Challenges in Embedded Database System Administration

Margo I. Seltzer, Harvard University
Michael A. Olson, Sleepycat Software

Database configuration and maintenance have historically been complex tasks, often requiring expert knowledge of database design and application behavior. In an embedded environment, it is not possible to require such expertise or to perform ongoing database maintenance. This paper discusses the database administration challenges posed by embedded systems and describes how Sleepycat Software's Berkeley DB architecture addresses these challenges.

1 Introduction

Embedded systems provide a combination of opportunities and challenges in application and system configuration and management. As an embedded system is most often dedicated to a single application or small set of cooperating tasks, the operating conditions of the system are typically better understood than those of general-purpose computing environments. Similarly, as embedded systems are dedicated to a small set of tasks, one expects that the software to manage them would be small and simple. On the other hand, once an embedded system is deployed, it must continue to function without interruption and without administrator intervention.

Database administration has two components, initial configuration and ongoing maintenance. Initial configuration includes database design, manifestation, and tuning. The instantiation of the design includes decomposing the data into tables, relations, or objects and designating proper indices and their implementations (e.g., B-trees, hash tables, etc.). Tuning the design requires selecting a location for the log and data files, selecting appropriate database page sizes, specifying the size of in-memory caches, and determining the practical limits of multi-threading and concurrency. As we expect that an embedded system is created by experienced and knowledgeable engineers, requiring expertise during the initial system configuration process is acceptable, and we focus our efforts on the ongoing maintenance of the system. In this way, our emphasis differs from other projects that focus on automating database design, such as Microsoft's AutoAdmin project [3]and the "no-knobs" administration that is identified as an area of important future research by the Asilomar authors [1].

In this paper, we focus on what the authors of the Asilomar report call "gizmo" databases [1], databases that reside in devices such as smart cards, toasters, or telephones. The key characteristics of these databases are that their functionality must be completely transparent to users, no explicit database operations or database maintenance is ever performed, the database may crash at any time and must recover instantly, the device may undergo a hard reset at any time (requiring that the database return to its initial state), and the semantic integrity of the database must be maintained at all times. In Section 2, we provide more detail on the sorts of tasks typically performed by database administrators (DBAs) that must be automated in an embedded system.

The rest of this paper is structured as follows. In Section 2, we outline the requirements for embedded database support. In Section 3, we discuss how Berkeley DB is conducive to the hands-off management required in embedded systems. In Section 4, we discuss novel features that enhance Berkeley DB's suitability for the embedded applications. In Section 5, we discuss issues of footprint size. In Section 6, we discuss related work, and we conclude in Section 7.

2 Embedded Database Requirements

Historically, much of the commercial database industry has been driven by the requirements of high performance online transaction processing (OLTP), complex query processing, and the industry standard benchmarks that have emerged (e.g., TPC-C [9], TPC-D [10]) to allow for system comparisons. As embedded systems typically perform fairly simple queries and only rarely require high, sustained transaction rates, such metrics are not nearly as relevant for embedded database systems as are ease of maintenance, robustness, and a small memory and disk footprint. Because of continuously falling hardware prices, robustness and ease of maintenance are the key issues of these three requirements. Fortunately, robustness and ease of maintenance are side effects of simplicity and good design. These, in turn, lead to a small size, contributing to the third requirement of an embedded database system.

2.1 The User Perspective

Users must be able to trust the data stored in their devices and must not need to manually perform any database or system administration in order for their gizmo to perform correctly.

In the embedded database arena, it is the ongoing maintenance tasks that must be automated, not the initial system configuration. There are five tasks traditionally performed by DBAs, which must be performed automatically in embedded database systems. These tasks are log archival and reclamation, backup, data compaction/reorganization, automatic and rapid recovery, and reinitialization from an initial state.

