

COTS databases for embedded systems

By Steve Graves

Many embedded computer systems are excellent candidates for embedded databases. Steve defines what an embedded database is and what features developers can expect to be or not to be in them, and discusses which of those features, whether present or not, make embedded databases appropriate for embedded systems.

Databases, large and small

Certain types of software are small footprint almost by definition. For example, boot loaders and device drivers aren't expected to hog memory and CPU resources, so embedded systems developers generally don't think twice about incorporating such code into their projects. On the other hand, certain software virtually screams, "BIG!" and is rarely used in resource-constrained systems.

Database Management Systems (DBMSs) once fit squarely in the second category. Certainly *enterprise databases* demand far more memory and processing cycles than typical embedded systems provide. Even most *embedded databases*, a product category developed about 20 years ago for embedding in server and desktop software, are simply too big, slow, and complex to build into, say, a set-top box, cell phone, or onboard automotive system.

But as embedded systems have evolved to include devices with faster (32-bit) CPUs, more available memory, and sophisticated Real-Time Operating Systems (RTOSs), COTS database systems are finding their way into devices where they would never have been expected. For example:

- Major European mobile phone handset manufacturers rely on a COTS DBMS for address book, user notification, and message retrieval

- Japanese consumer products giant JVC embeds a database in MP3 player technology for playlists and other data organization features
- In the United States, an industry-leading provider of emergency communications gear integrates a DBMS into its product family of wireless infrastructure devices

Today, a COTS database system can be crucial to a product's success. To support expanding features, intelligent devices must manage growing volumes of more complex data. Writing data storage and manipulation code from scratch – the "homegrown" solution – simply takes too long, especially when factoring in quality assurance, portability, and scalability issues. Conversely, an off-the-shelf database system saves development time and, by separating application logic from data management logic, simplifies code reuse and maintenance. Given the competitive need to roll out and update products quickly, a COTS DBMS provides distinct advantages over homegrown code.

When investigating DBMSs, developers quickly discover that a dozen or so vendors offer embedded databases. This creates confusion because only a few embedded database systems are practical for use in *embedded systems* such as cell phones, industrial automation, and the like. Not every embedded database provides the performance, efficiency, and small footprint needed for embedded systems.

Which database management technology fits?

For database management systems, devices represent a new development domain with unique needs. Requirements include small footprint because manufacturers seek to reduce devices' memory, CPU, and storage requirements for economic reasons. Another important need is real-time performance. Communications equipment, consumer devices, aerospace and defense systems, and other categories of embedded software are expected to respond instantly with consequences ranging from annoying to dire if this fails to happen. Therefore, data management for embedded systems must be inherently fast and efficient.

A database for embedded systems must be reliable. While a software crash might be tolerated once in a while in desktop or server software, it's not accepted at all in a cell phone or set-top box. The user will simply demand to return the product. For mission- or life-critical embedded applications (for example, medical, aerospace, and some communications systems) a database will ideally include built-in high availability or a method for establishing and updating multiple copies of a database with automatic failover in case of disruption on the primary node.

Embedded systems rarely manage data that fits into neat, tabular structures. Instead, they must address complex data such as trees and arbitrarily long arrays of simple or complex fields. A DBMS for embedded systems must provide the tools to work efficiently with such data types. Another reason an embedded systems database needs such tools is because the developer usually predefines on-device data access. Tight integration between data management code and the application creates greater runtime efficiency, reducing the need for CPU cycles and other computing resources. Therefore, sophisticated development tools also contribute to a smaller footprint.

Sorting through database software features

Given that embedded database software is diverse, what aspects of the technology best meet the previously described requirements? By considering a few of the key features embedded databases offer, some answers – or at least a clearer picture of the trade-offs – begin to emerge.

Storage modality

On-disk storage

The slowest function of most DBMSs is disk access, the mechanical process by which working copies of information are transferred to persistent media. Traditional on-disk databases are designed to cache frequently requested data in memory for faster access, but also to write all database updates, insertions, and deletes through the cache to be stored to disk. Most DBMSs are on-disk. The advantage of this storage approach is permanence. An in-memory data store, after all, can disappear if its hardware or software environment fails as in a power outage, for example.

