In-Memory Database Systems: Myths and Facts

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

In the past decade, software vendors have emerged to offer in-memory database system (IMDS) products. IMDSs are described as a new type of database management system (DBMS) that accelerates information storing, retrieving and sorting by holding all records in main memory. IMDSs never go to disk. This eliminates a major source of processing overhead and delivers performance gains of an order of magnitude or more, the vendors say.

But is this idea new? For years, DBMSs have employed caching, which keeps frequently requested records in RAM for fast access. In addition, several traditional database systems offer a feature called memory tables, to hold chosen data sets in memory.

And long before IMDSs came on the scene, DBMSs were occasionally deployed entirely in memory, on RAM-disks; more recently, it's been suggested that using DBMSs on Flash-based solid state drives (SSDs) will deliver breakthrough responsiveness by eliminating physical disk I/O. Clearly, all these techniques leverage memory-based data retrieval. Do IMDSs really add anything unique?

In fact, the distinction between these technologies and true in-memory database systems is significant, and can be critical to the success of real-time software projects. This white paper explains the key differences, seeking to replace IMDS myths with facts about this powerful new technology.

Myth 1: In-Memory Database Performance Can Be Obtained Through Caching

Caching is the process whereby on-disk databases keep frequently-accessed records in memory, for faster access. However, caching only speeds up retrieval of information, or "database reads." Any database write – that is, an update to a record or creation of a new record – must still be written through the cache, to disk. So, the performance benefit only applies to a subset of database tasks.

Caching is also inflexible. While the user has some control over cache size, the data to be stored there is chosen automatically, usually by some variant of most-frequently-used or least-frequently-used algorithms. The user cannot designate certain records as important enough to always be cached. It is typically impossible to cache the entire database.

In addition, managing the cache imposes substantial overhead. To select and then to add or remove a record from cache, the algorithms described above use memory and CPU cycles. When the cache memory buffers fill up, some portion of the data is written to the file system (logical I/O). Each logical I/O requires a time interval, which is usually measured in microseconds. Eventually, the file system buffers also fill up, and data must be written to the hard disk (at which point logical I/O implicitly becomes physical I/O). Physical I/O is usually measured in milliseconds, therefore its performance burden is several orders of magnitude greater than logical I/O.

In-memory databases store data in main memory, keeping all records available for instant access. IMDSs eliminate cache management as well as logical and physical I/O, so that they will always turn in better performance than an on-disk DBMS with caching.

Myth 2: An IMDS Is Essentially a "Traditional" Database That Is Deployed In Memory, Such As On a RAM-Disk

A RAM disk, or RAM drive, is software that enables applications to transparently use memory as if it were a hard disk, usually in order to avoid physical disk access and improve performance. Applications such as telecom call routing have deployed on-disk database systems on RAM disks in order to accomplish real-time tasks.

But DBMSs deployed in this manner are still hard-wired for disk storage. Processes such as caching and file I/O continue to operate, and to drain performance.

Just how much of a drain? In a published benchmark, McObject compared the same application's performance using an embedded on-disk database system, using an embedded in-memory database, and using the embedded on-disk database deployed on a RAM-disk. Moving the on-disk database to a RAM drive resulted in read accesses that were almost 4X faster, and database updates that were more than 3X faster.

Moving this same benchmark test to a true in-memory database system, however, provided much more dramatic performance gains: the in-memory database outperformed the RAM-disk database by 4X for database reads and turned in a startling 420X improvement for database writes. (A report on this benchmark test, titled "Main Memory vs. RAM-Disk Databases: A Linux-based Benchmark", is available from www.mcobject.com/downloads.)