Log archival and backup are tightly coupled. Database backups are part of any recoverable database installation, and log archival is analogous to incremental backup. It is not clear what the implications of backup and archival are for an embedded system. Consumers do not back up their telephones or refrigerators, yet they do (or should) back up their personal computers or personal digital assistants. For the remainder of this paper, we assume that some form of backups are required for gizmo databases (consider manually re-programming a telephone that includes functionality to compare and select long-distance services based on the caller's calling pattern and connection times). Furthermore, these backups must be completely transparent and must not interrupt service, as users should not be aware that their gizmos are being backed up, will not remember to initiate the backups explicitly, and are unwilling to wait for service.

Data compaction or reorganization has traditionally required periodic dumping and restoration of database tables and the recreation of indices in order to bound lookup times and minimize database growth. In an embedded system, compaction and reorganization must happen automatically.

Recovery issues are similar in embedded and traditional environments with two significant exceptions. While a few seconds or even a minute of recovery is acceptable for a large server installation, consumers are unwilling to wait for an appliance to reboot. As with archival, recovery must be nearly instantaneous in an embedded product. Additionally, it is often the case that a system will be completely reinitialized, rather than performing any type of recovery, especially in systems that do not incorporate non-volatile memory. In this case, the embedded database must be restored to its initial state and must not leak any resources. This is not typically an issue for large, traditional database servers.

2.2 The Developer Perspective

In addition to the maintenance-free operation required of embedded database systems, there are a number of requirements based on the constrained resources typically available to the "gizmos" using gizmo databases. These requirements are a small disk and memory footprint, short code-path, programmatic interface (for tight application coupling and to avoid the time and size overhead of interfaces such as SQL and ODBC), support for entirely memory-resident operation (e.g., systems where file systems are unavailable), application configurability and flexibility, and support for multi-threading.

A small footprint and short code-path are self-explanatory; however, what is not as obvious is that the programmatic interface requirement is their logical result. Traditional interfaces, such as ODBC and SQL, add a significant size overhead and frequently add multiple context/thread switches per operation, not to mention several IPC calls. An embedded product is unlikely to require the complex and powerful query processing that SQL enables. Instead, in the embedded space, the ability for an application to obtain quickly its specific data is more important than a general query interface.

As some systems do not provide storage other than memory, it is essential that an embedded database work seamlessly in memory-only environments. Similarly, many embedded operating systems have a single address space architecture, so a fast, multi-threaded database architecture is essential for applications requiring concurrency.

In general, embedded applications run on gizmos whose native operating system support varies tremendously. For example, the embedded OS may or may not support user-level processes or multi-threading. Even if it does, a particular embedded application may or may not need it, e.g., not all applications need more than one thread of control. An embedded database must provide mechanisms to developers without deciding policy. For example, the threading model in an application is a matter of policy, and depends not on the database software, but on the hardware, operating system and library interfaces, and the application's requirements. Therefore, the data manager must provide for the use of multi-threading, but not require it, and must not itself determine what a thread of control looks like.

3 Berkeley DB: A Database for Embedded Systems

The current Berkeley DB package, as distributed by Sleepycat Software, is a descendant of the hash and B-tree access methods that were distributed with the 4BSD releases from the University of California, Berkeley. The original software (usually referred to as DB 1.85), was originally intended as a public domain replacement for the `dbm` and `ndbm` libraries that were proprietary to AT&T. Instead, it rapidly became widely used as a fast, efficient, and easy-to-use data store. It was incorporated into a number of Open Source packages including Perl, Sendmail, Kerberos, and the GNU standard C library. Versions 2.0 and later were distributed by Sleepycat Software (although they remain Open Source software) and added functionality for concurrency, logging, transactions, and recovery.

Berkeley DB is the result of implementing database functionality using the UNIX tool-based philosophy. Each piece of base functionality is implemented as an independent module, which means that the subsystems can be used outside the context of Berkeley DB. For example, the locking subsystem can be used to implement general-purpose locking for a non-DB application and the shared memory buffer pool can be used for any application performing page-based file caching in main memory. This modular design allows application designers to select only the functionality necessary for their application, minimizing memory footprint and maximizing performance. This directly addresses the small footprint and short code-path criteria mentioned in the previous section.

