Re-Inventing Data Management For Intelligent Devices

Abstract: To survive in the market, intelligent devices such as set-top boxes and networking gear must rapidly expand their features. This means adding software "smarts" and managing larger volumes of more complex data—a challenge typically met with database management systems (DBMS). But traditional databases, with roots in business processing, present CPU and memory requirements that are too expensive for price-sensitive high-tech gear.

The situation demands a new kind of data management engine, designed to exclude unnecessary, business-oriented processing layers while delivering developer tools for the tightest possible integration. This paper examines the new on-device database requirements, and looks at one product, *e*X*treme*DB, developed in response to these needs.

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

Overview

Devices ranging from set-top boxes to fighter jet avionics require proven data management in order to support innovative new features and capabilities. Existing DBMSs, with their roots in business computing, provide only partial solutions in these radically different application domains. For economic as well as technical reasons, the ideal data management engine for high-tech gear must offer a tiny memory and CPU footprint, real-time performance, optimization for handling streams of complex data, and the ability to integrate tightly with device-based applications.

This paper examines one new product, McObject's *eXtreme*DB, that meets these challenges with an engine utilizing main memory data management and an architecture that has been designed with an understanding of what is required—and just as important, what represents excess baggage—in a device-based database management system.

Devices – Data Management's New Frontier

Growth in intelligent, connected devices is soaring, even as sales of traditional PCs level off. In offices and cars, atop TV sets, in pockets and purses, and built into industrial, communications and transportation systems, these hardware devices typically lack the familiar PC-style interface but offer advantages in size, mobility and ease of use. Once considered mere "embedded systems" with minimal smarts, such gear has evolved to include powerful CPUs and sophisticated software.

To support expanding feature sets, applications generally must manage larger volumes of more complex data. This computing truism has held true for devices, and as a result, many device manufacturers are exploring moving from self-developed data management solutions, to proven commercial database management systems (DBMSs). During a short period recently, McObject worked with manufacturers and systems including:

- A new digital television **set-top box** requiring data management to support a dynamic programming guide, including images and video clips

- A leading defense contractor seeking off-the-shelf database software for **navigation and mapping electronics** in combat jets

- A **blowout prevention system** for deep-sea oil and gas drilling, which continuously collects and analyzes environmental and machine data to assess risk

- A **smart telecommunications switch** for long-distance carriers, needing real-time user and equipment data in order to route calls

- A major Japanese consumer electronics manufacturer whose **MP3 players and other portable audio devices** required efficient, small-footprint software to manage song titles, artists, playlists and other music meta-data

- **Automobile diagnostics equipment** including a data store for pinpointing engine trouble and providing solutions

Which Data Management Technology Fits?

Device developers are turning to commercial database software with greater frequency—but existing solutions have not provided the ideal fit. Relational databases, the most widely used type, emerged well over two decades ago to support corporations' business functions. Their features include support for SQL—a high-level interface—and other business-oriented features such as lock arbitration, cache control, and abnormal termination recovery. But on a device—within a set-top box or next-generation fax machine, for example—these abilities are often unnecessary and cause the application to exceed the memory, CPU and storage resources available on the device.

Object-oriented DBMSs have also been promoted as a solution, but these, like relational databases, were originated to meet business objectives, and have been too slow for the real-time needs of intelligent devices. At various times, relational and object-oriented vendors offered lighter products by removing features. But these stripped-down relational and OO databases begged the question, "What would a database look like if it were designed from the start for intelligent, connected devices?"

Devices Are Different

Devices represent a new development domain, with unique needs. What constitutes the ideal database system for this new kind of application?

Small footprint. Manufacturers strive to reduce devices' memory, CPU and storage requirements for economic reasons. Many devices are intended for mass distribution, often in extremely competitive markets. Even a slightly lower per-unit price increases market share, and a lower per-unit cost drops right to the bottom line. Software that saves kilobytes of RAM, or requires a slightly less expensive processor, can determine product success.

Real-time performance. Latency, or the wait between system command and response, is an accepted part of desktop and client/server computing. Not so for devices—consumer electronics, communications gear, industrial controllers and weaponry systems are expected to perform instantly, and "hard real-time" systems must respond within pre-set bounds often measured in milliseconds. Data management for devices must be based on an inherently efficient architecture, with sources of performance overhead minimized or, better yet, eliminated.

Streams of complex data. Unlike business applications, devices rarely manage data that fits neat, tabular structures. High-tech gear often must work with complex data, such as trees and arbitrarily long arrays of simple or complex fields. And while business databases are optimized

to handle very large, complex transactions, devices more often deal with streams of small, fast transactions, such as a telecom switch routing ongoing calls according to available hardware channels, or a gaming device responding to the unfolding events of real-time play.

