IBM solidDB: In-Memory Database Optimized for Extreme Speed and Availability

Jan Lindström, Vilho Raatikka, Jarmo Ruuth, Petri Soini, Katriina Vakkila
IBM Helsinki Lab, Oy IBM Finland Ab
jplindst@gmail.com, raatikka@iki.fi, Jarmo.Ruuth@fi.ibm.com, Petri.Soini@fi.ibm.com, vakkila@fi.ibm.com

Abstract

A relational in-memory database, IBM solidDB is used worldwide for its ability to deliver extreme speed and availability. As the name implies, an in-memory database resides entirely in main memory rather than on disk, making data access an order of several magnitudes faster than with conventional, disk-based databases. Part of that leap is due to the fact that RAM simply provides faster data access than hard disk drives. But solidDB also has data structures and access methods specifically designed for storing, searching, and processing data in main memory. As a result, it outperforms ordinary disk-based databases even when the latter have data fully cached in memory. Some databases deliver low latency but cannot handle large numbers of transactions or concurrent sessions. IBM solidDB provides throughput measured in the range of hundreds of thousands to million of transactions per second while consistently achieving response times (or latency) measured in microseconds. This article explores the structural differences between in-memory and disk-based databases, and how solidDB works to deliver extreme speed.

1 Introduction

IBM solidDB [1, 18] is a relational database server that combines the high performance of in-memory tables with the nearly unlimited capacity of disk-based tables. Pure in-memory databases are fast but strictly limited by the size of memory. Pure disk-based databases allow nearly unlimited amounts of storage but their performance is dominated by disk access. Even if the server has enough memory to store the entire database in memory, database servers designed for disk-based tables can be slow because data structures that are optimal for disk-based tables [13] are far from optimal for in-memory tables.

The solidDB solution is to provide a single hybrid database server [5] that contains two optimized engines:

- The disk-based engine (DBE) is optimized for disk-based access.
- The main-memory engine (MME) is optimized for in-memory access.

Copyright 2013 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Both engines coexist inside the same server process and a single SQL statement may access data from both engines. The key components of solidDB MME technology (Vtrie and index concurrency control) are discussed in more detail in section 2.

To take the full advantage of the in-memory technology, you can also link your client application directly to the database server routines for higher performance and tighter control over the server. The shared memory access (SMA) feature of solidDB uses direct function calls to the solidDB server code. The in-memory database is located in a shared memory segment available to all applications connected to the server via SMA. Multiple concurrent applications running in separate processes can utilize this connection option to reduce response times significantly. This means that the applications ODBC or JDBC requests are processed almost fully in the application process space, with no need for a context switch among processes.

In addition to a fully functional relational database server, solidDB provides synchronization features between multiple solidDB instances or between solidDB and other enterprise data servers. The solidDB server can also be configured for high availability. Synchronization between solidDB instances uses statement-based replication and supports multi-master topologies.

The same protocol extends to synchronization between solidDB and mobile devices running the IBM Mobile Database. For synchronization between solidDB and other data servers, log-based replication is used. The solidDB database can cache data from a back-end database, for example, to improve application performance using local in-memory processing or to save resources on a potentially very expensive back-end database server. Caching functionality supports back-end databases from multiple vendors including IBM, Oracle and Microsoft.

The Hot-Standby functionality in solidDB combines with the in-memory technology to provide durability with high performance. This is discussed in more detail is section 3.

2 Makings of solidDB in-memory technology

Database operations on solidDB in-memory tables can be extremely fast because the storage and index structures are optimized for in-memory operation. The in-memory engine can reduce the number of instructions needed to access the data. For example, the solidDB in-memory engine does not implement page-oriented indexes or data structures that would introduce inherent overhead for in-page processing.

In-memory databases typically forgo the use of large-block indexes, sometimes called bushy trees, in favor of slimmer structures (tall trees) where the number of index levels is increased and the index node size is kept to a minimum to avoid costly in-node processing. IBM solidDB uses an index called trie [4] that was originally created for text searching but is well suited for in-memory indexing. The Vtrie and index concurrency control designs are discussed in more detail in sections 2.1 and 2.3.