In-memory storage

A newer approach is the In-Memory Database System (IMDS), which eliminates disk access and stores data in main memory, sending data to the hard disk only when specified by the application. This all-in-memory storage means IMDSs are very fast, and that speed advantage has made them popular for real-time embedded systems. While elimination of mechanical disk I/O is the main reason for in-memory databases' speed, their streamlined design usually also removes the internal handing off of multiple data copies as well as logical processes, especially those related to caching, that are no longer needed. This simpler design also gives IMDSs a smaller code footprint, which can be crucial for embedded systems such as consumer electronics devices in which system resources are strictly limited.

It is true that system failure can disrupt data stored in memory. But IMDSs compensate for this risk with features such as maintaining the database in nonvolatile RAM, backing up to secondary data storage (disk or flash), high availability schemes that manage redundant database copies, and transaction logging, in which changes (data transactions) are recorded in a transaction log file to enable recovery if needed. IMDSs include Polyhedra from Enea and McObject's *eXtremeDB*.

API

SQL API

The SQL API is the universal enterprise database interface, and is also used in some embedded databases. This familiarity may shorten embedded system developers' learning curve if they've had prior experience with enterprise database systems. In addition, with its ability to express complex queries relatively succinctly, SQL can make a claim to greater coding efficiency.

But is SQL applicable to embedded systems? The answer may hinge on how complex the system's queries are and how deterministic or predictable a response is required. SQL's simplicity also makes it a "black box," inside of which developers exercise little control over performance or resource consumption. Simply put, with SQL statements it is difficult to see at the level of program execution what the DBMS is doing. This increases the chance that some change in software will cause the SQL optimizer, the software responsible for deciding an SQL statement's execution plan, to choose an inferior execution plan. Unless queries are so complicated that SQL's expressiveness lends an advantage, the SQL API's overhead and inherent unpredictability may not be justifiable.

Navigational API

Many embedded databases provide a navigational database API closely integrated with programming languages like C and C++. Embedded systems programmers tend to have advanced knowledge of these languages, and navigational APIs leverage these skills. In contrast to the high-level SQL, navigational APIs work on one record at a time, literally

navigating through the database, though much of the traversal occurs programmatically via looping and other structured programming techniques. Compared to SQL, navigational APIs are deterministic – it is known exactly how the data will be traversed when the application is compiled – and are faster, executing at the speed of compiled C/C++ code while avoiding the parsing, optimization, and execution dynamic SQL requires.

Embedded database systems featuring SQL include Pervasive Software's PSQL, McObject's *eXtremeSQL*, and Solid Technology's EmbeddedEngine.

DBMSs with navigational APIs include McObject's *eXtremeDB*, BerkeleyDB from Oracle, and Birdstep Technology's RDM Embedded. Database systems with both SQL and proprietary navigational programming interfaces include McObject's *eXtremeDB*, Birdstep Technology's RDM Server, and Faircom's c-treeSQL Server.

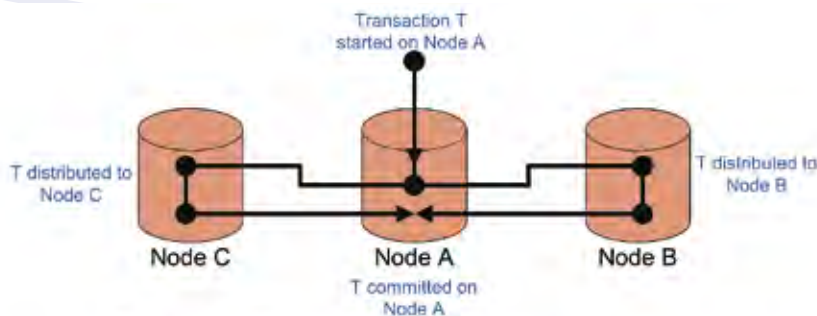
System architecture

Client/server

Client/server has been the predominant distributed computing model since the mid '90s. In this architecture, client applications send requests to a database or other shared application that resides on a server. The World Wide Web is global client/server computing. If you're reading this article online, your Web browser is likely the client with the article itself stored on a Web server, perhaps on another continent. Even when client and server processes reside on the same computer, they are separated by an interprocess communication layer.

Many embedded databases are based on this client/server model. But a handful of others, which might be called *more embedded* or *truly embedded*, consist not of client and server modules, but of code libraries (functions in the Java, C, and C++ languages) actually embedded in and compiled with a software application. Companies including Birdstep, McObject, and Empress provide such truly embedded databases.

