IMDSs Bring Off-the-Shelf Database Software to the Electronics Mainstream

Electronic Design
Steve Graves
Wed, 2014-10-22 10:44

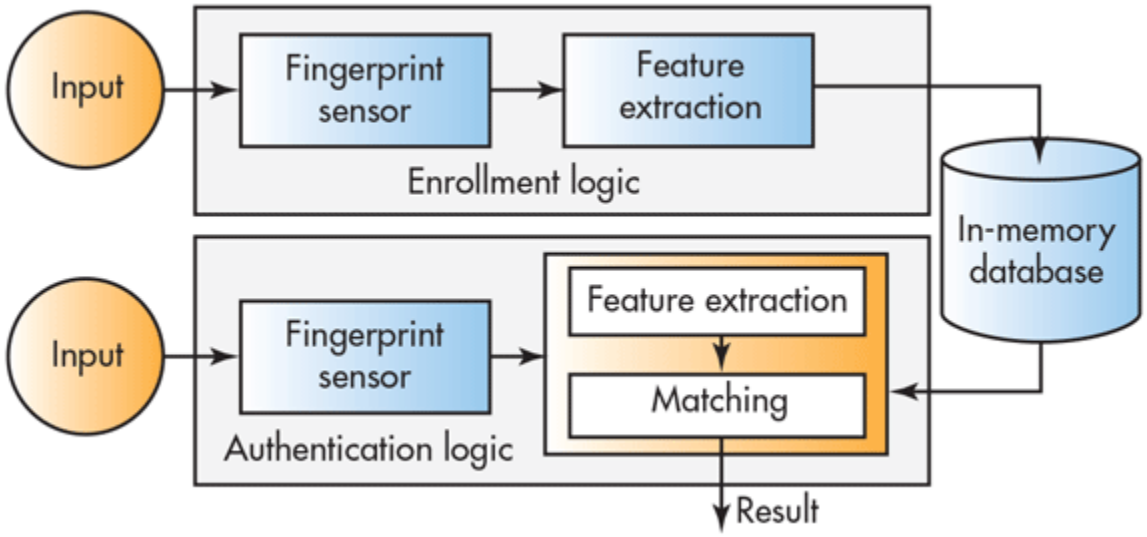
Most of today's mid-career engineers remember when the idea of including a database management system (DBMS) within electronics' embedded software was considered laughable. DBMSs' hardware demands outstripped target devices' RAM, CPU, and storage capacities, and their many "moving parts"—ranging from SQL parsing to transaction management, disk and file system interaction, and caching—imposed unacceptable latency.

Today's electronics operate in a far different environment, often with 32- and 64-bit processors, megabytes (or more) of system memory, and solid-state persistent storage. There is often headroom for a DBMS, which is good, because today's electronics also manage ever-increasing amounts of data, as illustrated by the following examples:

• Consumer electronics devices such as set-top boxes, portable audio players, and even digital cameras need to store, sort, and organize data that facilitates access to multimedia content.

• Industrial-control systems have devolved some automated decision-making to the level of field-based controllers, requiring beefed up on-device data crunching.

• Some biometrics-based access-control systems now store user profiles on fingerprint- and face-scanning devices to eliminate the latency imposed by accessing an external database *(Fig. 1)*.

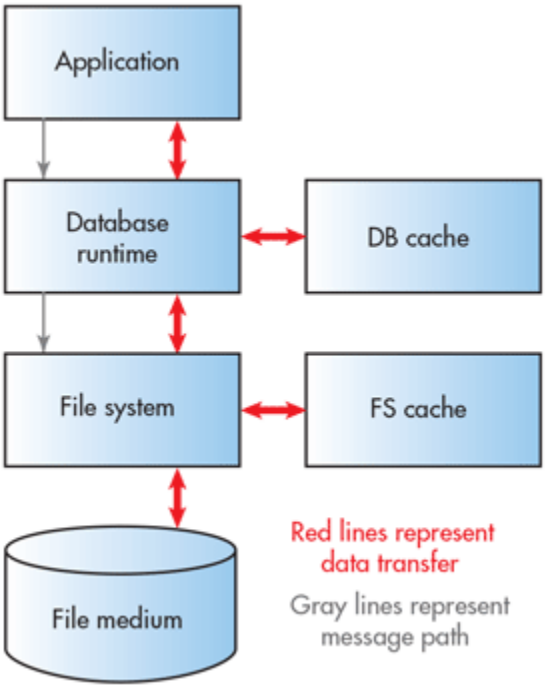


›mmunications equipment faces increased challenges in managing metrics, routes and network topology, l configuration data.

