Embedded Databases

D. Cox

1Corporate Research, ABB Corporation, Raleigh, NC, US

Abstract The world’s first data management systems were, in fact, part of embedded systems and were designed to manage data generated by devices such as space craft guidance systems. Over time these data management systems evolved as off-the-shelf databases independent of embedded systems. Today, commercial and open-source databases manage data from countless domains. However, given the overriding need for high performance and high reliability, embedded systems have largely continued to rely on specialized solutions rather than their off-the-shelf “offspring”. This paper reviews a number of off-the-shelf databases to see where they now stand – particularly with an eye on performance, a critical quality attribute of embedded systems.

Keywords: embedded databases, in-memory databases, relational databases

1 Introduction

1.1 Historical Perspective

Embedded systems trace their roots to the first inertial guidance systems for rockets and space craft. A pioneer in the field was CS Draper at the MIT Instrumentation Laboratory who was instrumental in designing the Apollo Guidance System for the Apollo moon missions [4]. The moon missions stimulated research leading to the miniaturization of electronics and ultimately the computer. Miniaturization, in turn, led to the incorporation of computers and sensors in objects other than rockets. For example, the first computer embedded in an automobile was in a Volkswagen in 1968 to control the fuel injection system [14].

Embedded computers have always been married to sensors. Initially the data collected by these sensors was used to calculate values such as the velocity, acceleration, and direction of a moving vehicle. Knowing these derived values allowed the computer to adjust course to automatically allow a vehicle (e.g. a rocket) reach its destination safely and on schedule.

In these early systems the amount of data to manage was small owing to the small number of measured parameters and the lack of need to archive data. In time, however, more complex systems evolved that collected data from dozens to hundreds of sensors. Moreover, the need to archive data arose. As a consequence, strategies evolved for managing data.

The simplest approach for managing data was to store records of data in arrays or files. Searching these linear structures was straightforward – begin at the top of a list and step through the list until the desired match is found. For faster searching a hash value could be calculated for a record’s key and used to locate the desired data in a single step.

In the 1960’s Charles Bachman created the Integrated Data Store (IDS) using a navigational data model (also known as the network data model. Bachman’s goal was to create a database to automate business processes of the General Electric Low Voltage Switch Gear Department in Philadelphia,) [5]. Locating data in navigational databases requires following references from record to record until the desired record is found. Although not optimal, on average the navigational method shortened search times and allowed arbitrary relationships between records.

By the 1970’s the relational model had been introduced by EF Codd and by the 1980’s was becoming widely adopted by the IT industry and supplanting the navigational model [1]. Much has been written about the relational model and the reasons it has come to dominate modern database systems. Suffice it to say that despite their dominance in the industry, relational databases did not become generally employed in embedded systems. There are several reasons.

First, embedded systems have performance requirements that available relational databases have historically not been able to satisfy. The lack of adequate performance was particularly true for early relational databases [9].

Second, embedded systems require high reliability. They operate with little or no human interaction. For many applications they simply cannot fail. Relational databases can be fault tolerant. However, achieving fault tolerance often meant further sacrificing performance [9].

Third, the licensing and costs of commercial databases prevented their adoption in embedded systems [9].

Fourth, there was often a mismatch between embedded hardware and operating systems and the available relational databases. In short, relational database software did not exist for embedded systems that have historically consisted of proprietary hardware and software.
Beyond Performance

Performance is the key quality attribute for the mission critical operations of many embedded systems. In contrast, embedded systems also perform operations where performance is not so critical. An example is system configuration. Embedded systems interact with many devices – possibly hundreds to thousands of them. Configuring a system requires data that is loaded at startup or as devices are brought online. This data does not change unless the system is reconfigured in some fashion. However, doing so typically does not require real-time performance.

Another example is tracking asset health. Throughout their operation the health of devices might be periodically checked and written to an asset health database. It is common to collect and monitor this data over time to ensure a system’s continued operation. Typically, such checks are performed a few to several times a day and, again, does not require real-time performance.