The solidDB in-memory technology is further enhanced by checkpoint execution. A checkpoint is a persistent image of the whole database, allowing the database to be recovered after a system crash or other cases of down time. IBM solidDB creates a snapshot-consistent checkpoint that is alone sufficient to recover the database to a consistent state that existed at some point in the past.

2.1 Vtrie indexes

The basic index structure in the in-memory engine is a Vtrie (variable length trie) that is optimized variation of path- and level-compressed trie. Vtrie is built on top of leaf node level very much similar to those used in B+-trees. Key values are encoded in such a way that comparison of corresponding key parts can use bytewise comparison and end of keypart evaluates smaller than any byte value. Each Vtrie node can point to maximum 257 child nodes, one for each byte value plus “end of keypart”. Vtrie index stores the lowest key value of each leaf node, which guarantees the Vtrie index will not occupy excessive space even if the key value distribution would cause low branching factor. The worst case branching factor for Vtrie is 2 which occurs when key values

are strings consisting of only two possible byte values. The B+-tree -like leaf level guarantees the Vtrie still has far fewer nodes than there are stored key values in the index.

There are two types of key values stored in the index: routing and row keys. Routing keys are stored in Vtrie nodes, while row keys are stored at the leaf level, which consists of an ordered list of leaf nodes. Every leaf node is pointed to by one routing key stored in the Vtrie. The value of the routing key is higher than the high key of the node's left sibling and at most the same as the low key of the leaf node it points to.

Leaf node size varies depending on the keys stored to it but the default size is a few cache lines. Unlike B-, and binary trees Vtrie does not execute any comparisons during tree traversal. Each part of a key is applied as an array index to a pointer array of a child node. Contrary to a value comparison, array lookup is a fast operation if the array is cached in processor caches. When individual array index is sparsely populated, it is compressed to avoid unnecessary cache misses. Finally, on the leaf level, row key lookup is performed by scanning prefix-compressed keys in cache-aligned leaf node.

2.2 Main-memory checkpointing

A checkpoint is a persistent image of the whole database, allowing the database to be recovered after a system crash or other case of down time. IBM solidDB executes a snapshot-consistent checkpoint [10] that is alone sufficient to recover the database to a consistent state that existed at some point in the past. Other database products do not normally allow this; the transaction log files must be used to reconstruct a consistent state. However, solidDB allows transaction logging to be turned off, if desired. The solidDB solution is memory-friendly because of its ability to allocate row images and row shadow images (different versions of the same row) without using inefficient block structures. Only those images that correspond to a consistent snapshot are written to the checkpoint file, and the row shadows allow the currently executing transactions to run unrestricted during checkpointing.

2.3 Main-memory index concurrency control

The in-memory engine's indexing scheme [12] allows any number of readers to simultaneously traverse the index structure without locking conflicts. A read request on an index uses an optimistic scheme that relies on version numbers in index nodes to protect the integrity of the search path. The read-only traverse algorithm is presented on Figure 1.

```
retry:
  parent = root; current = root; parent_version = parent.version;
  traverse_ongoing = TRUE
  while traverse_ongoing loop
    current_version = current.version;
    if parent.version_mismatch(parent_version) then
      goto retry;
    child = current.child_by_key(key)
    if current.version_mismatch(current_version) then
      goto retry;
    if child.terminal() then
      result = child; traverse_ongoing = FALSE;
    else
      parent_version = current_version; parent = current;
    end if
  end loop
  return result;
```

Figure 1: Index concurrency control algorithm.

A write operation on an index uses a two-level locking scheme to protect the operation against other write requests.

To support the lock-free read-only traversal scheme, when a writer changes any index node it first increments the node version number (to an odd value). After the consistency of the path being modified is achieved again the (odd) version numbers are incremented again (to even). The method version_mismatch() above returns TRUE if either of these apply:

1. node.version <> version argument OR
2. version is an odd value.