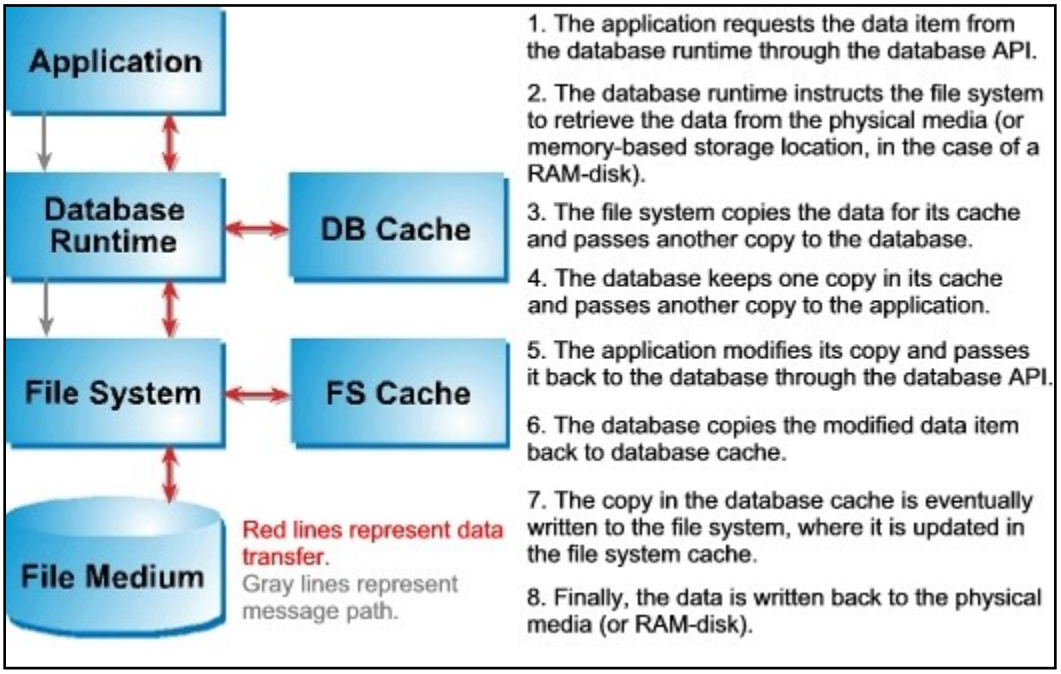

Figure 1. Data transfer in an on-disk database system.

The performance gap is stark, in part because "traditional" (on-disk) DBMSs impose processing overhead beyond the caching and I/O discussed above. For example, data moves around a lot in on-disk databases. Figure 1 shows the handoffs required for an application to read a piece of data

from an on-disk database, modify it and write it back to the database. These steps require time and CPU cycles, and cannot be avoided in a traditional database, even when it runs on a RAM disk. Still more copies and transfers are required if transaction logging is active.

Myth 3: IMDS Performance Can Be Obtained By Deploying a Traditional DBMS on a Solid-State Drive (Flash)

NAND Flash-based solid state drives (SSDs) have made inroads as data storage for Web sites, data centers and even some embedded applications. Because they have no mechanical parts, SSDs can outperform traditional hard disks for data access. This has sparked speculation that using an SSD as storage for an on-disk database system might deliver the performance of an IMDS.

Storage on an SSD eliminates physical disk I/O, resulting in better responsiveness. However, as in the example of deploying a DBMS on a RAM-disk, other drains on performance would remain, even with SSD storage. These overheads include cache processing, logical I/O, data duplication (copying), data transfer, and more. Recall that in the benchmark, moving to RAM-disk storage delivered a tripling of update time, while an IMDS boosted database write performance by 420 times.

And whether used with a database or another application, an SSD is unlikely to challenge memory-based data storage. An SSD has an access time of .2 to .3 milliseconds, or 200 to 300 microseconds. That's a lot faster than a spinning disk, but much slower than 60 nanosecond DRAM.

Myth 4: With the 'Memory Tables' Feature, On-Disk Relational Databases Can Challenge IMDS' Performance

Some DBMSs provide a feature called "memory tables" through which certain tables can be designated for all-in-memory handling. Can memory tables replace true in-memory databases? Not if the goal is to obtain the best performance, smallest footprint, and maximum flexibility.

Memory tables don't change the underlying assumptions of database system design—and the optimization goals of a traditional DBMS are diametrically opposed to those of an IMDS. With an on-disk database, the primary burden on performance is file I/O. Thus its design seeks to reduce I/O, often by trading off memory consumption and CPU cycles to do so. This includes using extra memory for a cache, and CPU cycles to maintain the cache.

