Building durability into data management for real-time systems

by **Andrei Gorine,** McObject

SEVERAL STRATEGIES HAVE EMERGED FOR ALL-IN-MEMORY DATA STORES TO ACHIEVE HIGH AVAILABILITY AND DURABILITY OF DATA. THIS ARTICLE DISCUSSES DIFFERENT POSSIBILITIES TO REACH THIS GOAL.

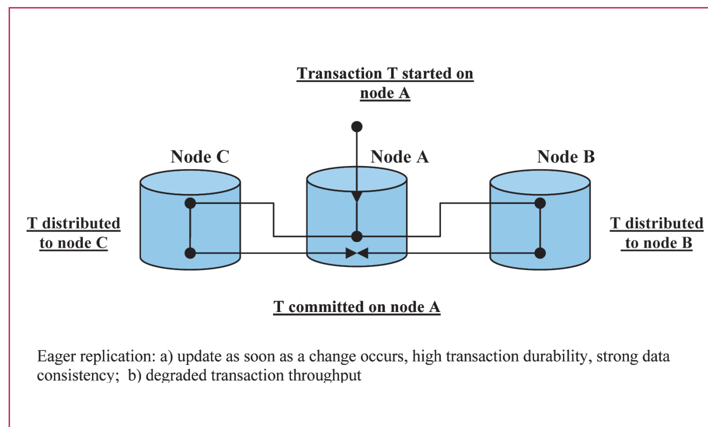

Figure 1. Eager (synchronous) replication

Main-memory database management systems are widely used in embedded systems, providing high performance and predictability often required in real-time settings. Several strategies have emerged for these all-in-memory data stores to achieve high availability and durability of data. Options include support for NVRAM databases, on-line backup, transaction logging and database replication. With these tools at their disposal, embedded systems engineers can achieve the identical degree of data protection as can be obtained when using traditional disk-based products. Just as importantly, these approaches enable the developers to adjust the level of durability to achieve the desired throughput and application performance. Embedded systems designers can choose the appropriate option based on the specific needs of their applications.

In recent years, deployment of real-time and embedded systems has dramatically increased. At the same time, as a result of demand for enhanced features and capabilities, such systems are managing ever greater volumes of more complex structured data. Developers are recognizing the need for database management systems to support storage functionality in embedded devices. To reduce time-to-market and enhance scalability and reliability, developers increasingly turn to commercially available, off-the-shelf data management software to replace “homegrown” data stores.

The nature of real-time processing places unique demands on databases. Real-time databases handle transactions with temporal constraints and also maintain the logical consistency of data. In contrast to desktop and enterprise applications, data processed too late in a real-time environment often becomes incorrect or even dangerous, so it is vital for the actual state of the external world and the state represented in the database to be close enough to remain within the limits of the application. For example, in a nuclear power plant, a database embedded in an automated controller will collect and process data received from hundreds of sensors throughout the plant. If the data (such as the exterior radiation levels or the reactor temperature) is not processed within a certain time limit, the consequences could be disastrous. Furthermore, in addition to providing predictability and high performance, embedded systems’ data stores need to run without human intervention - they must be able to recover from failure automatically and continue providing data access.

The real-time performance and predictability needs of these systems often dictate the use of in-memory database systems - that is, databases that execute entirely in memory, never going to disk. Eliminating mechanical disk access cuts an important source of overhead and, by forgoing disk access-related logic such as caching, in-memory execution makes possible a streamlined architecture that utilizes CPU cycles more efficiently. By avoiding latencies often associated with disk I/O, main-memory databases can also provide predictable response time to guarantee the completion of time-critical transactions. However, by its nature, a database that operates entirely in RAM is vulnerable - if the RAM content is destroyed, the database is destroyed with it. How can a memory-only database achieve durability, or the capacity to survive failure of the hardware device on which it resides or the software environment in which it operates?