To help a read request to complete the search even if the search path is being heavily updated, a hot-spot protection scheme is also implemented. A search that needs to run a retry several times due to a version conflict falls back to locking mode that performs the traversal as if it were a write request. In practice this is very seldom needed.

The above methods still needs some refinement: When a ”child” pointer is dereferenced, two issues arise.

1. Can we safely even read the version number dereferencing the pointer?
2. Even if the pointer value is still a legal memory reference, how do we know that the version number field mismatches in case the memory location has been freed and then reallocated for some other purpose?

Both those issues would certainly be problems if freeing of an unused tree node would be a direct call to the allocator's free()-function. To resolve this issue, solidDB uses a memory manager for versioned memory objects [11] that guarantees the version number field is neither overwritten by an arbitrary value nor decremented. The engine has a ”purge level” which is a deallocation counter value before the oldest index traversal started. When an index traversal ends, it may increase the purge level. When the purge level increases, freed nodes can be released to underlying memory manager. This ensures that the version-checking traversal remains safe even when index nodes are freed. Recycling of memory to new versioned memory objects, i.e. index tree nodes, is still possible, because unpurged freed nodes can be used as new versioned memory objects.

3 High Availability Through the Use of Hot-Standby Replication

IBM solidDB provides high availability through automated error detection and fast fail over functionality in two-node Hot-Standby cluster (HSB, for short). HSB can be efficiently combined with in-memory engine because replicating REDO logs provides failure resiliency without having to wait a single disk access before commit. Full durability is achieved by means of asynchronous logging. Prior to active HSB replication the nodes, namely Primary (master) and Secondary (standby, slave) establish a connection which guarantees that database in nodes are in sync. Connection is maintained during normal operation and monitored by heartbeat mechanism. When connected, Primary accepts write transactions, generates row-level redo log records of changes and sends them to Secondary. Secondary secures the consistency of standby database by first acquiring long-term locks for rows before re-executing log operations. As a consequence, conflicting updates [16] are serialized due the use of locks, but non-conflicting ones benefit from parallelized, multi-threaded execution in multi-core environments.

In an HSB database, transaction logs are sent to the Secondary server by way of a replication protocol. In order to preserve the database consistency in the presence of fail over, the replication protocol is built very much on the same principles as physical log writing: the transaction order is preserved, and commit records denote committed transactions. If a fail over happens, the standby server performs a similar database recovery as if a transaction log was used: the uncommitted transactions are removed and the committed ones are queued for execution.

Synchronous (2Safe) replication protocols provide varying balance between safety and latency [16]. However, they all ensure that there is no single point of failure which would lose transactions. There are three 2Safe variables available. 2Safe Received commits as soon as Secondary acknowledges that is has received transaction log. 2Safe Visible and 2Safe Committed both commit when Secondary has executed and committed the transaction, but when 2Safe Visible is used, Secondary sends acknowledgment to the Primary prior to physical log writing. Unlike 2Safe protocols, asynchronous replication protocol (1Safe) prefers performance over safety by committing immediately without waiting for Secondary's response.

When solidDB server is used in Hot-Standby mode, it is state conscious: inside solidDB there is a non-deterministic finite-state automaton, which at any moment, unambiguously defines the server's capabilities. For example, whether the server can execute write transactions, solidDB provides an administrative API for querying and modifying HSB states.

solidDB also provides a High-Availability Controller (HAC, for short) software to automatically detect errors in HSB and recover from them. Every HSB node includes a solidDB HSB server, and a HAC instance. In addition to sub-second failure detection and fail over, HAC handles all single failures and several failure sequences without downtime. Network errors are detected by using an External Reference Entity (ERE) [15]. External Reference Entity which can be any device sharing the network with HSB nodes, which responds to ping command. When HSB connection gets broken, HAC first attempts to reach the ERE to determine the correct failure type. If ERE responds, HAC concludes that the other server has either failed or become isolated. Only after that the local HSB server can continue (if it was Master) or start (otherwise) to execute write transactions.

