere so that when the transaction is committed, this page gets written back to the database. In RDM Embedded, this is done by writing the modified page to the transaction log file. When the RDM database is in memory, this will limit the maximum size of a transaction to the number of pages that will fit in the shared memory segment that is created to represent the LOG file. Let’s also look at how this impacts performance:

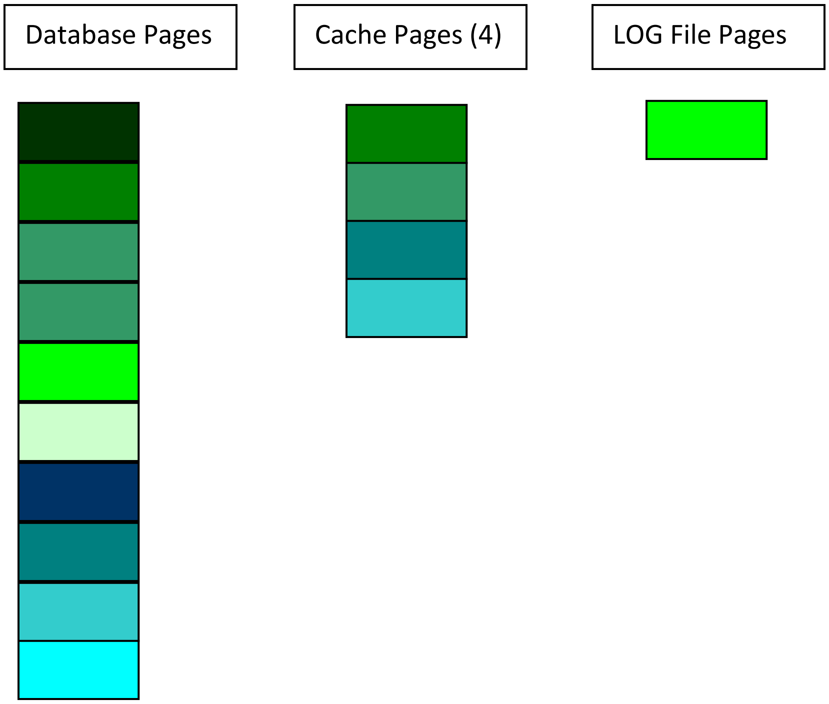


This represents the database, cache, and log files at a point when the cache is full during a write transaction (the application has caused four pages of the cache to be modified by insert/update/delete operations).

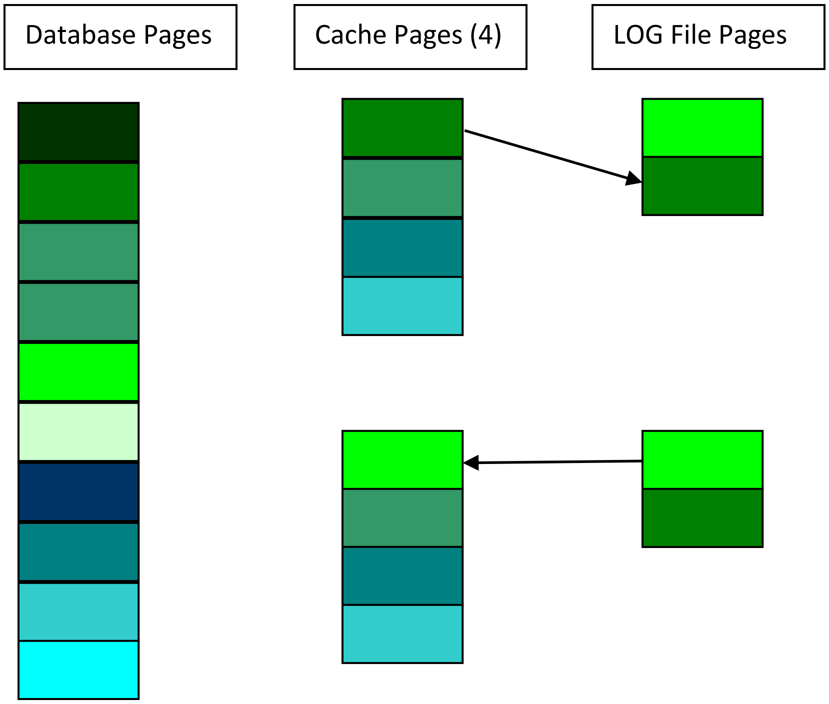


When a fifth page must be read, one of the four modified pages has to be moved from the cache to the log file to make room in the cache for the new page.

