MVStore

Overview

The MVStore is a persistent, log structured key-value store. It is used as default storage subsystem of H2, but it can also be used directly within an application, without using JDBC or SQL.

- MVStore stands for "multi-version store".
- Each store contains a number of maps that can be accessed using the `java.util.Map` interface.
- Both file-based persistence and in-memory operation are supported.
- It is intended to be fast, simple to use, and small.
- Concurrent read and write operations are supported.
- Transactions are supported (including concurrent transactions and 2-phase commit).
- The tool is very modular. It supports pluggable data types and serialization, pluggable storage (to a file, to off-heap memory), pluggable map implementations (B-tree, R-tree, concurrent B-tree currently), BLOB storage, and a file system abstraction to support encrypted files and zip files.

Example Code

The following sample code shows how to use the tool:

```
import org.h2.mvstore.*;

// open the store (in-memory if fileName is null)
MVStore s = MVStore.open(fileName);

// create/get the map named "data"
MVMap<Integer, String> map = s.openMap("data");

// add and read some data
map.put(1, "Hello World");
System.out.println(map.get(1));

// close the store (this will persist changes)
s.close();
```

Store Builder

The `MVStore.Builder` provides a fluid interface to build a store if configuration options are needed. Example usage:

```
MVStore s = new MVStore.Builder().
    fileName(fileName).
    encryptionKey("007".toCharArray()).
    compress().
    open();
```

The list of available options is:

- autoCommitBufferSize: the size of the write buffer.
- autoCommitDisabled: to disable auto-commit.
- backgroundExceptionHandler: a handler for exceptions that could occur while writing in the background.
- cacheSize: the cache size in MB.
- compress: compress the data when storing using a fast algorithm (LZF).
- compressHigh: compress the data when storing using a slower algorithm (Deflate).
- encryptionKey: the key for file encryption.
- fileName: the name of the file, for file based stores.
- fileStore: the storage implementation to use.
- pageSplitSize: the point where pages are split.
- readOnly: open the file in read-only mode.

R-Tree

The `MVRTreeMap` is an R-tree implementation that supports fast spatial queries. It can be used as follows:

```
// create an in-memory store
MVStore s = MVStore.open(null);

// open an R-tree map
MVRTreeMap<String> r = s.openMap("data",
        new MVRTreeMap.Builder<String>());

// add two key-value pairs
// the first value is the key id (to make the key unique)
// then the min x, max x, min y, max y
r.add(new SpatialKey(0, -3f, -2f, 2f, 3f), "left");
r.add(new SpatialKey(1, 3f, 4f, 4f, 5f), "right");

// iterate over the intersecting keys
Iterator<SpatialKey> it =
        r.findIntersectingKeys(new SpatialKey(0, 0f, 9f, 3f, 6f));
for (SpatialKey k; it.hasNext();) {
    k = it.next();
    System.out.println(k + ": " + r.get(k));
}
s.close();
```

The default number of dimensions is 2. To use a different number of dimensions, call `new MVRTreeMap.Builder<String>().dimensions(3)`. The minimum number of dimensions is 1, the maximum is 32.

Features

Maps

Each store contains a set of named maps. A map is sorted by key, and supports the common lookup operations, including access to the first and last key, iterate over some or all keys, and so on.

Also supported, and very uncommon for maps, is fast index lookup: the entries of the map can be efficiently accessed like a random-access list (get the entry at the given index), and the index of a key can be calculated efficiently. That also means getting the median of two keys is very fast, and a range of keys can be counted very quickly. The iterator supports fast skipping. This is possible because internally, each map is organized in the form of a counted B+-tree.

In database terms, a map can be used like a table, where the key of the map is the primary key of the table, and the value is the row. A map can also represent an index, where the key of the map is the key of the index, and the value of the map is the primary key of the table (for non-unique indexes, the key of the map must also contain the primary key).

Versions

A version is a snapshot of all the data of all maps at a given point in time. Creating a snapshot is fast: only those pages that are changed after a snapshot are copied. This behavior is also called COW (copy on write). Old versions are readable. Rollback to an old version is supported.