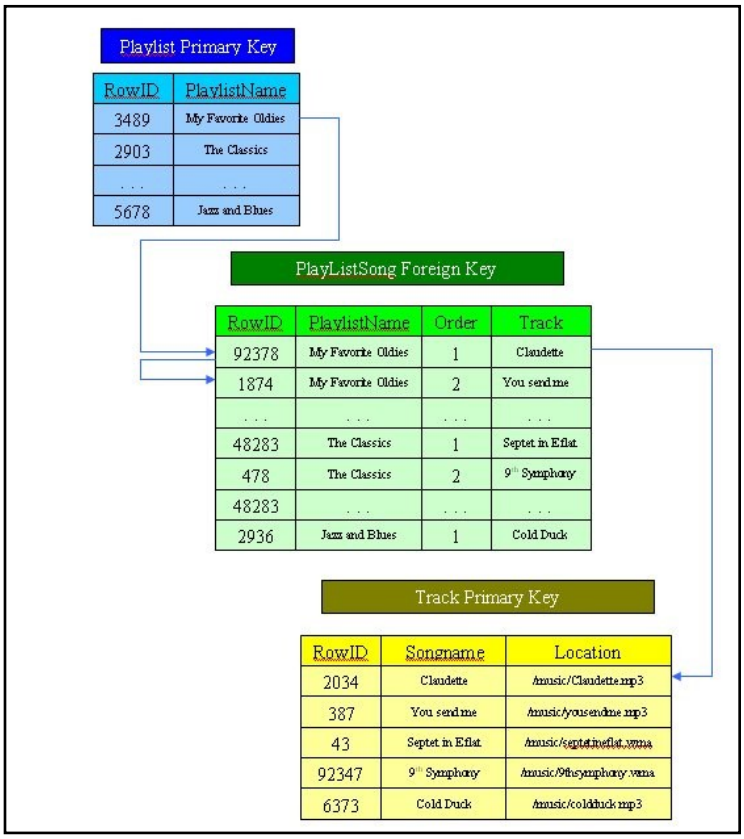

Figure 2. Design of an MP3 player's embedded database, which manages playlists, track names, artists and other meta-data related to songs. If an on-disk DBMS is used, the keys (indexes) will store redundant data.

Another example: much redundant data is stored in the indexes that are used in searching for data in on-disk DBMSs' tables. This is useful in avoiding I/O because if searched-for data resides in the index, there is no need to retrieve it from the file. To reduce I/O, on-disk databases can justify the extra storage space to hold redundant data in indexes. Such tradeoffs are "baked into" on-disk DBMSs. The redundant data in indexes is present, consuming extra storage space even when a table is "in memory" And the tradeoff can no longer be justified (there's no performance advantage to be gained since the I/O has been eliminated by virtue of being in memory, and storage space is at a premium when memory is the storage space.

What's more, a closer look at specific implementations reveals that memory tables have more restrictions than the conventional file tables used in the same DBMS.

For example, in MySQL, memory tables cannot contain BLOB or TEXT columns, and the maximum table size is 4 GB. In addition, space that is freed up 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 in an IMDS, that free space goes back into the general database memory pool and can be reused for any need.

In addition, MySQL memory tables must use fixed length slots and the maximum key length is 500 bytes. That restriction tells you that the physical structure of b-tree index nodes is identical for memory tables and for file (non-memory) tables, because when key values are not stored in the key slot (as in an in-memory database) they don't affect the width of the slot and therefore don't impose artificial limits on the length of a key. This limits how MySQL memory tables can

be used (no key lengths of more than 500 bytes) and also confirms that indexes used with its memory tables will be burdened with redundant data, as described above.

An additional limitation involves MySQL's replication. When a master database comes back after a system failure or shutdown, its memory tables are empty (they were lost when the system went down). The master and replica non-memory tables will be automatically synchronized on restart, but in the absence of some explicit action on the part of the master, the master's and replicas' memory tables will not be synchronized. Therefore the replication/synchronization model is broken with respect to databases that include both types of table. Upon restart, the master process has to explicitly recreate the memory table, or take other explicit action, to reinstate the consistency between master and slave databases.