And as the technological environment of electronics has moved toward accommodating DBMSs, database systems themselves have evolved to better meet embedded systems' need for speed, compactness, and reliability. In the examples above, successful designs utilized a special type of DBMS called an in-memory database system (IMDS). In-memory databases store records entirely in main memory, eliminating latency-inducing file I/O.

In-memory storage permits a radically simplified database system design. Traditional (on-disk) DBMSs are built around the assumption that all data will eventually be stored persistently. To mitigate the performance penalty of going to disk, they cache frequently requested items in RAM. While accessing records from cache prevents I/O-related latency, the process of caching imposes its own drag on performance, eating up CPU cycles (and processing time) as it figures out which records to move into and out of cache, and performs these transfers. In contrast, since IMDSs by definition store the entire database in RAM, they eliminate the entire DBMS caching subsystem *(Fig. 2)*.

As a result, IMDSs deliver performance orders of magnitude faster than traditional DBMSs. This can be seen even when an on-disk database system eliminates physical I/O by storing records on a RAM-disk (i.e., a memory-based analog of disk storage).



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 the 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).

McObject's benchmark tests compared database performance on identical work in three database/storage configurations: using an on-disk database system that stored data on a hard drive; using an on-disk database with storage on a RAM-disk; and using an IMDS. Moving the on-disk database to a RAM drive accelerated database reads by nearly four times, and database writes by more than three times, thanks to the elimination of the hard disk's mechanical I/O. However, swapping in an IMDS to perform the same tasks, and thereby cutting out the I/O as well as the cache processing and aforementioned data transfers, accelerated performance by an additional four times (compared to the RAM-disk-deployed DBMS) for database reads, and by 420X for database writes.

s simpler IMDS design brings two additional benefits. First, eliminating big chunks of DBMS processing amely, the I/O, caching and data transfer described above—results in a smaller database code size. Most on-disk DBMSs weigh in at several megabytes or more (sometimes *much* more), while a few streamlined DBMSs may require 500K. However, IMDSs for embedded systems can shrink this size down to 150K or less. This smaller footprint and superior performance may enable use of a less powerful (and less expensive) processor and still attain equivalent performance for an electronics project, thus saving manufacturing costs. Second, a simpler design implies higher reliability, because it requires fewer simultaneous processes and interactions.

Off-the-Shelf vs. Custom DBMSs

For embedded software developers, moving to an in-memory database system typically means leaving behind proprietary data-management code, often based on storage in flat files. Therefore, many of the advantages gained by incorporating an IMDS will result from moving to an off-the-shelf database system (i.e., they are not specific to an in-memory database system) from such a "homegrown" solution. These benefits include:

• *Transactions*, or the ability to group operations into units that succeed (commit) or fail together, preferably with support for the ACID properties (atomic, consistent, isolated, and durable transactions). Transactions are critical for preventing orphaned or out-of-synch records.

• *Multi-user concurrency* to synchronize database access between processes or threads. Often this will take the form of locking, which makes all or parts of the database off-limits to every user except the one writing to it. Database systems can also offer non-locking "optimistic" concurrency. An efficient synchronization mechanism can greatly enhance system stability and performance.

• An extensive *application programming interface (API)*, or library of functions to carry out tasks, such as inserting, querying, and sorting records, and for database administration.

• A *database definition language (DDL)* used to create database schemas (designs) specifying elements like tables, columns, and indexes for a given database. This makes it easy to understand what is stored and its inter-relationships. In contrast, without a schema, users depend on initial developers documenting the source code exceptionally well. Some database systems provide compilers that process schemas to produce code (e.g., C++ header files, API functions) that's needed for a project.

Of course, a "homegrown" database *could* include these features, but creating them would entail months or even years of development and QA. An off-the-shelf database system offers such capabilities ready-to-go, often with an extensive track record of usage that has driven out even obscure potential bugs. Thus, the two major benefits of using a "real" database system—including an IMDS—are time savings and reliability.

Choosing Among IMDSs

Beyond the shared characteristic of in-memory storage, and database system "must haves" like those discussed earlier, IMDSs exhibit significant technical diversity. If an in-memory database seems to be a good conceptual fit for a project, additional research is needed to identify the right IMDS product. The majority of IMDSs target enterprise, business, and other non-embedded uses. Criteria for choosing the right IMDS for electronics include:

IMDS architecture

Some IMDSs are based on client/server architecture, in which client applications send requests to a database server. When residing on the same computer, these client and server processes mostly interact via inter-process communication (IPC) messaging. Otherwise, they interact through remote procedure calls (RPCs). Other IMDSs

plement an in-process (aka embedded) architecture in which the database system runs entirely within the application process. Instead of providing separate client and server modules, in-process IMDSs consist of object code libraries that are linked in (compiled with) the application.