The following sample code show how to create a store, open a map, add some data, and access the current and an old version:

```
// create/get the map named "data"
MVMap<Integer, String> map = s.openMap("data");

// add some data
map.put(1, "Hello");
map.put(2, "World");

// get the current version, for later use
long oldVersion = s.getCurrentVersion();

// from now on, the old version is read-only
s.commit();

// more changes, in the new version
// changes can be rolled back if required
// changes always go into "head" (the newest version)
map.put(1, "Hi");
map.remove(2);

// access the old data (before the commit)
MVMap<Integer, String> oldMap =
        map.openVersion(oldVersion);

// print the old version (can be done
// concurrently with further modifications)
// this will print "Hello" and "World":
System.out.println(oldMap.get(1));
System.out.println(oldMap.get(2));

// print the newest version ("Hi")
System.out.println(map.get(1));
```

Transactions

To support multiple concurrent open transactions, a transaction utility is included, the `TransactionStore`. The tool supports "read committed" transaction isolation with savepoints, two-phase commit, and other features typically available in a database. There is no limit on the size of a transaction (the log is written to disk for large or long running transactions).

Internally, this utility stores the old versions of changed entries in a separate map, similar to a transaction log, except that entries of a closed transaction are removed, and the log is usually not stored for short transactions. For common use cases, the storage overhead of this utility is very small compared to the overhead of a regular transaction log.

In-Memory Performance and Usage

Performance of in-memory operations is about 50% slower than `java.util.TreeMap`.

The memory overhead for large maps is slightly better than for the regular map implementations, but there is a higher overhead per map. For maps with less than about 25 entries, the regular map implementations need less memory.

If no file name is specified, the store operates purely in memory. Except for persisting data, all features are supported in this mode (multi-versioning, index lookup, R-tree and so on). If a file name is specified, all operations occur in memory (with the same performance characteristics) until data is persisted.

As in all map implementations, keys need to be immutable, that means changing the key object after an entry has been added is not allowed. If a file name is specified, the value may also not be changed after adding an entry, because it might be serialized (which could happen at any time when autocommit is enabled).

Pluggable Data Types

Serialization is pluggable. The default serialization currently supports many common data types, and uses Java serialization for other objects. The following classes are currently directly supported: `Boolean, Byte, Short, Character, Integer, Long, Float, Double, BigInteger, BigDecimal, String, UUID, Date` and arrays (both primitive arrays and object arrays). For serialized objects, the size estimate is adjusted using an exponential moving average.

Parameterized data types are supported (for example one could build a string data type that limits the length).

The storage engine itself does not have any length limits, so that keys, values, pages, and chunks can be very big (as big as fits in memory). Also, there is no inherent limit to the number of maps and chunks. Due to using a log structured storage, there is no special case handling for large keys or pages.

BLOB Support

There is a mechanism that stores large binary objects by splitting them into smaller blocks. This allows to store objects that don't fit in memory. Streaming as well as random access reads on such objects are supported. This tool is written on top of the store, using only the map interface.

R-Tree and Pluggable Map Implementations

The map implementation is pluggable. In addition to the default `MVMap` (multi-version map), there is a multi-version R-tree map implementation for spatial operations.

Concurrent Operations and Caching

Concurrent reads and writes are supported. All such read operations can occur in parallel. Concurrent reads from the page cache, as well as concurrent reads from the file system are supported. Write operations first read the relevant pages from disk to memory (this can happen concurrently), and only then modify the data. The in-memory parts of write operations are synchronized. Writing changes to the file can occur concurrently to modifying the data, as writing operates on a snapshot.

Caching is done on the page level. The page cache is a concurrent LIRS cache, which should be resistant against scan operations.

For fully scalable concurrent write operations to a map (in-memory and to disk), the map could be split into multiple maps in different stores ('sharding'). The plan is to add such a mechanism later when needed.

Log Structured Storage

Internally, changes are buffered in memory, and once enough changes have accumulated, they are written in one continuous disk write operation. Compared to traditional database storage engines, this should improve write performance for file systems and storage systems that do not efficiently support small random writes, such as Btrfs, as well as SSDs. (According to a test, write throughput of a common SSD increases with write block size, until a block size of 2 MB, and then does not further increase.) By default, changes are automatically written when more than a number of pages are modified, and once every second in a background thread, even if only little data was changed. Changes can also be written explicitly by calling `commit()`.