As Berkeley DB grew out of a replacement for `dbm`, its primary implementation language has always been C and its interface has been programmatic. The C interface is the native interface, unlike many database systems where the programmatic API is a layer on top of an already-costly query interface (e.g. embedded SQL). Berkeley DB's heritage is also apparent in its data model; it has none. The database stores unstructured key/data pairs, specified as variable length byte strings. This leaves schema design and representation issues the responsibility of the application, which is ideal for an embedded environment. Applications retain full control over specification of their data types, representation, index values, and index relationships. In other words, Berkeley DB provides a robust, high-performance, keyed storage system, not a particular database management system. We have designed for simplicity and performance, trading off the complex, general purpose support found in historic data management systems. While that support is powerful and useful in many applications, the overhead imposed (regardless of its usefulness to any single application), is unacceptable in embedded applications.

Another element of Berkeley DB's programmatic interface is its customizability. Applications can specify Btree comparison and prefix compression functions, hash functions, error routines, and recovery models. This means that an embedded application is able to tailor the underlying database to best suit its data demands and programming model. Similarly, the utilities traditionally bundled with a database manager (e.g., recovery, dump/restore, archive) are implemented as tiny wrapper programs around library routines. This means that it is not necessary to run separate applications for the utilities. Instead, independent threads can act as utility daemons, or regular query threads can perform utility functions. Many of the current products built on Berkeley DB are bundled as a single large server with independent threads that perform functions such as checkpoint, deadlock detection, and performance monitoring.

As described earlier, living in an embedded environment requires flexible management of storage. Berkeley DB does not require any preallocation of disk space for log or data files. While typical commercial database systems take complete control of a raw disk device, Berkeley DB cooperates with the native system's file system, and can therefore safely and easily share the file system with other applications. All databases and log files are native files of the host environment, so whatever standard utilities are provided by the environment (e.g., UNIX `cp`) can be used to manage database files as well.

Berkeley DB provides three different memory models for its management of shared information. Applications can use the IEEE Std 1003.1b-1993 (POSIX) `mmap` interface to share data, they can use system shared memory, as frequently provided by the `shmget` family of interfaces, or they can use per-process heap memory (e.g., `malloc`). Applications that require no permanent storage on systems that do not provide shared memory facilities can still use Berkeley DB by requesting strictly private memory and specifying that all databases be memory-resident. This provides pure-memory operation.

Finally, Berkeley DB is designed for rapid start-up, e.g., recovery happens automatically as part of system initialization. This means that Berkeley DB works correctly in environments where gizmos are shut down without warning and restarted.

4 Extensions for Embedded Environments

All the features described in the previous section are useful both in conventional systems as well as in embedded environments. In this section, we discuss a number of features and "automatic knobs" that are specifically geared toward the more constrained environments found in gizmo databases.

4.1 Automatic compression

Following the programmatic interface design philosophy, we support application-specific (or default) compression routines. These can be geared toward the particular data types present in the application's dataset, thus providing better compression than a general-purpose routine. Applications can specify encryption functions as well, and create encrypted databases instead of compressed ones. Alternately, the application might specify a function that performs both compression and encryption.

As applications are also permitted to specify comparison and hash functions, an application can chose to organize its data based either on uncompressed and clear-text data or compressed and encrypted data. If the application indicates that data should be compared in its processed form (i.e., compressed and encrypted), then the compression and encryption are performed on individual data items and the in-memory representation retains these characteristics. However, if the application indicates that data should be compared in its original form, then entire pages are transformed upon being read into or written out of the main memory buffer cache. These two alternatives provide the flexibility to choose the correct relationship between CPU cycles, memory and disk space and security.

4.2 In-memory logging & transactions

One of the four key properties of transaction systems is durability. This means that transaction systems are designed for permanent storage (most commonly disk). However, as described above, embedded systems do not necessarily contain any such storage. Nevertheless, transactions can be useful in this environment to preserve the semantic integrity of the underlying storage. Berkeley DB optionally provides logging functionality and transaction support regardless of whether the database and logs are on disk or in memory.