Related

Interview: Jeff Chang And Steve Graves Cover In-Memory Databases

Embedded Database Management System Targets iOS Developers

Raima Boosts Database Manager Proficiency

Client/server is a mainstay of enterprise DBMSs. Its advantages include the ability to right-size a network by installing the server software on a more powerful computer (clients need not be upsized). As a result, client/server can lend itself naturally to supporting larger numbers of users. In-process architecture is simpler, which means it has a smaller code size, more frugal hardware resource usage, and a lower probability of defects. In-process design eliminates IPC, resulting in lower latency. In-process IMDSs further accelerate performance and reduce complexity by eliminating server tasks such as managing sessions and connections, and allocating and de-allocating resources.

Application programming interfaces (APIs)

SQL is used widely in finance and other enterprise sectors. Benefits include succinctness in implementing some complex queries. On the negative side, SQL relies on a query optimizer to weigh potential execution plans and choose the best one. This optimizer amounts to a “black box” for developers seeking hands-on control over speed and predictability. The risk of a SQL query bogging down while the optimizer considers execution strategies can make it a poor fit for minimizing latency. Furthermore, the risk of the optimizer selecting different plans from one execution to another may undermine efforts to achieve a deterministic (predictable) response, which is needed in electronics such as avionics and communications equipment.

As an alternative, some IMDSs provide a navigational API that’s closely integrated with programming languages like C/C++. Using this type of API, a run-time application navigates through the database one record at a time. The application code defines the execution plan, and latency is known and predictable. Software relying on such an API will be faster, due to bypassing the parsing, optimization, and execution of dynamic SQL. Information is stored as native-language data types (including structures, vectors and arrays), eliminating the overhead of conversion to SQL data types.

A major perceived advantage of SQL for enterprise/business use is familiarity—most programmers in this sector have some exposure to it. In contrast, C/C++ is the lingua franca of embedded software, and developers tend to be more familiar (and more productive) with the tools and programming style of a navigational C/C++ API. If SQL is a requirement, latency is generally lower with a rules-based rather than a cost-based optimizer. Also, some IMDSs support use of both SQL and native APIs, even in the same application, where the navigational interface can be “plugged in” for performance-critical operations.

Recoverability mechanisms

Data stored in RAM is volatile, but IMDSs can provide features to ameliorate risk in the event that someone pulls the plug. Potential safeguards include transaction logging, database replication and use of non-volatile RAM:

• *Transaction logging:* With this technique, the database system records transactions in a log file, which can be played back to restore a database to its pre-failure state. One objection to transaction logging with an

nemory database is that it re-introduces persistent writes. In fact, transaction logging does carry a formance penalty. However, benchmark tests show that an IMDS with transaction logging still significantly outperforms an on-disk DBMS for key database operations. For example, an IMDS that keeps its log file on hard disk outperforms a DBMS with a hard disk by 3.2 times for database inserts. This advantage grows to more than 20 times when the IMDS transaction log is written to memory-tier NAND flash storage, such as Fusion-io's ioDrive2.

• *Replication:* In-memory database systems also tame volatility through replication. One or more standby databases on independent nodes are synchronized with the "master" database, and can take over the master's role if needed. Such failover can occur almost instantly. For this reason, replication is used to achieve the database high availability (HA) needed in systems like telecommunications switches that promise "five nines" (99.999%) uptime. In contrast, transaction logging is a tool for achieving data durability, i.e., preventing loss of any committed transactions. While database recovery using the transaction log is typically automated, it often comes into play following an unexpected system shutdown, which implies a cold re-start (e.g. re-booting), a minutes-long process.

• *Non-volatile RAM support:* Battery-backed RAM, which enables data held on a DRAM chip to survive a system power loss, has been available for decades. Consequently, some IMDSs (including McObject's *eXtreme*DB) have added "hooks" to enable database recovery using this form of non-volatile memory. Battery-backed RAM has not caught on widely, though, due to restrictive temperature requirements and the risk of leaking corrosive and toxic fluids.