When storing, all changed pages are serialized, optionally compressed using the LZF algorithm, and written sequentially to a free area of the file. Each such change set is called a chunk. All parent pages of the changed B-trees are stored in this chunk as well, so that each chunk also contains the root of each changed map (which is the entry point for reading this version of the data). There is no separate index: all data is stored as a list of pages. Per store, there is one additional map that contains the metadata (the list of maps, where the root page of each map is stored, and the list of chunks).

There are usually two write operations per chunk: one to store the chunk data (the pages), and one to update the file header (so it points to the latest chunk). If the chunk is appended at the end of the file, the file header is only written at the end of the chunk. There is no transaction log, no undo log, and there are no in-place updates (however, unused chunks are overwritten by default).

Old data is kept for at least 45 seconds (configurable), so that there are no explicit sync operations required to guarantee data consistency. An application can also sync explicitly when needed. To reuse disk space, the chunks with the lowest amount of live data are compacted (the live data is stored again in the next chunk). To improve data locality and disk space usage, the plan is to automatically defragment and compact data.

Compared to traditional storage engines (that use a transaction log, undo log, and main storage area), the log structured storage is simpler, more flexible, and typically needs less disk operations per change, as data is only written once instead of twice or 3 times, and because the B-tree pages are always full (they are stored next to each other) and can be easily compressed. But temporarily, disk space usage might actually be a bit higher than for a regular database, as disk space is not immediately re-used (there are no in-place updates).

Off-Heap and Pluggable Storage

Storage is pluggable. Unless pure in-memory operation is used, the default storage is to a single file.

An off-heap storage implementation is available. This storage keeps the data in the off-heap memory, meaning outside of the regular garbage collected heap. This allows to use very large in-memory stores without having to increase the JVM heap, which would increase Java garbage collection pauses a lot. Memory is allocated using `ByteBuffer.allocateDirect`. One chunk is allocated at a time (each chunk is usually a few MB large), so that allocation cost is low. To use the off-heap storage, call:

```
OffHeapStore offHeap = new OffHeapStore();
MVStore s = new MVStore.Builder().
        fileStore(offHeap).open();
```

File System Abstraction, File Locking and Online Backup

The file system is pluggable. The same file system abstraction is used as H2 uses. The file can be encrypted using a encrypting file system wrapper. Other file system implementations support reading from a compressed zip or jar file. The file system abstraction closely matches the Java 7 file system API.

Each store may only be opened once within a JVM. When opening a store, the file is locked in exclusive mode, so that the file can only be changed from within one process. Files can be opened in read-only mode, in which case a shared lock is used.

The persisted data can be backed up at any time, even during write operations (online backup). To do that, automatic disk space reuse needs to be first disabled, so that new data is always appended at the end of the file. Then, the file can be copied. The file handle is available to the application. It is recommended to use the utility class `FileChannelInputStream` to do this. For encrypted databases, both the encrypted (raw) file content, as well as the clear text content, can be backed up.

Encrypted Files

File encryption ensures the data can only be read with the correct password. Data can be encrypted as follows:

```
MVStore s = new MVStore.Builder().
    fileName(fileName).
    encryptionKey("007".toCharArray()).
    open();
```

The following algorithms and settings are used:

- The password char array is cleared after use, to reduce the risk that the password is stolen even if the attacker has access to the main memory.
- The password is hashed according to the PBKDF2 standard, using the SHA-256 hash algorithm.
- The length of the salt is 64 bits, so that an attacker can not use a pre-calculated password hash table (rainbow table). It is generated using a cryptographically secure random number generator.
- To speed up opening an encrypted stores on Android, the number of PBKDF2 iterations is 10. The higher the value, the better the protection against brute-force password cracking attacks, but the slower is opening a file.
- The file itself is encrypted using the standardized disk encryption mode XTS-AES. Only little more than one AES-128 round per block is needed.

Tools

There is a tool, the `MVStoreTool`, to dump the contents of a file.

Exception Handling

This tool does not throw checked exceptions. Instead, unchecked exceptions are thrown if needed. The error message always contains the version of the tool. The following exceptions can occur:

- `IllegalStateException` if a map was already closed or an IO exception occurred, for example if the file was locked, is already closed, could not be opened or closed, if reading or writing failed, if the file is corrupt, or if there is an internal error in the tool. For such exceptions, an error code is added so that the application can distinguish between different error cases.
- `IllegalArgumentException` if a method was called with an illegal argument.
- `UnsupportedOperationException` if a method was called that is not supported, for example trying to modify a read-only map.
- `ConcurrentModificationException` if a map is modified concurrently.