4.3 Remote Logs

While we do not expect users to backup their television sets and toasters, it is certainly reasonable that a set-top box provided by a cable carrier will need to be backed up by the cable carrier. The ability to store logs remotely can provide "information appliance" functionality, and can also be used in conjunction with local logs to enhance reliability. Furthermore, remote logs provide for true catastrophic recovery, e.g., loss of the gizmo, destruction of the gizmo, etc.

4.4 Application References to Database Buffers

In typical database systems, data must be copied from the data manager's buffer cache (or data pages) into the application's memory when it is returned to the application. However, in an embedded environment, the robustness of the total software package is of paramount importance, and maintaining an artificial isolation between the application and the data manager offers little advantage. As a result, it is possible for the data manager to avoid copies by giving applications direct references to data items in the shared memory cache. This is a significant performance optimization that can be allowed when the application and data manager are tightly integrated.

4.5 Recoverable database creation/deletion

In a conventional database management system, the creation of database tables (relations) and indices are heavyweight operations that are not recoverable. This is not acceptable in a complex embedded environment where instantaneous recovery and robust operation in the face of all types of database operations are essential. Berkeley DB provides transaction-protected file system utilities that allow us to recover both database creation and deletion.

4.6 Adaptive concurrency control

The Berkeley DB package uses page-level locking by default. This trades off fine-grained concurrency control for simplicity during recovery. (Finer-grained concurrency control can be obtained by reducing the page size in the database or the maximum number of items permitted on each page.) However, when multiple threads of control perform page-locking in the presence of writing operations, there is the potential for deadlock. As some environments do not need or desire the overhead of logging and transactions, it is important to provide the ability for concurrent access without the potential for deadlock.

Berkeley DB provides an option to perform coarser grained, deadlock-free locking. Rather than locking on pages, locking is performed at the interface to the database. Multiple readers or a single writer are allowed to be active in the database at any instant in time, with conflicting requests queued automatically. The presence of cursors, through which applications can both read and write data, complicates this design. If a cursor is currently being used for reading, but will later be used to write, the system will be deadlock-prone unless special precautions are taken. To handle this situation, the application must specify any future intention to write when a cursor is created. If there is an intention to write, the cursor is granted an intention-to-write lock, which does not conflict with readers, but does conflict with other intention-to-write and write locks. The end result is that the application is limited to a single potentially writing cursor accessing the database at any point in time.

Under periods of low contention (but potentially high throughput), the normal page-level locking provides the best overall throughput. However, as contention rises, so does the potential for deadlock. At some cross-over point, switching to the less concurrent, but deadlock-free locking protocol will ultimately result in higher throughput as operations must never be retried. Given the operating conditions of an embedded database manager, it is useful to make this change automatically as the system detects high contention in the application's data access patterns.

4.7 Adaptive synchronization

In addition to the logical locks that protect the integrity of the database pages, Berkeley DB must synchronize access to shared memory data structures, such as the lock table, in-memory buffer pool, and in-memory log buffer. Each independent module uses a single mutex to protect its shared data structures, under the assumption that operations that require the mutex are very short and the potential for conflict is low. Unfortunately, in highly concurrent environments with multiple processors present, this assumption is not always true. When this assumption becomes invalid (that is, we observe significant contention for the subsystem mutexes), Berkeley DB can switch over to a finer-grained concurrency model for the mutexes. Once again, there is a performance trade-off. Fine-grained mutexes impose a penalty of approximately 25% (due to the increased number of mutex acquisitions and releases for each operation), but allow for higher overall throughput. Using fine-grained mutexes under low contention would cause a decrease in performance, so it is important to monitor the system carefully, so that the change happens only when it will increase system throughput without jeopardizing latency.

