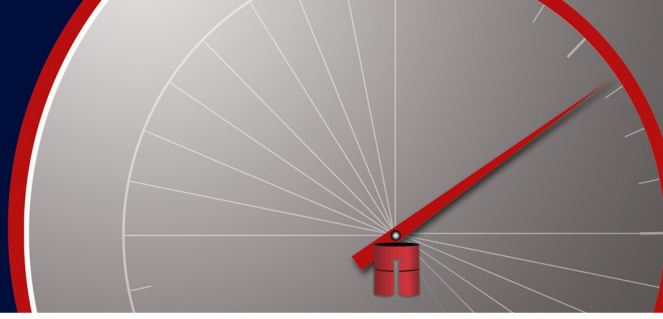
ITTIA
The Power to Manage Data



Benefits of Database for Embedded Systems

Table of Contents

Introduction

Many developers with a background in enterprise database are now building software for embedded, intelligent, and mobile devices. Other developers have been working building embedded systems for years and are now encountering the challenges of data management. This white paper addresses the question of how to leverage these experiences and effectively utilize embedded database technology.

Some of the benefits of an embedded database are:

- Relational Database Functionality
- Small Footprint
- Linked Library
- Lower Price & TCO
- Cross-platform Portability
- Shorter Time to Market
- Shorter Sales Cycle
- Ease of Use

What is an embedded database?

An embedded database is a software library used by application developers to store data.

The library adds database features to the application such as transaction logging, scalable index algorithms, and isolated concurrency.

Unlike enterprise databases, an embedded database is distributed with the application and is not installed separately by the end-users.

Embedded databases are especially well-suited for special-purpose devices and embedded systems with limited resources and a dedicated user interface.

Embedded systems, intelligent devices, and mobile devices are distributed with built-in software developed for a specific hardware environment. These applications are generally developed within a strict deadline and limited budget. To meet the deadline and satisfy hardware restrictions, many developers build custom solutions from scratch, preferring a quick, simple implementation to a scalable one. As the product line evolves, these solutions can become very complex and difficult to reconcile. Other developers will search for outside solutions to benefit from and build on the experience of others.

Data management is a problem now faced by a growing number of embedded developers. When an application needs to store information, a developer can either build a framework for data management from scratch, or use an off-the-shelf database solution. Building a database framework from scratch that is compatible with the future growth of the application is not a trivial task. Years of work go into the development of a stable, high-performance database platform with a footprint small enough to be suitable for embedded devices.

Selecting an available off-the-shelf database is a good alternative that allows

developers to focus on application business logic and leave data management to solutions available in the market. Embedded databases are a unique alternative that is invisible to the end user, linked directly into the application as an in-process software library, and requires almost no database administration. This provides the application all the features of a relational database (RDBMS) without the overhead and complexity of a traditional database management system. Zero database administration reduces the Total Cost of Application Ownership as the database is managed from within the application.

Alternative to Database: Flat Files

New software development is driven by the evolution of hardware. Embedded and mobile devices have gained significant storage capability in recent years, making new software development options possible. Previously, most devices stored little or no data persistently, only synchronizing when docked in a cradle. These devices typically stored data in flat files, using either a simple text format or binary format customized for the device's built-in applications.

Flat text files use a simple format that is human readable and loosely structured. While text files are easy to write and can be created in any text editor, every change, large or small, usually requires the entire file to be rewritten. This makes text files especially susceptible to data loss because the entire file is put at risk every time any data is modified. When a large text file is repeatedly erased and written to flash memory, the extra wear reduces the total lifetime of the flash media. Text files also cannot be very large because of the time required to write changes and because the entire file must be read into memory to efficiently search for data.

Binary flat files are more versatile because the strict structure of these files makes it possible to efficiently divide the file into smaller pieces that can be used independently. But binary files cannot be viewed in a text editor, so special tools must be developed to use these files outside the application. Binary files provide some protection against corruption of the entire file, but can still lose data when changes are only partially written before an unexpected power failure. Binary files also make it more difficult to store variable-width data.

An application can partially protect against corruption of a flat file by writing to a new file each time it is saved. The new file can then be renamed after erasing the old file. However, most file systems will cache the rename operation, effectively canceling the benefit of this approach.

Flat files also have a limited data management life cycle offering, as devices and embedded systems are becoming more aware of each other and other systems.

This awareness causes data growth and these systems require the capability to store, retrieve, manipulate, and query data efficiently. Flat files are useful for limited data storage but have limited scalability and durability.

Most of all, an elaborate flat file storage framework distracts engineering resources from application logic. Many developers effectively end up building a proprietary database management system that they do not have time to maintain or fully optimize.