Storage Engine for H2

For H2 version 1.4 and newer, the MVStore is the default storage engine (supporting SQL, JDBC, transactions, MVCC, and so on). For older versions, append `;MV_STORE=TRUE` to the database URL.

File Format

The data is stored in one file. The file contains two file headers (for safety), and a number of chunks. The file headers are one block each; a block is 4096 bytes. Each chunk is at least one block, but typically 200 blocks or more. Data is stored in the chunks in the form of a log structured storage. There is one chunk for every version.

```
[ file header 1 ] [ file header 2 ] [ chunk ] [ chunk ] ... [ chunk ]
```

Each chunk contains a number of B-tree pages. As an example, the following code:

```
MVStore s = MVStore.open(fileName);
MVMap<Integer, String> map = s.openMap("data");
for (int i = 0; i < 400; i++) {
    map.put(i, "Hello");
}
s.commit();
for (int i = 0; i < 100; i++) {
    map.put(i, "Hi");
}
s.commit();
s.close();
```

will result in the following two chunks (excluding metadata):

Chunk 1:
- Page 1: (root) node with 2 entries pointing to page 2 and 3
- Page 2: leaf with 140 entries (keys 0 - 139)
- Page 3: leaf with 260 entries (keys 140 - 399)

Chunk 2:
- Page 4: (root) node with 2 entries pointing to page 5 and 3
- Page 5: leaf with 140 entries (keys 0 - 139)

That means each chunk contains the changes of one version: the new version of the changed pages and the parent pages, recursively, up to the root page. Pages in subsequent chunks refer to pages in earlier chunks.

File Header

There are two file headers, which normally contain the exact same data. But once in a while, the file headers are updated, and writing could partially fail, which could corrupt a header. That's why there is a second header. Only the file headers are updated in this way (called "in-place update"). The headers contain the following data:

```
H:2,block:2,blockSize:1000,chunk:7,created:1441235ef73,format:1,version:7,fletcher:3044e6cc
```

The data is stored in the form of a key-value pair. Each value is stored as a hexadecimal number. The entries are:

- H: The entry "H:2" stands for the H2 database.
- block: The block number where one of the newest chunks starts (but not necessarily the newest).
- blockSize: The block size of the file; currently always hex 1000, which is decimal 4096, to match the disk sector length of modern hard disks.
- chunk: The chunk id, which is normally the same value as the version; however, the chunk id might roll over to 0, while the version doesn't.
- created: The number of milliseconds since 1970 when the file was created.
- format: The file format number. Currently 1.
- version: The version number of the chunk.
- fletcher: The Fletcher-32 checksum of the header.

When opening the file, both headers are read and the checksum is verified. If both headers are valid, the one with the newer version is used. The chunk with the latest version is then detected (details about this see below), and the rest of the metadata is read from there. If the chunk id, block and version are not stored in the file header, then the latest chunk lookup starts with the last chunk in the file.

Chunk Format

There is one chunk per version. Each chunk consists of a header, the pages that were modified in this version, and a footer. The pages contain the actual data of the maps. The pages inside a chunk are stored right after the header, next to each other (unaligned). The size of a chunk is a multiple of the block size. The footer is stored in the last 128 bytes of the chunk.

```
[ header ] [ page ] [ page ] ... [ page ] [ footer ]
```

The footer allows to verify that the chunk is completely written (a chunk is written as one write operation), and allows to find the start position of the very last chunk in the file. The chunk header and footer contain the following data:

```
chunk:1,block:2,len:1,map:6,max:1c0,next:3,pages:2,root:4000004f8c,time:1fc,version:1
chunk:1,block:2,version:1,fletcher:aed9a4f6
```

The fields of the chunk header and footer are:

- chunk: The chunk id.
- block: The first block of the chunk (multiply by the block size to get the position in the file).
- len: The size of the chunk in number of blocks.
- map: The id of the newest map; incremented when a new map is created.
- max: The sum of all maximum page sizes (see page format).
- next: The predicted start block of the next chunk.
- pages: The number of pages in the chunk.
- root: The position of the metadata root page (see page format).
- time: The time the chunk was written, in milliseconds after the file was created.
- version: The version this chunk represents.
- fletcher: The checksum of the footer.