4 Performance and Conclusions

In solidDB, the main focus is on short response times that naturally results from the fact that the data is already in memory. Additional techniques are available to avoid any unnecessary latency in the local database usage, such as linking applications with the database server code by way of special drivers. By using those approaches, one can shorten the query response time to a fraction (one tenth or even less) of that available in a traditional database system. The improved response time also fuels high throughput. Nevertheless, techniques of improving the throughput of concurrent operations are applied too. The outcome is a unique combination of short response times and high throughput.

The advantages of solidDB in-memory database over a disk-based database are illustrated in Figure 2 where the response time in milliseconds of single primary key fetch and single row update based on the primary key is shown.

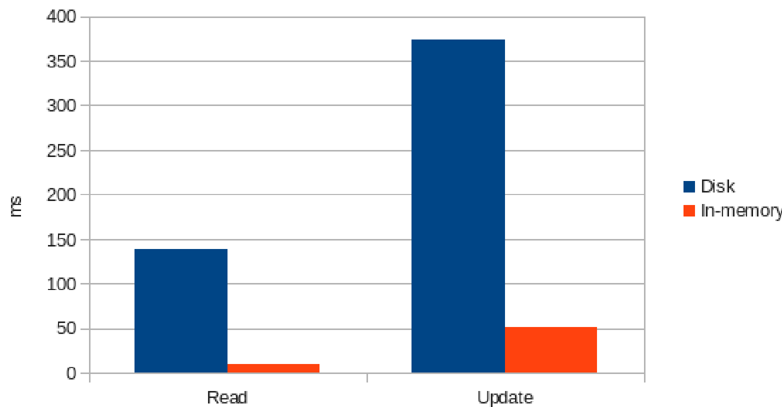

Figure 2: solidDB single operation response time.

Figure 3 shows the results of an experiment involving a database benchmark called Telecom Application Transaction Processing (TATP) [1] that was run on a middle-range system [2] platform. The TATP benchmark simulates a typical Home Location Register (HLR) database used by a mobile carrier. The HLR is an application mobile network operators use to store all relevant information about valid subscribers, including the mobile phone number, the services to which they have subscribed, access privileges, and the current location of the subscriber's handset. Every call to and from a mobile phone involves look ups against the HLRs of both parties, making it a perfect example of a demanding, high-throughput environment where the workloads are pertinent to all applications requiring extreme speed: telecommunications, financial services, gaming, event processing and alerting, reservation systems, and so on. The benchmark generates a flooding load on a database server. This means that the load is generated up to the maximum throughput point that the server can sustain. The load is composed of pre-defined transactions run against a specified target database.

The benchmark uses four tables and a set of seven transactions that may be combined in different mixes. The most typical mix is a combination of 80% or read transactions and 20% of modification transactions. The TATP benchmark has been used in industry [6] and research [7, 3, 8, 9, 14]. Experiment used database containing 1 million subscribers. The results of a TATP benchmark show the overall throughput of the system, measured as the Mean Qualified Throughput (MQTh) of the target database system, in transactions per second, over the seven transaction types. This experiment used shared memory access for clients and figure shows the scalability of the solidDB when number of concurrent clients are increased.

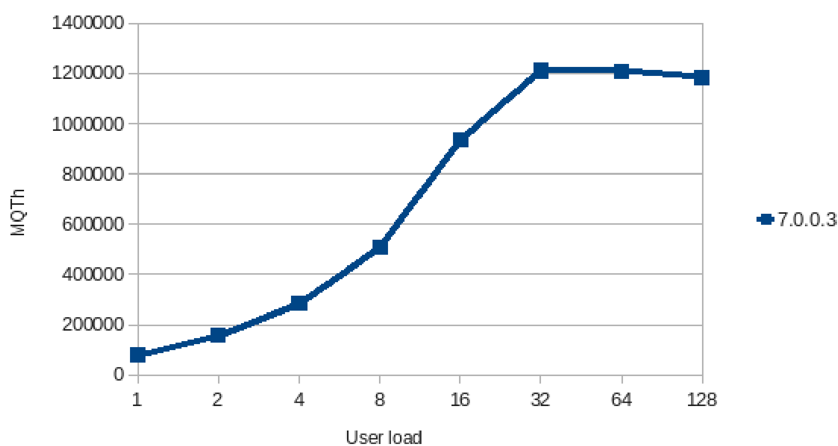

Figure 3: solidDB user load scalability.