4.8 Source code availability

Finally, full source code is provided for Berkeley DB. This is an absolute requirement for debugging embedded applications. As the library and the application share an address space, an error in the application can easily manifest itself as an error in the data manager. Without complete source code for the library, debugging easy problems can be difficult, and debugging difficult problems can be impossible.

5 Footprint of an Embedded System

Embedded database systems compete on disk and memory footprint as well as the traditional metrics of features, price and performance. In this section of the paper, we focus on footprint.

Oracle reports that Oracle Lite 3.0 requires 350 KB to 750 KB of memory and approximately 2.5 MB of hard disk space [7]. This includes drivers for interfaces such as ODBC and JDBC. In contrast, Berkeley DB ranges in size from 75 KB to under 200 KB, foregoing heavyweight interfaces such as ODBC and JDBC. At the low end, applications requiring a simple single-user access method can choose from either extended linear hashing, B+ trees, or record-number based retrieval and pay only the 75 KB space requirement. Applications requiring all three access methods will observe the 110 KB footprint. At the high end, a fully recoverable, high-performance system occupies less than a quarter megabyte of memory. This is a system you can easily incorporate in a toaster oven. Table 1 shows the per-module breakdown of the Berkeley DB library. (Note, however, that this does not include memory used to cache database pages.)

6 Related Work

Every three to five years, leading researchers in the database community convene to identify future directions in database research. They produce a report of this meeting, named for the year and location of the meeting. The most recent of these reports, the 1998 Asilomar report, identifies the embedded database market as one of the high growth areas in database research [1]. Not surprisingly, market analysts identify the embedded database market as a high-growth area in the commercial sector as well [5].

The Asilomar report identifies a new class of database applications, which they term "gizmo" databases, small databases embedded in tiny mobile appliances, e.g., smart-cards, telephones, personal digital assistants. Such databases must be self-managing, secure and reliable. They will require plug

Subsystem	Object Size (in bytes)		
	Text	**Data**	**Bss**
Btree-specific routines	28812	0	0
Recno-specific routines	7211	0	0
Hash-specific routines	23742	0	0
Memory Pool	14535	0	0
Access method common code	23252	0	0
OS compatibility	4980	52	0
Support utilities	6165	0	0
Full Btree	77744	52	0
Full Recno	84955	52	0
Full Hash	72674	52	0
All Access Methods	108697	52	0
Locking	12533	0	0
Recovery	26948	8	4
Logging	37367	0	0
Full Package	**185545**	**60**	**4**

Table 1. Berkeley DB memory footprint.

and play data management with no database administrator (DBA), no human configurable parameters, and the ability to adapt to changing conditions. More specifically, the Asilomar authors claim that the goal is self-tuning, including defining the physical and logical database designs, generating reports, and executing utilities [1]. To date, few researchers have accepted this challenge, and there is a dearth of research literature on the subject.

Our approach to embedded database administration is fundamentally different from that described by the Asilomar authors. We adopt their terminology, but view the challenge in supporting gizmo databases to be that of self-sustenance *after* initial deployment. We find it, not only acceptable, but desirable to expect application developers to control initial database design and configuration. To the best of our knowledge, none of the published work in this area addresses this approach.

As the research community has not provided guidance in this arena, most work in embedded database administration has been done by the commercial vendors. These vendors can be partitioned into two groups: companies selling databases specifically designed for embedding or programmatic access and the major database vendors (e.g., Oracle, Informix, Sybase). The embedded vendors all acknowledge the need for automatic administration, but normally fail to identify precisely how their products accomplish this. A notable exception is Interbase, whose white paper comparison with Sybase and Microsoft's SQL servers explicitly addresses features of maintenance ease. Interbase states that as they use no log files, there is no need for log reclamation, checkpoint tuning, or other tasks normally associated with log management. However, Interbase uses Transaction Information Pages, and it is unclear how these are reused or reclaimed [6]. Additionally, with a log-free system, they must use a FORCE policy (flushing all modified pages to disk at commit), as defined by Haerder and Reuter [4]. This has serious performance consequences for disk-based systems. Berkeley DB does use logs and therefore requires log reclamation, but provides interfaces such that applications may reclaim logs safely and programmatically. While Berkeley DB requires checkpoints, the goal of tuning the checkpoint interval is to bound recovery time. The checkpoint interval can be expressed as an amount of log data written. The application designer sets a target recovery time, and selects the amount of log data that can be read in that interval and specifies the checkpoint interval appropriately. Even as load changes, the time to recover does not.