Developer features. Business databases are designed to be accessible to non-programmers using SQL, reporting packages, and other high-level tools. In contrast, devices are usually designed for dedicated purposes, with data access defined, in advance, by the developer. In addition, tight integration between data management code and the application creates greater run-time efficiency, reducing the need for CPU cycles and other computing resources. These factors emphasize the need for developer tools. Instead of providing querying languages for end-users, a database for intelligent devices must empower the developer with features that integrate with the powerful programming environments—namely C, C++ and Java—utilized in such projects.

Introducing *eXtreme*DB

McObject introduced *eXtreme*DB, the first data management engine built from scratch for intelligent, connected devices. Founded by software, database and embedded systems development veterans, McObject incorporated into its technology lessons from the real-world device projects described above. The result is *eXtreme*DB, the first small-footprint, in-memory database system optimized for fast streams of complex data, with powerful developer tools and an innovative architecture that bypasses many of the sources of performance overhead found in traditional DBMS software.

*eXtreme*DB's Architecture

*eXtreme*DB's design recognizes the unique performance needs of embedded systems—as well as the irrelevance of certain resource-intensive features found in business-oriented DBMSs. One important fact is that embedded systems are inherently *not* distributed. All of the processes or threads interacting with the database are running in the same CPU. This doesn't mean there isn't a network—a set-top box is part of a very large network. But one of the threads running in the set-top box application, not the database, is responsible for communication. For many embedded systems, this renders unnecessary remote procedure call mechanisms and other complex communications logic at the core of business databases. A database system for embedded systems should have remote procedure call mechanisms available when needed, but not carried along as dead weight when not needed.

*eXtreme*DB is also built with the recognition that intelligent device-based systems will present different requirements for databases, determined by the complexity of the system. A MP3 player or personal navigation device, for example, will have relatively few concurrent tasks, while a metro or core router will have many. To address these different needs, developers using *eXtreme*DB can choose between two different transaction managers, MURSIW (MUltiple Reader SIngle Writer) and MVCC (Multi Version Concurrency Control).

For assured data integrity, *eXtreme*DB fully supports ACID-compliant transactions. However, the data management engine's transaction queue has been designed for minimal resource

consumption. It also provides for dynamic queuing, which allows the application to assign priority to different kinds of transactions.

This lean architecture, along with several additional capabilities described below, keep *eXtreme*DB's footprint to a minimum—100K or less, depending on the processor and compiler. It also enables optimal performance on the less powerful processors favored for economic reasons in consumer devices.

Main Memory, Direct Access Database

*eXtreme*DB stores data entirely in main memory, eliminating disk access overhead. Main memory processing, combined with a transaction manager specifically designed for intelligent devices, accelerates the handling of streams of small, fast transactions. It also cuts overhead from data transfer. Earlier data management technology required copying records from database to cache, and then to a new location for manipulation by the application. With *eXtreme*DB, the application works directly with the data in main memory, eliminating the need for duplicate data sets as well as the requirement to move data from persistent to transient environments (see Figure 1).