In addition to telecom solutions, solidDB has shown its strength on various other business areas where predictable low-latency and high-throughput transactional data processing is a must, such as media delivery [2] and electronic trading platform.

References

[1] Chuck Ballard, Dan Behman, Asko Huumonen, Kyösti Laiho, Jan Lindström, Marko Milek, Michael Roche, John Seery, Katriina Vakkila, Jamie Watters and Antoni Wolski, *IBM solidDB: Delivering Data with Extreme Speed*, IBM RedBook, ISBN 07384355457, 2011.

[1] http:tatpbenchmark.sourceforge.net

[2] Hardware configuration: Sandy Bridge EP Processor: Intel Xeon E5-2680 (2.7 GHz, 2 socket, 8 cores16 threads per socket), Memory: 32GB memory (8 x 4GB DDR3 1333GHz), Storage: 4x 32GB Intel X25-E SSDs, OS: RHEL 6.1 (64-bit)

[2] *Fabrix Systems - Clustered Video Storage, Processing and Delivery Platform*, http://www-304.ibm.com/partnerworld/gsd/solutiondetails.do?solution=44455&expand=true&lc=en, IBM, 2013.

[3] Ru Fang, Hui-I Hsiao, Bin He, C. Mohan , and Yun Wang: *A Novel Design of Database Logging System using Storage Class Memory*. ICDE 2011.

[4] Edward Fredkin: *Trie Memory*, Communications of the ACM 3 (9): 1960.

[5] Joan Guisado-Gámez, Antoni Wolski, Calisto Zuzarte, Josep-Lluís Larriba-Pey, and Victor Muntés-Mulero, *Hybrid In-memory and On-disk Tables for Speeding-up Table Accesses*, DEXA 2010.

[6] *Intel and IBM Collaborate to Double In-Memory Database Performance*, Intel 2009 http://communities.intel.com/docs/DOC-2985,

[7] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki: *Data-Oriented Transaction Execution*. PVLDB, 3(1), 2010.

[8] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki: *Scalability of write-ahead logging on multicore and multisocket hardware.* VLDB Journal 21(2), 2011.

[9] Per-Áke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling: *High-Performance Concurrency Control Mechanisms for Main-Memory Databases*. VLDB 2012.

[10] Antti-Pekka Liedes and Antoni Wolski, *SIREN: A Memory-Conserving, Snapshot-Consistent Checkpoint Algorithm for in-Memory Databases*, ICDE 2006.

[11] Antti-Pekka Liedes and Petri Soini, *Memory allocator for optimistic data access*, US Patent number 12/121,133, 2008.

[12] Antti-Pekka Liedes, *Bottom-up Optimistic Latching Method For Index Trees*, US Patent 20120221531, 2012.

[13] Kerttu Pollari-Malmi, Jarmo Ruuth and Eljas Soisalon-Soininen, *Concurrency Control for B-Trees with Differential Indices*, IDEAS 2000.

[14] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N. Bhuyan: *No More Backstabbing... A Faithful Scheduling Policy for Multithreaded Programs*. PACT 2011.

[15] Vilho Raatikka and Antoni Wolski, *External Reference Entity Method For Link Failure Detection in Highly Available Systems*, International Workshop On Dependable Services and Systems (IWoDSS), 2010.

[16] Antoni Wolski and Vilho Raatikka, *Performance Measurement and Tuning of Hot-Standby Databases*, ISAS 2006, Lecture Notes in Computer Science, Volume 4328, 2006.

[17] Antoni Wolski and Kyösti Laiho, *Rolling Upgrades for Continuous Services*, ISAS 2004.

[18] Antoni Wolski, Sally Hartnell: *solidDB and Secrets of the Speed*, IBM Data Management (1), 2010.