In these cases, off-the-shelf databases can certainly provide the required performance. As will be presented in the Results section of this paper, off-the-shelf databases also provide numerous other desirable features ranging from flexible query processing to triggers to extensive transaction processing making them suitable for the non-mission critical operations of embedded system.

Tradeoffs

Off-the-shelf databases can be more cost effective for the embedded developer because the cost of database software development is avoided. With commercial databases the benefit is usually a licensing cost that compares favorably with the cost of in-house development. With open source databases there is often no licensing cost. Well established commercial or open source offerings are generally well tested and well documented. Off-the-shelf databases often provide more features and capabilities. Frequently off-the-shelf databases can also be extended to provide new features that are not provided “out of the box”.

However, with these benefits also come risks. Maintenance support may be hard to find or not delivered in a timely manner. When problems arise, it might be impossible to fix or work around those problems. As a consequence, there might be a loss of functionality or even serious crippling of the embedded software. However, if the benefits of off-the-shelf databases outweigh these risks, then an off-shelf-solution can be appropriate.

This paper takes a fresh look at several available off-the-shelf databases to help weigh these considerations.
2 Methods

Many embedded systems operate on cycles of duration one second or less. Within each cycle the embedded system will acquire data (often from sensors), perform calculations, and report and update results. Within each cycle the embedded system will also retrieve previously stored data and, when finished, save data. For the fastest performance, data is saved in-memory to avoid the latencies of accessing disk and other, slower non-volatile memory.

As a consequence, this study looked exclusively at off-the-shelf data management software that provides an in-memory option. The software examined include SQLite, Oracle Times Ten, Oracle Berkeley DB, MySQL Cluster, VoltDB, and memSQL. This study did not examine divide-and-conquer approaches such as Hadoop MapReduce or Spark. On a single machine those approaches do not offer any significant advantages. Their real power comes from distributing work in parallel across many machines. Even with commodity hardware, building an embedded system based on many machines is not cost effective when requirements can be met with a single system. Consequently, these approaches were not studied.

SQLite, Times Ten, MySQL Cluster, VoltDB, and memSQL are relational databases. The main interface to these databases is with SQL statements. In contrast, Berkeley DB is a key/value data store and access to data is via key values. An optional SQL interface is available for Berkeley DB. However, that interface negatively impacts performance. Consequently, the SQL interface to Berkeley DB was not studied.

The literature was reviewed for each of these offerings and a summary of findings is provided in the next section. Three offerings (SQLite, Times Ten, and Berkeley DB) were selected for actual testing. SQLite was selected because it is well established, has favorable licensing and is nearly universally availability. Times Ten and Berkeley DB were also selected for these same reasons except for licensing. To incorporate Times Ten or Berkeley DB in a commercial product does require a commercial license as do the remaining offerings.

Tests were performed by constructing an in-memory data table consisting of 30,000 records. This size was chosen as one typical of many embedded applications. Once created, tests were performed to time how long it takes to read all records from the database and to update all records. All tests were performed using a Lenovo laptop running 64 bit Windows 7. The laptop was configured with 32 GB of memory and an Intel i7-4800MQ CPU operating at 2.70GHz.

The next section of this paper presents the results of the literature review and the results of the tests that were conducted.

3 Results

Table 1 summarizes the findings for SQLite, Oracle Times Ten, VoltDB, MySQL Cluster, Berkeley DB, and memSQL. All offerings provide an option for creating an in-memory database. All offerings except Berkeley DB provide a native relational data model. In contrast, Berkeley DB provides a key/value data model. An option for a SQL interface is available for Berkeley DB. However, the literature indicates that this interface negatively impacts performance. Consequently, the native key/value API was only considered for Berkeley DB.