Features of Database Technology

Database technology aids applications that store persistent data storage in three important ways: reliability, scalability, and shared access. Embedded databases accomplish this using features such as transaction logging, indexed search, and row-level locking.

Transaction logging prevents data corruption when a sudden power failure or crash occurs. Changes to a database file are grouped into transactions. Transactions are first written to a separate log file so that if the application is interrupted while writing changes to the storage device, it can recover the database to a known good state. Transactions are guaranteed to be atomic, meaning that each transaction will either finish completely or make no changes at all.

Relational embedded databases organize information into tables, large or small, that can be quickly searched using B+ tree or similar indexes. The table-driven data model used by relational databases facilitates interoperability and maintainability because it is a straightforward format that is compatible with many industry standards, such as SQL and ODBC. Developers can easily leverage experience gained from one relational database when working with another.

Databases can also support multiple connections to a database, allowing multiple tasks to be performed concurrently without interfering with each other. High-level concepts, such as transactions and isolation, simplify analysis of how tasks interact and eliminate the need for manual locking of data.

Relational databases provide these key features:

- High performance
- Main-memory and disk-based tables
- Shared access
- SQL
- ODBC
- ACID Transactions
- Multi-table joins
- Automatic locking
- BLOBs (Binary Large Objects)
- Unicode character support

Value of Embedded Database

Modern embedded devices are now responsible for storing more data than ever before. Some devices get an edge on the competition by synchronizing data without interrupting normal use. Important data must not be lost to corruption caused by a power failure. For these devices, performance and reliability are critical.

Power failure is an important issue on embedded devices. Many devices are battery-powered and are prone to sudden power failure. If persistent data is not managed carefully, unexpected power failure can result in lost data or even corruption, making all persistent data on the device vulnerable. Database uses transaction recovery to both protect against corruption and ensure that data is completely consistent after a power failure. Vendors cannot afford to bring a product to the market that crashes at a customer's site. For example, if a GPS vendor introduces a new product that fails after installation, the reputation of the vendor is at risk.

Application code is frequently re-used to expand an existing product line or to jump-start development of a new product. Using a database gives an application a consistent architecture for persistent data storage, making it easy to add new features and migrate the application code to a new environment. Database is a good long-term investment because it lets applications scale through the entire life cycle of the application.

For example, many devices are initially designed to store data locally and only synchronize with other systems as a dedicated task. If such an application is database-driven, it can use row-level locking to interact with other systems and share data without interrupting normal usage. Sending and receiving data and running custom queries is greatly simplified by the database API.

Embedded devices use a variety of processor architectures, and ARM is one of the most popular. To improve efficiency compared to other architectures, ARM processors require integers in memory to be aligned on a word boundary. ARM processors can also be configured to use either big-endian or little-endian byte order. Features like these improve the performance and flexibility of software written for ARM processors.

As a result, the memory layout of a data structure, such as a C struct, is determined by the processor architecture. So it is not possible to copy data directly from memory on one processor architecture to memory on another processor architecture. Instead, the processor architecture must be recognized so that data can be transformed into an architecture-independent format if it is to be shared between devices and other computer systems.

If data is copied directly from memory to a binary flat file, it is not easy to open the

file on another architecture, such as the x86 architecture used by desktop workstations. An embedded database that understands the structure of data can automatically perform the necessary conversions when a database is shared between devices, either by copying the database file itself or using network communications to access the database.

Many devices, such as mobile inventory management devices, are used to manipulate a subset of data from a large back-end database server. An embedded database on the device allows this data to be saved reliably in a format that is easily shared with a back-end server.

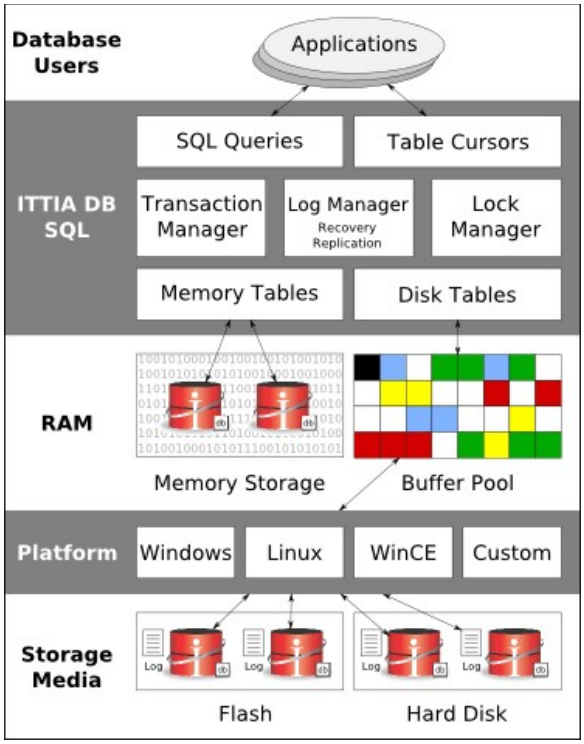
An out-of-the box database solution also reduces total cost of ownership and provides a shorter time to market because the developers can focus on core application development.