Memory tables in the SQLite database also impose limitations. For example, they cannot be shared, which really relegates their usefulness to nothing more than user-defined temporary tables. Another restriction is that with SQLite's memory tables, the entire database, not just certain tables, must be in-memory. As a technique to gain greater concurrency (since SQLite's lock granularity is the database), SQLite allows an application to open and attach to more than one database and to have transactions that span database boundaries. If the main database is in-memory, then SQLite no longer enforces the ACID properties of transactions.

In addition, in-memory databases cannot be saved to persistent storage with the SQLite database system, though you can find one or more 3rd party patches that claim to provide this ability.

Myth 5: Database Systems Are Large, Therefore the IMDS Itself Will Have a Large Memory Footprint

Equating "database management system" with "big" is justified, generally speaking. Even some embedded DBMSs are megabytes in code size. This is true largely because traditional on-disk databases – including some that have now been adapted for use in memory, and are pitched as IMDSs—were not written with the goal of minimizing code size (or CPU cycles).

As discussed above, disk I/O is the greatest threat to on-disk databases' performance, therefore these systems' overriding design goal is reducing I/O. When placed in memory, an on-disk database system's "footprint" will be large due to extra memory for a cache, redundant data stored in indexes, and other factors. Designers of on-disk DBMSs have viewed disk space as cheap, so they proceed with the assumption that storage space is virtually limitless.

In stark contrast, an in-memory database system carries no file I/O burden. From the start, its design can be more streamlined, with the optimization goals of reducing memory consumption and CPU cycles. A database system designed from first principles for in-memory use can be much smaller, requiring less than 100K of memory, compared to many 100s of kilobytes up to many megabytes for other database architectures. This reduction in code size results from:

- Elimination of on-disk database capabilities that become redundant and/or irrelevant for in-memory use, such as all processes surrounding caching and file I/O
- Elimination of many features that are unnecessary in the types of application that use in-memory databases. An IP router does not need separate client and server software

modules to manage routing data. And a persistent Web cache doesn't need user access rights or stored procedures

- Hundreds of other development decisions that are guided by the IMDS design philosophy that memory equals storage space, so efficient use of that memory is paramount

Myth 6: An In-Memory Database Is, By Definition, an "Embedded Database"

"Embedded database" refers to a database system that is built into the software program by the application developer, is invisible to the application's end-user and requires little or no ongoing maintenance. Many in-memory databases fit that description, but not all do. In contrast to embedded databases, a "client/server database" refers to a database system that utilizes a separate dedicated software program, called the database server, accessed by client applications via inter-process communication (IPC) or remote procedure call (RPC) interfaces. Some in-memory database systems employ the client/server model while others provide remote interfaces that facilitate access to an in-memory database that resides in another node of the network.

Myth 7: Maximum Practical Size for an IMDS Is Measured in Gigabytes, While On-Disk Databases Can Grow to Terabytes

IMDS technology scales well beyond the terabyte size range. McObject's benchmark report, *In-Memory Database Systems (IMDSs) Beyond the Terabyte Size Boundary* (to download, see http://www.mcobject.com/130/EmbeddedDatabaseWhitePapers.htm) detailed this scalability with a 64-bit in-memory database system deployed on a 160-core SGI Altix 4700 server running SUSE Linux Enterprise Server version 9 from Novell. The database grew to 1.17 terabytes and 15.54 billion rows, with no apparent limits on it scaling further.

Performance remained consistent as the database size grew into the hundreds of gigabytes and exceeded a terabyte, suggesting nearly linear scalability. For a simple SELECT against the fully populated database, the IMDS (McObject's *eXtreme*DB®-64) processed 87.78 million query transactions per second using its native application programming interface (API) and 28.14 million transactions per second using a SQL ODBC API. To put these results in perspective, consider that query performance is usually discussed in terms of transactions per minute.

Myth 8: It Takes an Enormous Amount of Time to Populate an In-Memory Database

"A long time" is relative. For example, using McObject's *eXtreme*DB, a 19 megabyte in-memory database loads in under 6.6 seconds (under 4 seconds if reloading from a previously saved database image). The 1.17 terabyte database described earlier loaded in just over 33 hours.