Given the critical nature of performance for embedded applications, tests were performed for SQLite, Oracle Times Ten, and Berkeley DB rather than simply relying on values reported in the literature. Figure 1 illustrates the number of fetches (i.e. records read) per second and the number of updates per second that were obtained for each of these products.

![Relative Database Performance](image)

**Figure 1**

According to the literature the performance of Oracle Times Ten is reported as an average response time. For update transactions the average response time is reported as 7.67 microseconds per record. For read transactions the average response time is reported as 2.37 microseconds. These values translate into 130K updates per second and 422K reads per second. Exactly how these results were achieved was not explained. It should be noted that these results are ten times faster than what we observed.

The performance reported in the literature for SQLite is somewhat lower than the results noted here. Read and updates were reported as 12K per second and 16K per second respectively. We observed 45K to 55K reads per second and 30K to 46K updates per second. The differences might be accounted for by differences in hardware, database size, software version, etc.

A 2006 white paper for Berkeley DB reports about 1M single record reads per second and about 600K single record writes per second. All operations were non-transactional without the benefit of record locking. Results for transactional systems...
are reported as 125K single record writes per second and 260K single record reads per second (on Windows, results were higher for Linux). These transactional results are consistent with the results that we observed (up to 109K writes per second and 157K reads per second).

<table>
<thead>
<tr>
<th>Feature</th>
<th>SQLite</th>
<th>Oracle Times Ten</th>
<th>VoltDB</th>
<th>MySQL Cluster</th>
<th>Berkeley DB</th>
<th>memSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paradigm</td>
<td>Relational</td>
<td>Relational</td>
<td>Relational</td>
<td>Relational</td>
<td>Key/Value</td>
<td>Relational</td>
</tr>
<tr>
<td>Platforms</td>
<td>iOS, Android, Windows, Linux, OS-X</td>
<td>Linux, Unix, Windows</td>
<td>Runs under Java VM</td>
<td>Linux, Solaris, Windows</td>
<td>Linux, Unix, Windows, Android, iOS, VxWorks</td>
<td>64 bit Linux. Minimum 3 nodes w/ 64 to 96GB memory</td>
</tr>
<tr>
<td>Architecture</td>
<td>In Process / Application Library</td>
<td>In Process / Application Library</td>
<td>Client / Server</td>
<td>Client / Server</td>
<td>In Process / Application Library</td>
<td>Client / Server</td>
</tr>
<tr>
<td>Storage Capabilities</td>
<td>Volatile or non-volatile memory. Storage option: B-tree</td>
<td>In memory with options for disk-based logging and checkpoints to ensure durability. Storage options: hash, T-tree, B-tree</td>
<td>In memory with options for disk-based logging and checkpoints to ensure durability. Storage options: B-tree, hash</td>
<td>In memory with disk-based logging and checkpoints to ensure durability. Storage options: simple array, hash table, B-tree, queue</td>
<td>In memory with options for logging and check pointing</td>
<td></td>
</tr>
<tr>
<td>Query Capabilities</td>
<td>SQL</td>
<td>SQL</td>
<td>SQL</td>
<td>SQL</td>
<td>Key/Value</td>
<td>SQL</td>
</tr>
<tr>
<td>Programming Environments</td>
<td>Eclipse, Visual Studio, Other</td>
<td>Eclipse, Visual Studio, Other</td>
<td>Eclipse</td>
<td>Eclipse, Visual Studio, Other</td>
<td>Eclipse, Visual Studio, Other</td>
<td>Eclipse, Visual Studio, Other</td>
</tr>
<tr>
<td>Programming Languages</td>
<td>C/C++, Perl, Python, Ruby, Java, Tcl, PHP</td>
<td>SQL, PL/SQL via JDBC, ODBC, ODP.NET, Oracle Call Interface, and Pro C/C++</td>
<td>Java, SQL</td>
<td>C, C++, .NET languages, Java, Other</td>
<td>C, C++, C#, Java, TCL</td>
<td>JDBC/ODBC supported languages</td>
</tr>
<tr>
<td>Transactions Locking</td>
<td>Transctions supported Locking dependent on mechanisms of the underlying system</td>
<td>Supports SQL transactions and database, table, and row locking. Row locking offers the best concurrency. It is possible to set a wait time for acquiring a lock.</td>
<td>VoltDB uses a single threaded execution engine. Thus, database accesses happen serially eliminating need for locking</td>
<td>Transactions supported Row and table locking with lots of flexibility (e.g. next key locks, gap locks, etc)</td>
<td>Transactions supported with rollback and automatic recovery</td>
<td>Supports transactions and avoids locking by using lock-free skiplists.</td>
</tr>
<tr>
<td>Triggers</td>
<td>Supports triggers that fire before or after a database event. Events include: deletions, insertions, and updates. Triggers are written in SQL.</td>