Features like these have been available for decades in enterprise RDBMS (relational database management system) products, but the footprint requirements and cost of these monolithic systems make them impractical for embedded systems and devices. Enterprise databases are designed to support access from thousands of users at once and balance load across multiple server machines, requiring sophisticated management and administration. These products also include many deprecated features required to support decades of legacy applications and are optimized for servers with gigabytes of RAM and large hard disk arrays.

Embedded databases are designed to run in resource-limited environments. The database can be embedded directly in the application, completely hiding the database from the end-user and greatly simplifying management and maintenance. With a code footprint less than a megabyte, an embedded database can easily fit on an embedded device and can even be customized to satisfy strict footprint requirements. Embedded databases use algorithms that are efficient for small-scale applications, but that easily scale to store a large amount of data.

An Embedded Database Inspired by Enterprise Functionality

Just as enterprise developers once recognized a need for formal data management on mainframe servers, embedded/device developers are discovering new requirements that demand a robust RDBMS for local data management. As a result, these developers have begun searching for a data-driven storage engine. Many of these software developers require their applications to locally store, manipulate, and retrieve data, as their systems can no longer afford to be disconnected.



Many embedded database solutions started as a consulting project for a single application that was then repackaged for use in other applications. ITTIA DB SQL was developed from the ground up to recognize and solve issues with data management on modern embedded systems and devices.

Before developing ITTIA DB SQL, engineers at ITTIA interviewed embedded and enterprise development experts to identify their common requirements for data management. From this research, ITTIA learned that the most important issues facing embedded developers are compatibility, due to the diversity of hardware and software environments available, and small footprint. To ensure broad compatibility, ITTIA DB SQL was designed as cross-platform RDBMS using the standard relational data model. To accommodate footprint restrictions, careful attention was paid to algorithm selection and memory utilization.

ITTIA DB SQL stores data in a portable format that can be accessed directly using table cursors or with SQL queries. A database can be stored in a local file, or shared across a network. ITTIA DB SQL is also modular, allowing features such as shared access and even the SQL engine to be omitted to reduce footprint. Many embedded applications do not need the overhead of these features and can save space by leaving them out.

An application's storage requirements often change over time, requiring updates to the database schema. Dynamic schema alteration, supported by ITTIA DB SQL, allows applications to add and remove tables and columns to a live database. And

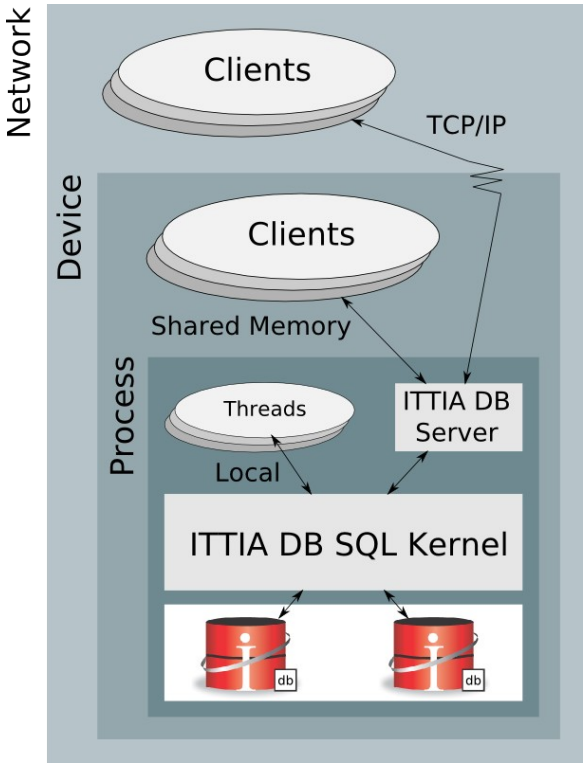
because the schema can be inspected at runtime, the application can use the new schema without being recompiled.

For best performance and minimal wear on storage media, ITTIA DB SQL uses a buffer pool to control memory usage and prevent fragmentation. This accountability for memory utilization is critical in embedded systems.

Available Solutions

Some database solutions provide shared access to a database by locking the entire database file. File system locks operate on an entire file at once, so when a transaction begins to modify the database, it must obtain exclusive access to the entire database file until the transaction is finished. This is not a problem when sharing is infrequent and every transaction only involves one or a few rows. However, one long-running transaction, such as a synchronization task, can block all other activity in the database, even when there is no real conflict.