The backup approaches taken by Interbase and Berkeley DB are similar in that they both allow online backup, but rather different in their effect on transactions running during backup. As Interbase performs backups as transactions [6], concurrent queries potentially suffer long delays. Berkeley DB uses native operating system utilities and recovery for backups, so there is no interference with concurrent activity, other than potential contention on disk arms.

There are a number of other vendors selling products into the embedded market (e.g., Raima, Centura, Pervasive, Faircom), but none of these highlight the special requirements of embedded database applications. On the other end of the spectrum, the major vendors (e.g., Oracle, Sybase, Microsoft), are all clearly aware of the importance of the embedded market. As discussed earlier, Oracle has announced its Oracle Lite server for embedded use. Sybase has announced its UltraLite platform for "application-optimized, high-performance, SQL database engine for professional application developers building solutions for mobile and embedded platforms" [8]. We believe that SQL is fundamentally incompatible with the gizmo database environment and the truly embedded systems for which Berkeley DB is most suitable. Microsoft research is taking a different approach, developing technology to assist in automating initial database design and index specification [2][3]. As described earlier, such configuration is, not only acceptable in the embedded market, but desirable so that application designers can tune their entire applications for the target environment.

7 Conclusions

The coming wave of embedded systems poses a new set of challenges for data management. The traditional server-based, large footprint systems designed for high performance on big iron are not the correct technical answer for this environment. Instead, application developers need small, fast, versatile data management tools that can be integrated easily within specific application environments. In this paper, we have identified several of the key issues in designing these systems and shown how Berkeley DB provides many of these requirements.

8 References

[1] Bernstein, P., Brodie, M., Ceri, S., DeWitt, D., Franklin, M., Garcia-Molina, H., Gray, J., Held, J., Hellerstein, J., Jagadish, H., Lesk, M., Maier, D., Naughton, J., Pirahesh, H., Stonebraker, M., Ullman, J., "The Asilomar Report on Database Research," *SIGMOD Record 27*(4): 74–80, 1998.

[2] Chaudhuri, S., Narasayya, V., "AutoAdmin 'What-If' Index Analysis Utility," *Proceedings of the ACM SIGMOD Conference*, Seattle, 1998.

[3] Chaudhuri, S., Narasayya, V., "An Efficient, Cost-Driven Index Selection Tool for Microsoft SQL Server," *Proceedings of the 23rd VLDB Conference*, Athens, Greece, 1997.

[4] Haerder, T., Reuter, A., "Principles of Transaction-Oriented Database Recovery," *Computing Surveys 15*,4 (1983), 237–318.

[5] Hostetler, M., "Cover Is Off A New Type of Database," Embedded DB News, `http://www.theadvisors.com/embeddeddb-news.htm`, 5/6/98.

[6] Interbase, "A Comparison of Borland InterBase 4.0 Sybase SQL Server and Microsoft SQL Server," `http://web.interbase.com/products/doc_info_f.html`.

[7] Oracle, "Oracle Delivers New Server, Application Suite to Power the Web for Mission-Critical Business," `http://www.oracle.com.sg/partners/news/newserver.htm`, May 1998.

[8] Sybase, Sybase UltraLite, `http://www.sybase.com/products/ultralite/beta`.

[9] Transaction Processing Council, "TPC-C Benchmark Specification, Version 3.4," San Jose, CA, August 1998.

[10] Transaction Processing Council, "TPC-D Benchmark Specification, Version 2.1," San Jose, CA, April 1999.