<td>Supports triggers and includes a transaction log API (XLA) that detects database updates and performs actions with lower overhead and higher performance than triggers.</td>
<td>Does not currently support triggers.</td>
<td>Triggers are supported to invoke stored procedures when specified actions occur.</td>
<td>Triggers are not supported through the native API.</td>
<td>Does not support triggers</td>
</tr>
<tr>
<td>Feature</td>
<td>SQLite</td>
<td>Oracle Times Ten</td>
<td>VoltDB</td>
<td>MySQL Cluster</td>
<td>Berkeley DB</td>
<td>memSQL</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------------------</td>
<td>-------------------</td>
<td>-------------------------</td>
<td>-----------------------------</td>
<td>----------------------------</td>
<td>----------------------------</td>
</tr>
<tr>
<td><strong>ACID</strong> (atomic, consistent, isolated, durable)</td>
<td>SQLite is ACID compliant</td>
<td>ACID compliant. Durability achieved with a combination of transaction logging and database check pointing.</td>
<td>ACID compliant. Durability achieved with a combination of transaction logging and database check pointing.</td>
<td>ACID compliant. Durability achieved with a combination of transaction logging and database check pointing.</td>
<td>ACID compliant. Durability achieved with a combination of transaction logging and database check pointing.</td>
<td>ACID compliant with logging and snapshots (similar to checkpoints) for ensuring durability.</td>
</tr>
<tr>
<td>Security</td>
<td>No built-in security</td>
<td>Provides access control</td>
<td>Provides minimal access control</td>
<td>Provides access control</td>
<td>No built-in security</td>
<td>Provides access control</td>
</tr>
<tr>
<td>Scalability</td>
<td>Size limited by available memory</td>
<td>Multiple CPUs. Data size limited by available memory.</td>
<td>Multiple nodes. Within a node, size limited to available memory.</td>
<td>Multiple nodes. Within a node, size limited to available memory.</td>
<td>Limited by the size of available memory.</td>
<td>Multiple nodes. Within a node, size limited to available memory.</td>
</tr>
<tr>
<td>Performance</td>
<td>Non-transactional Selects: 45K Updates: 30K All per second</td>
<td>Selects: 39K Updates: 21K All per second</td>
<td>250,000 transactions per second for a 3 node system. Assumes 90% read; 10% write transactions</td>
<td>250,000 transactions per second per the literature. This was not tested.</td>
<td>BTree write: 85K BTree read: 119K Hash write: 51K Hash read: 81K Array write:109K Array read: 157K All per second</td>
<td>20,000 writes per second per CPU in a cluster and about 80,000 queries per second</td>
</tr>
<tr>
<td>Extensibility</td>
<td>SQLite API can provide additional functionality through user defined functions.</td>
<td>TimesTen is extensible through the use of stored procedures.</td>
<td>VoltDB supports stored procedures as a method for extending the database's functionality.</td>
<td>MySQL supports stored procedures as a method for extending the database's functionality.</td>
<td>The native API does not provide stored procedures or other non-API mechanisms for extending database functionality.</td>
<td>Does not support stored procedures or other non-API mechanisms for extending database functionality.</td>
</tr>
<tr>
<td></td>
<td>SQLite is a general purpose relational database. Primary interface is SQL. There are several third party tools available to creating and managing databases.</td>
<td>Time Ten is a general purpose relational database. The primary interface is SQL. There are several tools available for creating and managing databases.</td>
<td>VoltDB is a general purpose relational database. The primary interface is SQL. There doesn't appear to be many additional tools.</td>
<td>MySQL is a general purpose relational database. The primary interface is SQL. There are several tools available for creating and managing databases.</td>
<td>The primary interface is the Berkeley DB API with an optional SQL interface. There are command line tools for database management.</td>
<td>The primary interface is with command line tools or third party tools capable of interface to databases via JDBC/ODBC.</td>
</tr>
<tr>
<td>Tools</td>
<td>Command line tool for managing databases. However, several third party tools are available.</td>
<td>Utility programs for interactive SQL, bulk copy, backup/restore, database migration, and system monitoring.</td>
<td>None identified</td>
<td>Utility programs for interactive SQL, bulk copy, backup/restore, database migration, and system monitoring.</td>
<td>Provides command line tools including performance monitoring tool called Drace.</td>
<td>Provides command line tools but third party tools that can manage databases via JDBC/ODBC are possible.</td>
</tr>
</tbody>
</table>
4 Discussion