What is true is that populating a very large in-memory database system can be much faster than populating an on-disk DBMS. During such "data ingest," on-disk database systems use caching to enhance performance. Consider what happens when populating an on-disk database, as the total amount of stored data increases:

First, as the database grows, the tree indexes used to organize data grow deeper, and the average number of steps into the tree, to reach the storage location, expands. Each step imposes a logical

disk I/O. Second, assuming that the cache size stays the same, the percent of the database that is cached is smaller. Therefore, it is more likely that any logical disk I/O is the more-burdensome physical I/O.

Third, as the database grows, it consumes more physical space on the disk platter, and the average time to move the head from position to position is greater. When the head travels further, physical I/O takes longer, further degrading performance.

In contrast, in-memory database ingest performance is roughly linear as database size increases.

Myth 8: Volatility is IMDSs' Achilles Heel – When the System Goes Down, Data is Lost

Data needn't be lost in the event of system failure. Most in-memory database systems offer features for adding persistence.

One important tool is transaction logging, in which periodic snapshots of the in-memory database (called "savepoints") are written to non-volatile media. If the system fails and must be restarted, the database either "rolls back" to the last completed transaction, or "rolls forward" to complete any transaction that was in progress when the system went down (depending on the particular IMDS's implementation of transaction logging).

In-memory database systems can also gain durability by maintaining one or more copies of the database. In this solution – called database replication – fail-over procedures allow the system to continue using a standby database. The "master" and replica databases can be maintained by multiple processes or threads within the same hardware instance. They can also reside on two or more boards in a chassis with a high-speed bus for communication, run on separate computers on a LAN, or exist in other configurations.

Non-volatile RAM or NVRAM provides another means of in-memory database persistence. One type of NVRAM, called battery-RAM, is backed up by a battery so that even if a device is turned off or loses its power source, the memory content—including the database—remains. Newer types of NVRAM, including ferroelectric RAM (FeRAM), magnetoresistive RAM (MRAM) and phase change RAM (PRAM) are designed to maintain information when power is turned off, and offer similar persistence options.

Finally, new hybrid database system technology adds the ability to apply disk-based storage selectively, within the broader context of an in-memory database. For example, with McObject's hybrid *eXtreme*DB Fusion, a notation in the database design or "schema" causes certain record types to be written to disk, while all others are managed entirely in memory. On-disk functions such as cache management are applied only to those records stored on disk, minimizing these activities' performance impact and CPU demands.

Myth 9: An In-Memory Database is Suited for One Computer, While an On-Disk Database Can Be Shared by Many Computers on a Network.

An in-memory database system can be either an "embedded database" or a "client/server" database system. Client/server database systems are inherently multi-user, but embedded in-memory databases can also be shared by multiple threads/processes/users. First, the database can be created in shared memory, with the database system providing a mechanism to control

concurrent access. Also, embedded databases can (and eXtremeDB does) provide a set of interfaces that allow processes that execute on network nodes remote from the database node, to read from and write to the database. Finally, database replication can be exploited to copy the in-memory database to the node(s) where processes are located, so that those processes can query a local database and eliminate network traffic and latency.

Myth 10: The Benefits of an IMDS can be Obtained via STL or Boost Collections, or With Self-Developed Memory-Mapped File(s).

That's the same as stating these alternatives are viable replacements for Oracle, Microsoft SQL Server, DB2, and other on-disk databases. Any database system goes far beyond giving you a set of interfaces to manage collections, lists, etc. This typically includes support for ACID (atomic, consistent, isolated and durable) transactions, multi-user access, a high level data definition language, one or more programming interfaces (including industry-standard SQL), triggers/event notifications, and more. The development and QA resources required to add these features to a self-developed solution is rarely cost-effective, compared to integrating an off-the-shelf IMDS.

Conclusion

In-memory database systems (IMDSs) represent a growing sub-set of database management system (DBMS) software. IMDSs emerged in response to new application goals, system requirements, and operating environments. While an IMDS may not be the chosen solution for every application requiring data management, it should be a strong candidate whenever requirements include low latency and a small database footprint. With the growing demand for efficient and high-capability telecom, networking, aerospace, defense, industrial and other real-time systems, and the burgeoning of “smart” portable devices, in-memory database systems deliver both a higher level of end-user satisfaction and shorter development time. By reducing demand for hardware components, IMDSs cut manufacturing costs. By eliminating caching and other processes with hard-to-predict completion time, IMDSs contribute to the determinism required by safety-critical applications.