ITTIA DB SQL can use either storage-level locking, or a less restrictive locking technique: row-level locking with isolation levels. The database automatically tracks all rows that are read or modified in a transaction. At the highest level of isolation, known as "serializable," rows are locked in such a way as to prevent all possible conflicts. And for most simple transactions, the isolation level can be reduced to minimize locking even further. This ensures that a transaction is only blocked when it would create a conflict with another transaction already in progress. In addition, an entire table can be locked manually.



Applications share access to ITTIA DB SQL databases by opening a separate connection for each task. Within a single process, each connection can access the database file directly. Multiple processes can share a database file by connecting to a server process through shared memory or TCP/IP. Regardless of how many connections there are or how they are made, each database file uses only one page cache.

Some databases use dynamic run-time typing, which means that the type of a

value is effectively only checked when it is read from the database and used. This flexibility is useful in prototyping, before the application's requirements are fully formed, because it allows data to be written to the database without much regard for how it will be used later. Production code, however, must be carefully audited to ensure that type mismatches are not possible or can be dealt with in a reasonable way. Dynamic typing works well with languages such as PHP that support manifest typing natively.

ITTIA DB SQL uses static typing, where type information is stored in the database schema as part of a table's description. Each column can contain only a specific type of data. This ensures that type mismatch errors are identified early, when there is the best chance to successfully fix the mistake. This is important when the database is shared between applications that are developed separately, as the database schema forms a contract by which all parties must abide.

ITTIA DB SQL exposes a C API that supports SQL queries, but it also provides a framework for direct table access. Many operations, such as inserting rows with dynamic values, are easier to express and have better performance when direct table access is used. ITTIA DB SQL also includes an object-oriented C++ API.

The terminology used by ITTIA DB, both in the API and documentation, should be familiar to enterprise database developers. Some embedded database products use unusual terms for common concepts, which forces developers to learn a new way of talking about databases.

Database files are divided into pages, which are the basic unit of I/O. Related data is often clustered together on one page, especially in indexes, which boosts performance by minimizing I/O operations. Recovery logging is greatly simplified by aligning the page size to the size of disk blocks.

For most write transactions, ITTIA DB will only append to the recovery log, without updating the database file itself. On disk media, this greatly improves performance because the head does not need to move. On flash media, this avoids unnecessary erasure, which both takes time and wears the storage media. Because ITTIA DB supports undo/redo logging, it can achieve this result without imposing a limit on transaction size. Databases that use only undo logging must always write to the database itself after each transaction, and database that use only redo logging must limit transactions to the available cache size.

Final Thoughts

Embedded database gives software developers the edge they need to differentiate themselves from the competition, deploy sooner, and cut costs. Every application that manipulates information must face at least one of the problems addressed by

database: reliable persistent storage, high-performance search, and safe data sharing. An embedded database library, like ITTIA DB SQL, is an out-of-the-box solution to all of these common data storage problems.

Database simplifies application development. Rather than building separate solutions for each of these problems, the application developer uses a standard framework for accessing and modifying data. Furthermore, reconciling these features with each other in a single application is not trivial and can become a great distraction to developing the business logic of the device.

ITTIA stands behind its database technology with guaranteed support through all phases of product development. ITTIA's experts provide indispensable advice during planning and design that gets development started in the right direction from the very beginning. ITTIA's training programs give developers an edge up with a thorough introduction to the capabilities of embedded database and best development practices. In application development, ITTIA supports customers with direct problem tracking and resolution. Support even extends to the deployment of the device and application, ensuring a smooth delivery and market adoption.

By cutting out a large amount of software development work, an application using an embedded database can be developed faster and at a lower cost compared to the same application without a database library. Software developers do not need to sacrifice performance in critical algorithms, because tables and indexes can be accessed directly through the C API. And with the database supporting standard features like SQL, the application can offer features that would otherwise be impractical to develop. Database makes it possible to rapidly develop feature-rich applications that will be able to scale to meet the demands of the future.

About ITTIA

ITTIA provides software and services for data management, offering standards, ease of use, and flexibility to our customers. Benefits of selecting ITTIA's technologies include leading-edge software, comprehensive documentation, scalability, efficiency, exceptional performance, and low total cost of ownership. Learn how customers such as Freescale Semiconductor, Panasonic, Puget Sound Energy, Fresenius, Boeing, and others have valued from ITTIA by visiting:

www.ittia.com

How to Reach Us

Home Page:
www.ittia.com

e-mail:
support@ittia.com

USA:
ITTIA
1611 116th Ave
Bellevue, WA 98004
425-462-0046