4.1 Mission Critical Operations

Off-the-shelf databases offer an alternative to in-house database development. But are they appropriate for the mission critical operations of embedded systems? Appropriateness ultimately depends on a system’s specific requirements. For embedded systems the primary requirement for runtime databases will be performance possibly including hard real-time deadlines that must be met. For each cycle of execution can the software read and write the required number of records within those prescribed deadlines?

If your cycle time is one second, then off-the-shelf databases should handle a few tens of thousands of records to several tens of thousands of records – depending on the choice of database. If your requirements are more stringent – e.g. shorter cycle times, more records – then off-the-shelf might not be the answer.

Can off-the-shelf databases meet hard real-time deadlines? Nothing was found in the literature for the products examined to suggest that they can consistently meet specified hard real-time deadlines. Yes, the products that were examined are very fast because the data resides in memory rather than disk. However, even after testing specific database operations on specific hardware, there is likely to be variation in latency. If you absolutely need a specified response one hundred percent of the time, then you should consider crafting a custom solution.

However, if your requirements are soft real-time deadlines – that is if you can live with operations that fail to meet specified deadlines some of the time, then these solutions might work. In most cases operations will execute within the necessary time span. However, there might be times when an operation fails to complete within that time span. Again, it all depends on the specific situation and requirements and whether or not an application can accept such outliers.

Aside from performance, off-the-shelf databases relieve the need for in-house development and provide features that might be difficult or impossible to develop in-house. One can benefit from off-the-shelf software that has been widely tested and demonstrated to be reliable. On the other hand, one does relinquish control of the software. If a bug is discovered, workarounds might be difficult or impossible to achieve until a fix is available. In the meantime, there is a potential loss of features and capabilities. In some cases, the loss of features and capabilities could cripple the embedded software.

Lastly, for off-the-shelf solutions there are licensing fees and terms and conditions. For SQLite, the restrictions are few. For others such as Berkeley DB and Times Ten, the licensing might be prohibitive.

4.2 Non-Mission Critical Operations

For non-mission critical operations such as configuration and tracking asset health, off-the-shelf databases are solutions that offer many features and more than adequate performance. They provide the flexibility to handle a wide variety of data types. Except for the newest offerings, off-the-shelf databases have been widely tested. That doesn’t mean to say that they are perfect. However, one can have very high confidence in their reliability and performance for these uses.

As embedded systems become more prevalent (particularly with the Internet of Things), the need to manage data on devices will continue to grow. No doubt the capabilities of off-the-shelf databases will continue to grow as well.
References