A new type of non-volatile memory—non-volatile dual in-line memory modules (NVDIMMs)—holds greater promise as a storage medium for IMDSs. NVDIMMs get rid of the battery and combine standard DRAM with NAND flash and an ultracapacitor power source. In the event of power loss, the ultracapacitor provides a burst of electricity used to write main memory contents to the NAND flash chip, where it can be held virtually indefinitely. Upon system recovery, the NVDIMM transfers data from NAND flash back into DRAM, restoring the in-memory database state prior to the power loss event. Another key feature of NVDIMMs is that they plug into existing DIMM sockets, simplifying integration into off-the-shelf platforms.

In McObject's tests (details available here), the features that make the *eXtreme*DB in-memory database system compatible with battery-backed RAM also enabled seamless recovery from failure when the IMDS used NVDIMMs as main memory storage. The benchmarks also showed that in-memory database speed with NVDIMMs (AGIGARAM from AgigA Tech, a Cypress Semiconductor subsidiary) matched performance using conventional DRAM. Like transaction logging, NVDIMMs provide recovery following a system shutdown, and therefore constitute a data-durability, rather than high-availability, solution.

Optional On-Disk Storage

Features of some IMDSs make it possible to selectively use persistent (disk or flash) storage alongside default in-memory. Reasons for implementing disk-based storage include performance optimization (faster in-memory storage can be used for record types that don't require persistence); cost, because a megabyte of persistent storage can be less expensive than the same amount of DRAM; and form factor (disk or flash can be smaller than DRAM). Such a "hybrid" in-memory database system is ideal in electronics product lines that include both disk-less and disk-enabled models. This permits more code reuse, which in turn reduces programming costs. It also eliminates the need to license, and learn, two separate database systems.

However, users should be wary of traditional on-disk DBMSs posing as hybrid IMDSs. With in-memory databases generating substantial buzz in recent years, many on-disk database vendors have jumped on the bandwagon. Often their claims to "IMDS-ness" are based on memory-based product extensions that fail to

minate latency-inducing characteristics, such as replacing disk-based files with shared memory (which is akin providing a built-in RAM-disk for record storage). When adding RAM-based storage, some on-disk DBMS vendors eliminate important capabilities such as multi-user synchronization and support for larger database sizes.

The key to an IMDS's performance lies in its origins as an IMDS. Creation of traditional databases involves 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 to reduce memory and CPU demands. Once the traditional database systems' strategies—minimizing I/O—are "baked into" database system code, they can't be undone, short of a rewrite.

Note that by having minimized memory and CPU demands from the start, the in-memory database system provider always has the option to later "use up" some of those spare CPU cycles and bytes of memory if design goals change. One such example is to support the selective use of an application's on-disk storage. Conversely, the creators of on-disk (traditional) DBMSs cannot "un-use" the CPU cycles or memory consumed to achieve the fundamental design goal of reducing disk I/O.

Conclusion

Companies developing electronics typically have a three-fold mission: offer exciting new products and/or features; get them to market before competitors; and stay ahead of competitors by chipping away at price. For many products, the software required for great new features has crossed a threshold, with data growing sufficiently complex to require a database system.

An in-memory database system is a natural fit due to its small footprint—devices' hardware resources have increased, but a desktop- or enterprise-class DBMS is still likely too bloated. Main memory data storage helps deliver needed performance to satisfy users who expect lower latency and higher-response time consistency from electronics than they do from PC or Web applications.

Quicker time-to-market results from the decision to use an off-the-shelf database system. Time and again, those who've made the switch from a "homegrown" to a "real" DBMS point to savings of programmer-months or even programmer-years in initial development, plus significantly faster upgrades and easier code reuse thanks to off-the-shelf database systems' natural separation of data-management logic from application logic (i.e., prevention of "spaghetti code").

Are there economic advantages to using an IMDS? First, developers cost money, so slashing programmer-months, as mentioned above, also saves money. In-memory databases also outperform their on-disk counterparts economically by reducing per-device hardware requirements. An IMDS can be smaller in terms of code size—kilobytes instead of megabytes—which whittles down storage needs, and its simpler design enables use of a less powerful (and cheaper) processor without sacrificing performance. These savings can drop to the bottom line, as profit. However, electronics developers often pass the lower production costs on to end users, as another tactic to bolster product competitiveness.

Steve Graves co-founded McObject in 2001. Previous to that, he was president and chairman of Centura Solutions Corp., and vice president of worldwide consulting for Centura Software Corp. (NASDAQ: CNTR). He also served as president and chief operating officer of Raima Corp.

Source URL: http://electronicdesign.com/embedded/imdss-bring-shelf-database-software-electronics-mainstream