

hamsterdb Transactional Storage:

A Technical Overview

July 2009

This paper provides a technical overview of hamsterdb Transactional Storage.

From [Wikipedia](#): **ACID** (*Atomicity, Consistency, Isolation, Durability*) is a set of properties that guarantee that database transactions are processed reliably. In the context of databases, a single logical operation on the data is called a transaction. An example of a transaction is a transfer of funds from one bank account to another, even though it might consist of multiple individual operations (such as debiting one account and crediting another).

Purpose

Modern data-driven applications running on today's servers or commodity PCs require data management facilities which are fast, concurrent and reliable. While most Database engines are still based on 30 year old algorithms, hamsterdb Transactional Storage is tailored to make the best use of today's hardware and to fulfill the requirements of today's applications.

Hamsterdb Transactional Storage is a reliable, ACID compliant and concurrent key/value Database engine. It is directly embedded in the application, resulting in fast performance and zero administration costs. This document provides a technical overview of hamsterdb Transactional Storage.

ACID Compliant Transactions

hamsterdb Transactional Storage provides full support for ACID compliant transactions to enable the development of e-commerce and other robust, business-critical applications.

In hamsterdb Transactional Storage, all Transactions are stored in RAM memory. They are lock-free (see below), which means that an active or committed Transaction does not hold any lock during its life time. If two Transactions try to modify the same key, a conflict is discovered immediately and reported to the application. When a Transaction is committed, a background thread flushes the Transaction to the disk. By moving expensive disk I/O to the background, high performance is guaranteed.

Since RAM memory is volatile, hamsterdb Transactional Storage writes logical log files with each Database operation. Since these log files only store information about the operation itself, but not about the physical file modifications, the files are very small and do not impose the typical performance penalty of a physical log file.

When Transactions are flushed to disk, they are written to B+Tree structures. All B+Trees of each index is managed in a single file. To avoid the need of another (physical) log file for this operation, the B+Tree algorithms have been carefully modified to be atomic, and therefore never leave the file in an inconsistent state after a system crash or surprise shutdown.

Lock-Free Architecture

In hamsterdb Transactional Storage, when a Transaction modifies a key/value pair, it never directly modifies the key/value pair on disk (in the B+Tree index). Instead, the key/value pair is stored in the Transaction itself. Each Transaction manages its own list of modified key/value pairs. To allow quick lookups, these pairs are stored in a sorted in-memory tree structure (a simplified Red Black Tree). No B+Tree buffer pages are locked during the lifetime of a Transaction. Actually, no locks are held at all.

As soon as a new Database operation starts, all Transactions are checked for possible conflicts (a conflict occurs if another active Transaction is modifying the same Database key). If a conflict is discovered, the operation returns with an error. Otherwise, the key/value pair and some additional information about this operation is stored in the Red Black-tree of this Transaction.

As soon as a Transaction is committed, a background thread flushes it to the B+Tree, and then removes the Transaction's in-memory structures.

Currently, only one Transaction isolation level is supported: "read committed". Other isolation levels will follow soon.

Logging and Recovery

The architecture of hamsterdb Transactional Storage has two potential points of failures. The first one are the in-memory structures of the Transactions. The other one is the flushing of a committed Transaction to the persistent file.

In the first case, the application which links hamsterdb Transactional Storage crashes and the in-memory Transaction structures are lost. Since even committed Transactions can remain in memory, this would violate the *Durability* of the ACID principle. Therefore hamsterdb Transactional Storage writes a logical log file. For each write operation, an entry describing this operation is appended to the log file. Groups of 20 Transactions share the same log file. If all those Transactions are either aborted or committed and flushed to disk, the log file is deleted.

hamsterdb Transactional Storage: A Technical Overview

In case of a system crash or unclean shutdown, the Transactions are not flushed and therefore the log files are not deleted. When hamsterdb Transactional Storage restarts, it finds remaining log files and therefore triggers the recovery process. During recovery, the log files are read and all committed (and, if desired, even uncommitted) Transactions are recreated. The hamsterdb Transactional Storage API also provides facilities to enumerate all uncommitted Transactions so the application can continue to work with them.

The recovery always starts with the oldest log file and re-applies all operations described in these log files. It is therefore possible that some operations are applied multiple times. To avoid this, the recovery is "idempotent". By storing the LSN (Log Serial Number) of each Database update in the modified Database pages, hamsterdb Transactional Storage can discover if this update had already been applied.

In the second case, the application or system crashes while a flush is in process. Since B+Tree structure modification operations can be complex and sometimes are composed of several disk accesses, a system crash can lead to a corrupt file. Instead of writing another, physical log file, hamsterdb Transactional Storage uses a modified B+Tree algorithm for those cases (i.e. a page split) where atomic operations are not possible per se. These operations store recovery information in the B+Tree pages during the page split. In case of a crash, hamsterdb Transactional Storage will recover and re-apply all the operations which are still in the log file. During this recovery, the corrupt page will be discovered and the recovery information in the page will be used to fix the page.

Not using a physical log file results in faster performance by avoiding expensive disk I/O and limits the file size of the Database.

Concurrency Support

hamsterdb Transactional Storage is not just thread safe. It uses a second thread to process expensive operations in the background. And it allows use of every API handle in arbitrary threads.

Internally, hamsterdb Transactional Storage uses mutexes and atomic operations (called "latches" in Database terminology) to protect its internal structures against concurrent access. API users therefore do not have to add any synchronization mechanism. Also,

hamsterdb Transactional Storage: A Technical Overview

most of the Database operations are in-memory only, therefore the latches are hold for a very short time and usually are not-exclusive.

In combination with the lock-free architecture, the hamsterdb Transactional Storage API user can create an arbitrary amount of (even long-running) Transactions in many different threads without having any worries about synchronization.

Easy to Use

hamsterdb Embedded Storage is simple to use - its API is straight forward and reportedly less complicated than other competing Databases. With hamsterdb Transactional Storage, some things are now even more simplified. The move to a handle based API (instead of pointers to structures) makes access to uninitialized or already freed structures impossible. Function names, parameter names and constants are never cryptic but self-explanatory. Extensive API documentation (in doxygen format) and samples allow for a smooth learning curve with fast results.

The API of hamsterdb Transactional Storage is not compatible with the API of hamsterdb Embedded Storage. The handle based API, the extensive use of Transactions and the mandatory use of an Environment (which is a collection of multiple Databases) are new. But the concepts are very similar, and migration is therefore straight forward.

Through the Looking Glass

As of today (July 2009), the first beta release of hamsterdb Transactional Storage is available for download. This release was created in over one year of development time. During the next months, many features will be added and many gaps will be closed. Among them are:

- 1 Reclaim erased file space - if a record or B+Tree page is erased, it is not yet reused
- 2 Various performance improvements
- 3 In-Memory Database/Environments
- 4 Record Number Databases ("Auto-Increment Databases")
- 5 Database Cursors
- 6 More Transaction isolation levels (Read Uncommitted, Serializable...)
- 7 Duplicate Keys for mapping n:m relations
- 8 APIs for C++, Python, .NET, Java and other languages
- 9 and much much more...

About hamsterdb Transactional Storage

hamsterdb Transactional Storage is a dual-licensed open source project. For more information, please visit <http://hamsterdb.com> or send an email to info@hamsterdb.com.

© Christoph Rupp, 2009