(continued on next page)



This represents the state of the database, cache and log file after removing one page from cache and reading another page from the database into the cache.



If the application needs to read a record that exists on the page that was previously flushed, another page has to be moved from cache to the log file, and then the log file page moved back into cache.

As a transaction continues, it’s clear how the small size of the cache exacerbates the need to constantly move pages between the log “file” and cache.

Because RDM Embedded requires different shared memory segments for data files and key files, and there has to be at least one of each, the database will also be unable to use memory freed by, for example, deleting rows of a table, for a purpose other than adding new rows. As stated above in reference to SQLite's memory tables, a true in-memory database should have just a single memory pool and provide flexibility to use memory as needed.

Sound Memory Management

Another key characteristic of true in-memory database systems is their reliance on sound memory management. Specifically, memory must be managed as efficiently as possible—one wasted byte per record means 1 million wasted bytes in a relatively small database of 1 million records. This presents demands on database system code in several ways. First, it affects how objects are laid out in storage. A disk-based DBMS would likely store data as it is defined, e.g.

```
create table XYZ ( char[3] abc; float def; )
```

This would result in a byte of padding between 'abc' and 'def', because floating point numbers must be aligned on an even byte (or even word) boundary. An in-memory database would hopefully be more intelligent about data layout, and rearrange the data so that 'def' is stored first, followed by 'abc', because character strings have no alignment requirement.

Second, the de facto memory manager (usually malloc/free in C or new/delete in C++) used within the source code of the vast majority of DBMSs imposes significant inefficiencies. These are general purpose memory managers, which perform acceptably for most allocation patterns, but do not excel for any one pattern. Substantial overhead is built into them. For example, in malloc/free and new/delete, the heap of memory is organized as a singly-linked list of pointers to free blocks. Each block has a size. So each block of free memory has 8 bytes of overhead (a 4 byte pointer to the next free block, and the 4-byte size of the block itself). As memory becomes more fragmented (more free blocks of increasingly smaller sizes), the proportionate overhead of managing it becomes greater. In addition, the time to walk the chain of pointers to find a free block of the required size can take longer and longer (and eventually fail, if no sufficiently large free block exists due to fragmentation). There are memory allocators that exhibit less overhead and better performance, such as block allocators, bitmap allocators, stack allocators, etc. Given that memory is the most precious resource for in-memory databases, an IMDS should employ these superior memory managers.

Finally, for IMDSs, the heap memory is a shared resource and access to the pointer chain must be serialized. In particular, chaos would ensue if one task in the process of breaking the chain to insert a new link (as when freeing memory) is swapped out by the scheduler, and another task is allowed to walk the (now broken) chain (as when malloc is searching for a free block large enough to satisfy a memory allocation request). A synchronization primitive (e.g. a semaphore) is used to block the second task until the first task has completed its operation and left the chain intact. Thus, access to the chain is serialized. The consequence of serialized access is that systems that rely heavily on dynamic memory management will not scale well on multi-CPU/multi-core systems when using a default, general purpose allocator. Multiple threads on multiple cores should be able to run in parallel, but serialization defeats this. IMDSs can fall into that trap, but “built from scratch” IMDSs can employ a specialized type of memory allocator to solve this problem.

Conclusion

When an in-memory database solution is required, carefully consider available options, and cast a skeptical eye on vendors’ claims. The degree to which a solution eliminates caching, and redundant data in indexes, for example, will directly affect performance and efficient use of memory. Avoid artificial restrictions that are by-products of an incomplete implementation of in-memory tables. Demand that any solution provide all standard database product features (replication, BLOB and TEXT columns, etc.), and refuse to tolerate artificial limits on in-memory database or transaction size. Test different solutions. Results will differ widely between purported “in memory database systems,” because some of these products really are IMDSs, while others are not. The performance gap between them arises directly from issues discussed in this paper.