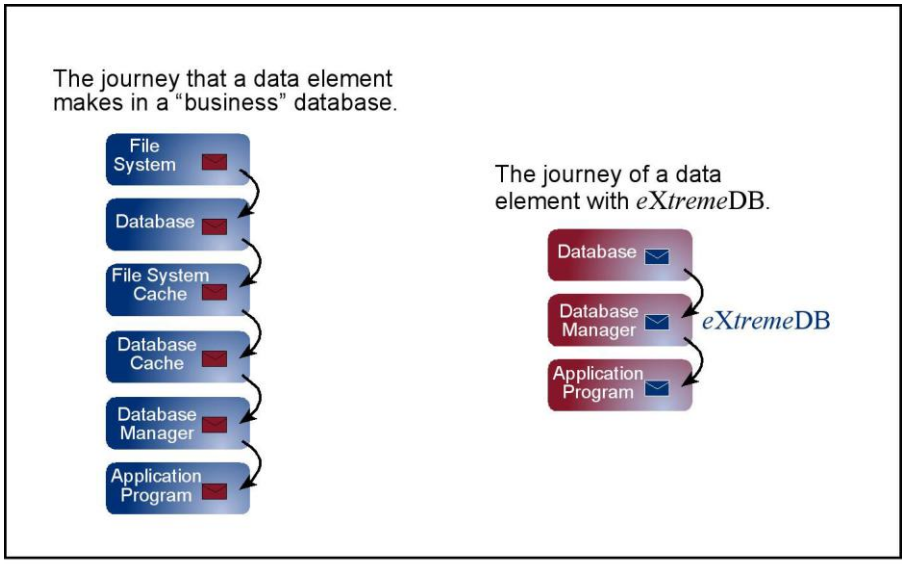

Figure 1

Unlike SQL databases, *eXtreme*DB stores information in the exact form in which it is used by the application. Other technologies require translation—mapping a C data element to a relational representation, for example, or requiring additional code to pick fields from tables and copy them to C structures. By eliminating this overhead, *eXtreme*DB reduces memory and CPU resource demands. Figure 2 illustrates the overhead of data decomposition and assembly imposed by a relational database when dealing with complex objects.

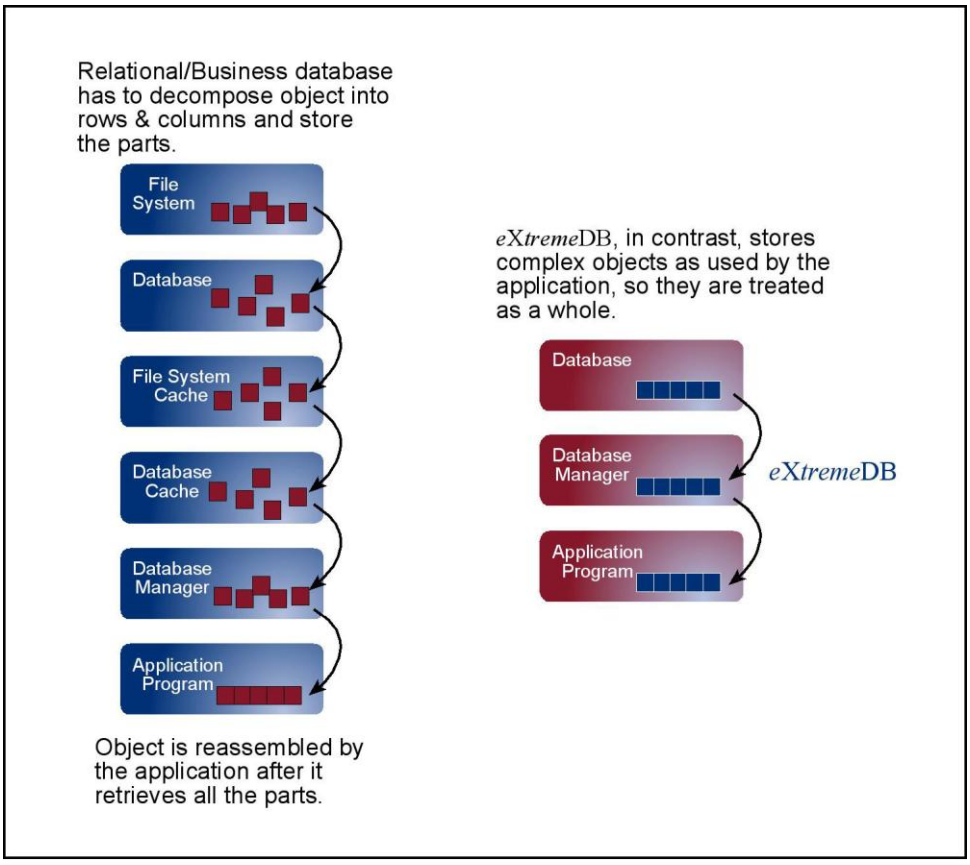

Figure 2

Developer Orientation

McObject understands that applications developed for intelligent devices will not "plug into" their databases. Rather, for optimal run-time efficiency, developers will tightly integrate data management with application software. Today, embedded systems are overwhelmingly developed in C/C++. *eXtreme*DB enhances the preferred development environment and supports the data-centric approach of object-oriented analysis and design. Important developer tools include *eXtreme*DB's support for complex data types and query methods, a rich and intuitive application programming interface, and a powerful debugging environment.

Support for complex data and efficient queries. Relational databases—even those permitting main memory hosting—require shoehorning data into neat rows and columns of basic data types. In contrast, *eXtreme*DB facilitates tight, efficient coding by supporting virtually all data types, including structures, vectors and BLOBs. For querying, McObject provides hash indexes for exact match searches; b-tree indexes for pattern match, range retrieval and sorting; and object-identifier references, for direct access. Rather than storing duplicate data, indexes contain only a reference to data, keeping memory requirements to an absolute minimum. (For more information, see "Appendix – Data Types and Storage Attributes, *eXtreme*DB vs. SQL database").