To achieve durability, in-memory database systems can offer several solutions. An in-memory data store often offers a means of maintaining copies of data, so the loss of RAM and its content does not mean loss of data access and the data itself. In this solution – called database replication – fail-over procedures allow the system to continue using a standby database. Other solutions involve using a non-volatile media that helps recover the data in the case of system failure. These different approaches - including synchronous and asynchronous replication, transaction logging and the use of a non-volatile memory device - each have their own performance and resource utilization im-

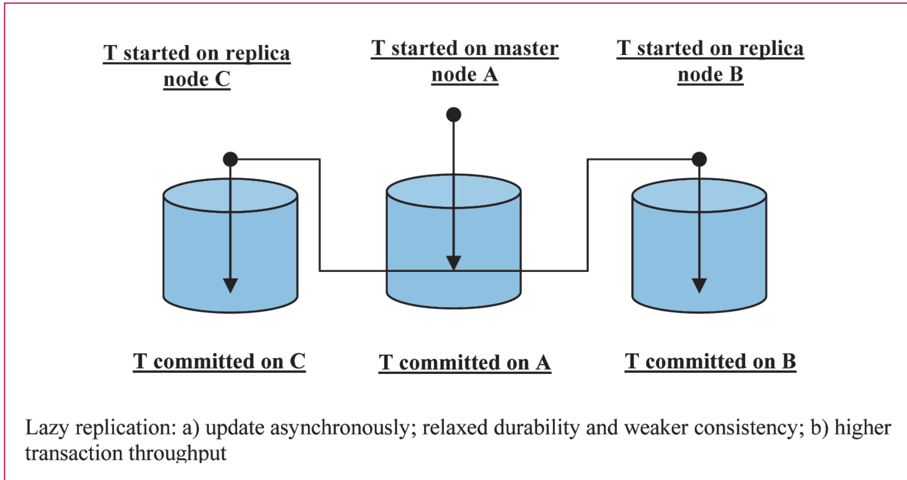

Figure 2. Lazy (asynchronous) replication

plications, which must be understood by the developer and taken into account when designing real-time data management.

A fundamental concept is that within virtually any application, data management is carried out via basic units of processing called transactions. A transaction is a sequence of read and write operations on data and must offer the following: atomicity, consistency, isolation, and durability, called ACID properties. In particular, transaction durability means that once a transaction is committed successfully, all the changes to the database are permanent, and must survive subsequent system failure or malfunctions. Thus, when discussing the durability of databases, we will refer to transaction durability and the ways database management systems maintain it.

Maintaining replicated copies of data is an effective way to achieve database transaction durability. A replicated database consists of failure-independent nodes. The database system manages the data distributed over multiple nodes, making sure data is not lost even in the case of a node failure. Traditional approaches to managing replicated databases can be classified as eager (synchronous) or lazy (asynchronous) based on the strategy used to propagate updates to replicas. In eager replication, all replicas are kept synchronized by applying updates to all replicas as part of the original transaction. While ensuring transaction durability during a replica takeover in the event of a node failure, eager replication might exhibit longer resource holding time and, consequently, degraded transaction throughput due to network delays.

In contrast, lazy replication commits a transaction without waiting for the updates to propagate to the replicas, resulting in shorter resource holding time. However, lazy replication risks a potential loss of committed transactions, if a replica must take over processing in case of failure at the transaction origination site before the updates reach the replicas.

Generally speaking, in lazy replication algorithms, the transaction durability is not as high. A better choice for an application requiring predictable response times may be data management using time-cognizant eager replication protocols to add time semantics to eager replication algorithms. Embedded systems frequently impose strict processing deadlines, and such an approach ensures on-time delivery of the transaction data between replicated nodes. As with any active replication scheme, fail-over procedures ensure transaction durability, but with time-cognizant eager replication, the predictability of transactions is enforced as well.

When non-volatile media such as a hard disk or a flash card is present in the embedded system, the in-memory database can be made persistent on this media. If the system fails and is restarted, the content of the database can be restored from the disk via the recovery proce-