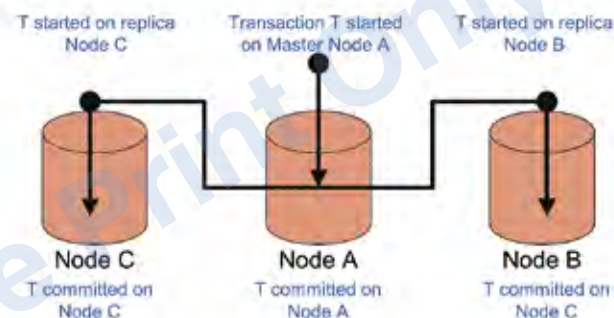
This architecture is simpler, which means it has a smaller code size and is less prone to defects because less complexity means less risk of things going wrong. Removing interprocess communication eliminates a performance barrier, and data storage and retrieval is further accelerated by eliminating server tasks such as the necessity to manage sessions and connections and to allocate and de-allocate resources on behalf of clients.



Eager replication:

- update as soon as a change occurs, high transaction durability, strong data consistency
- degraded transaction throughput

Figure 1



Lazy replication:

- update asynchronously, relaxed durability, and weaker data consistency
- higher transaction throughput

Figure 2

Advantages of client/server include the ability to appropriately size a network by installing the server software on a powerful computer with thin clients. In addition, client/server architecture lends itself naturally to supporting larger numbers of users. A server will typically start more threads as the workload increases, so if it's running on a multi-CPU box under symmetric multiprocessing, it can scale. In contrast, a database library offers no independent processing and relies on its host application to start additional threads.

High availability

High availability is so mandatory for some telecommunications equipment that it has its own slang. Systems with the requisite fault tolerance are said to have 5-nines or 99.999 percent uptime. This presents challenges for databases. An in-memory database can lose its stored content if memory fails, the server in client/server architecture can fail, and hard disk crashes can devastate an on-disk data store.

Database replication

Maintaining replicated copies of data distributed over multiple nodes can achieve high availability. Strategies for managing replicated databases include eager (synchronous) and lazy (asynchronous) replication, based on their approach to pushing out changes from the primary database to replica nodes. In *eager replication* (Figure 1), all updates are applied to the primary and replica nodes as part of the original transaction. While ensuring transaction integrity across nodes, the eager approach might exhibit longer resource holding time and, consequently, execute more slowly.

In contrast, *lazy replication* (Figure 2) commits a transaction without waiting for the updates to complete on the replicas. This means shorter resource holding time and faster

Employing a
time-cognizant eager
replication protocol
ensures predictable
resource-holding time.

performance. However, lazy replication is also more likely to lose a committed transaction if the primary node fails and transactions have not yet been propagated to the replicas. Generally speaking, lazy replication provides less transaction durability.

Another technology strategy for achieving high availability with predictable database response times is to apply a strict time requirement or processing deadline to eager replication. Employing a time-cognizant eager replication protocol ensures predictable resource-holding time. If the primary database instance does not receive acknowledgment of its communication by a preset deadline, it assumes the uncommunicative replica database has failed, decommissions it, and continues with normal processing.

Advantages bolster COTS database utilization

Growth of intelligent, connected devices is soaring. Whether found in the shirt pocket, automobile, airplane cockpit, or factory floor, these smart systems all incorporate embedded software and offer features that depend on searching, sorting, retrieving, and storing growing volumes of increasingly complex data. These are database management tasks, and to achieve them efficiently, embedded systems software developers are overcoming their propensity to build from scratch. Including a COTS database in an embedded systems project is not yet as common as using a commercial RTOS. But commercial databases' time to market, reliability, and resource consumption advantages are likely to drive the product category near that level of ubiquity. **ECD**

Steve Graves co-founded McObject in 2001. As the company's president and CEO, he has spearheaded McObject's growth and helped the company attain its goal of providing database technology that makes embedded systems smarter, more reliable, and more cost effective to develop and maintain. Prior to McObject, Steve was president and chairman of Centura Solutions Corporation and VP of worldwide consulting for Centura Software Corporation. He also served as president and COO of Raima Corporation. Steve is a member of the advisory board for the University of Washington's certificate program in Embedded and Real-Time Systems Programming.



To learn more, contact Steve at:

McObject LLC

22525 S.E. 64th Place, Suite 302

Issaquah, WA 98027

425-831-5964

steve@mcobject.com

www.mcobject.com