Programming interface. McObject provides a library of standard database functions, used in all *eXtreme*DB-based applications. However, most of the API for accessing persistent data in an application is generated by *eXtreme*DB when the database is compiled. Because it is based on the developer's own data design, this application-specific API is easy to learn and optimized for the project's exact needs.

Debugging environment. A developer edition of the McObject run-time takes advantage of numerous traps in the database code, detecting programming errors and enabling easy repair. In addition, compile-time type checking in the C environment applies to the methods McObject uses to access stored data. Methods that are generated to provide access to a certain object are type-safe, requiring a reference to that data type as a parameter. Any mistake will generate a compiler warning. This eliminates the possibility of data typing errors, a common source of expensive run-time bugs.

Conclusion

Gear manufacturers' two-fold mission remains constant: offer new capabilities—preferably *amazing* capabilities—continuously, and stay ahead of competitors by chipping away at price. In the past half decade, the software required for great new features crossed a threshold. Data grew sufficiently complex, and developers turned to databases, even when this meant shoehorning SQL databases where they didn't particularly fit. But just as relational databases arose to meet client/server needs, new data management technology is emerging for smart devices. The new software, evidenced by *eXtreme*DB, emphasizes developer flexibility, performance, and frugality in resource use as the three key requirements for device data management.

Appendix – Data Types and Storage Attributes

*e*X*treme*DB vs. SQL database

The table below compares data types and storage attributes supported by *e*X*treme*DB, with those of a traditional SQL-based relational database. Support for a wider range of data types, especially complex data types, is one of the main features that sets *e*X*treme*DB apart, both in flexibility to write more efficient code, and ability to support the complex data associated with device-based applications.

Vectors and **structures** are what give *e*X*treme*DB the ability to manage complex data. Without them, a database is left with simple (atomic) fields. With vectors and structures, the developer can address with a single data type information that in a relational database would require an entire table, and its accompanying processing overhead.

OID and **ref** enable the developer using *e*X*treme*DB to establish fast, efficient and natural relationships between classes that have natural object identifiers (OIDs) (e.g. a network of sensors would each have a unique identifier. Each measurement by a sensor would have a ref of the sensor that gathered the measurement).

The higher-level data types **Date**, **Time**, **Timestamp** and **Decimal** are not required by all applications; when footprint is important, it is better to leave the implementation of higher level types to the application rather than burden every application with the code to support them whether they are needed or not.

The *e*X*treme*DB data modifiers **Compact, Optional** and **Voluntary** are all important to minimizing the memory consumed by the database. Voluntary is also important for reducing CPU cycles. Declaring a class Compact will reduce the amount of overhead imposed by the database runtime to manage objects of the class: Compact objects are known to be less than 64K and can be used with a short integer (2 bytes) for the internal offsets, compared to non-compact objects requiring a large integer (4 bytes) for the offsets.

Optional fields do not consume space in the database unless they are explicitly populated by the application.

Voluntary means that an index is created only when the application causes it to be. Until it is created, or after it is dropped, it requires no space and the CPU cycles to maintain it during insert/update/delete operations are also not used. Index creation can be timed selectively to coincide with availability of computing resources.

Having **unsigned** types provides some built-in safety (you can’t accidentally store a negative number in a field that logically should never be negative, like weight).

*eXtreme*DB	SQL	Meaning
Char<n>	Char(N)	Fixed length character strings
String	Long Varchar	Variable length character strings < 64K
Signed<1>	Tinyint	One-byte signed integer
Signed<2>	Smallint	Two-byte signed integer
Signed<4>	Integer	Four-byte signed integer
Unsigned<1>		One-byte unsigned integer
Unsigned<2>		Two-byte unsigned integer
Unsigned<4>		Four-byte unsigned integer
Float	Real	Four-byte floating point number
Double	Float	Eight-byte floating point number
Blob	Long varbinary	Variable length binary (opaque) data
Vector		Variable length array of any (ex blob) type
OID		Unique object identifier
Ref		Reference to the OID of another object
Struct		Names a new type and specifies its fields, that can have different types
Optional		Declares a field that might not be stored
Date	Date	Vendor-specific storage format for YYYYMMDD dates
Time	Time	Vendor-specific storage for HHMMSS.dddd times
	Timestamp	Vendor-specific storage for YYYYMMDD:HHMMSS.dddd date/time
	Decimal	Vendor-specific storage for numeric (precision, scale) format, usually BCD
Compact		Declares that a class will always be < 64K, allowing McObject to minimize overhead
Voluntary		Declares an index that is built and destroyed on demand, saving space and CPU cycles to maintain it during periods when the index is not required or desired