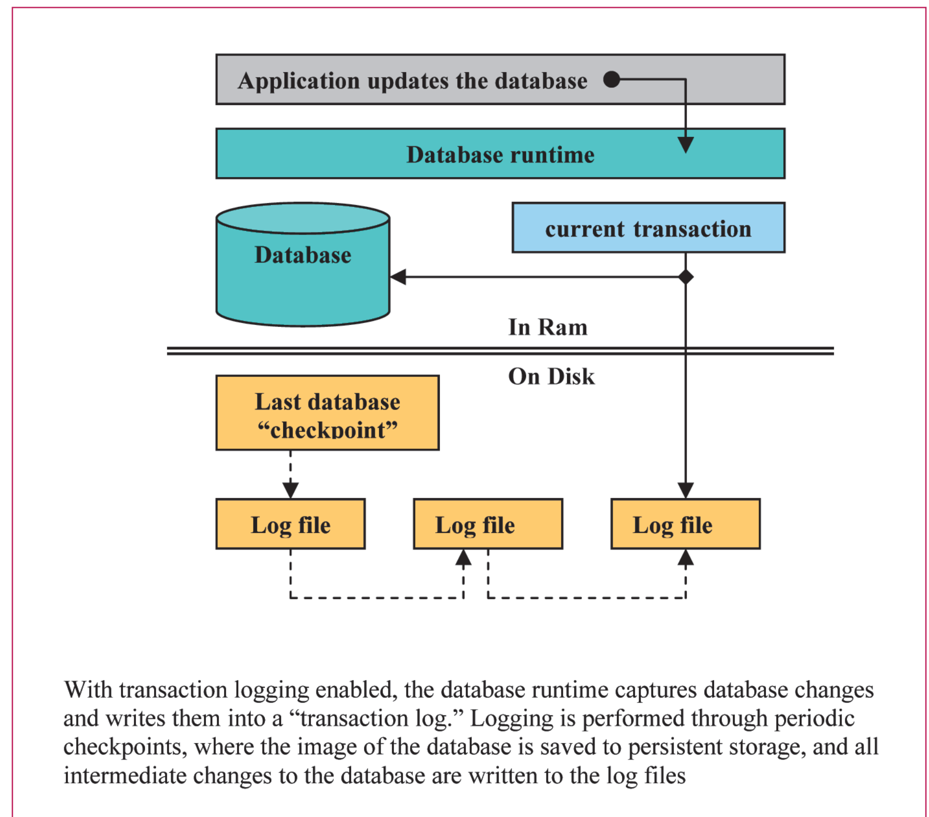

Figure 3. Transaction logging

dures provided by the DBMS. There are basically two different strategies for accomplishing this: roll-back recovery and roll-forward recovery. With either approach, during normal operation, periodic snapshots of the in-memory database (also called "checkpoints") are taken and stored on non-volatile media. During the roll-back recovery, upon system restart, the content of the last checkpoint is simply loaded into memory. All database changes performed after the last checkpoint are therefore lost. In roll-forward recovery, checkpoints are also stored regularly, but all intermediate transactions are written to a transaction log. For recovery, the log changes are applied chronologically from the last checkpoint, in the same order as they were entered into the log.

Doesn't this recording of database information to persistent storage introduce exactly the disk access overhead that IMDBs seek to eliminate? With most commercially available main memory database products, such as McObject's eXtremeDB, transaction logging may be set to different levels of transaction durability, allowing system designers to make intelligent trade-offs between performance and risk for lost transactions. For example, the transaction log can be written to persistent storage either synchronously or asynchronously. With the synchronous approach, transaction data is written to disk within the transaction. In this case, all transactions are logged, but the database is locked during this process, which can cause unpredictable delays during the real-time process. With asynchronous logging, writing log data is scheduled to occur when the overall system load is low, or at some other time of the application's choosing, or even by a separate process, in order to reduce performance overhead that could affect the main application process. In asynchronous logging, transaction durability is relaxed, but the overall database responsiveness is higher.

Generally, transaction logging does not alter the all-in-memory architecture of the IMDB, and the database retains a performance advantage over disk-based databases. Read performance of IMDBs is unaffected by transaction logging, and database write performance will far exceed that of traditional disk-based databases. The reason is simple: transaction logging requires exactly one write to the file system for one database transaction, while a disk-based database will perform many writes per transaction for data pages, index pages, transaction log, etc.(the larger the transaction and the more indexes that are modified, the more writes that will be necessary).

Today, many embedded devices feature NVRAM, a non-volatile memory whose content is saved when a computer is turned off or loses its external power source. NVRAM usually takes the form of static RAM with backup battery power, or an electrically erasable programmable ROM (EEPROM) used to save data. If NVRAM is used to store the database, the database management system can recover the data store from the last consistent state upon reboot. This approach presents a very attractive durability option for an in-memory database. In contrast to the transaction logging approach involving disk I/O related overhead, or to the replication approach with its communication overhead, an NVRAM database incurs no disk or network overhead during operation. Commercial databases rarely provide direct support for NVRAM data stores. This is primarily because high-capacity NVRAM devices capable of holding a database are rather expensive. On some occasions, though, the expense can be justified. For example, high-end routers could keep their configuration databases on NVRAM boards, and network storage devices could implement some of their architecture on NVRAM. For such applications the NVRAM database can provide invaluable performance advantages over other database durability options and therefore justify the incremental expense of NVRAM. ■