Chunks are never updated in-place. Each chunk contains the pages that were changed in that version (there is one chunk per version, see above), plus all the parent nodes of those pages, recursively, up to the root page. If an entry in a map is changed, removed, or added, then the respective page is copied, modified, and stored in the next chunk, and the number of live pages in the old chunk is decremented. This mechanism is called copy-on-write, and is similar to how the Btrfs file system works. Chunks without live pages are marked as free, so the space can be re-used by more recent chunks. Because not all chunks are of the same size, there can be a number of free blocks in front of a chunk for some time (until a small chunk is written or the chunks are compacted). There is a delay of 45 seconds (by default) before a free chunk is overwritten, to ensure new versions are persisted first.

How the newest chunk is located when opening a store: The file header contains the position of a recent chunk, but not always the newest one. This is to reduce the number of file header updates. After opening the file, the file headers, and the chunk footer of the very last chunk (at the end of the file) are read. From those candidates, the header of the most recent chunk is read. If it contains a "next" pointer (see above), those chunk's header and footer are read as well. If it turned out to be a newer valid chunk, this is repeated, until the newest chunk was found. Before writing a chunk, the position of the next chunk is predicted based on the assumption that the next chunk will be of the same size as the current one. When the next chunk is written, and the previous prediction turned out to be incorrect, the file header is updated as well. In any case, the file header is updated if the next chain gets longer than 20 hops.

Page Format

Each map is a B-tree, and the map data is stored in (B-tree-) pages. There are leaf pages that contain the key-value pairs of the map, and internal nodes, which only contain keys and pointers to leaf pages. The root of a tree is either a leaf or an internal node. Unlike file header and chunk header and footer, the page data is not human readable. Instead, it is stored as byte arrays, with long (8 bytes), int (4 bytes), short (2 bytes), and variable size int and long (1 to 5 / 10 bytes). The page format is:

- length (int): Length of the page in bytes.
- checksum (short): Checksum (chunk id xor offset within the chunk xor page length).
- mapId (variable size int): The id of the map this page belongs to.
- len (variable size int): The number of keys in the page.
- type (byte): The page type (0 for leaf page, 1 for internal node; plus 2 if the keys and values are compressed with the LZF algorithm, or plus 6 if the keys and values are compressed with the Deflate algorithm).
- children (array of long; internal nodes only): The position of the children.
- childCounts (array of variable size long; internal nodes only): The total number of entries for the given child page.
- keys (byte array): All keys, stored depending on the data type.
- values (byte array; leaf pages only): All values, stored depending on the data type.

Even though this is not required by the file format, pages are stored in the following order: For each map, the root page is stored first, then the internal nodes (if there are any), and then the leaf pages. This should speed up reads for media where sequential reads are faster than random access reads. The metadata map is stored at the end of a chunk.

Pointers to pages are stored as a long, using a special format: 26 bits for the chunk id, 32 bits for the offset within the chunk, 5 bits for the length code, 1 bit for the page type (leaf or internal node). The page type is encoded so that when clearing or removing a map, leaf pages don't have to be read (internal nodes do have to be read in order to know where all the pages are; but in a typical B-tree the vast majority of the pages are leaf pages). The absolute file position is not included so that chunks can be moved within the file without having to change page pointers; only the chunk metadata needs to be changed. The length code is a number from 0 to 31, where 0 means the maximum length of the page is 32 bytes, 1 means 48 bytes, 2: 64, 3: 96, 4: 128, 5: 192, and so on until 31 which means longer than 1 MB. That way, reading a page only requires one read operation (except for very large pages). The sum of the maximum length of all pages is stored in the chunk metadata (field "max"), and when a page is marked as removed, the live maximum length is adjusted. This allows to estimate the amount of free space within a block, in addition to the number of free pages.

The total number of entries in child pages are kept to allow efficient range counting, lookup by index, and skip operations. The pages form a counted B-tree.

Data compression: The data after the page type are optionally compressed using the LZF algorithm.

Metadata Map

In addition to the user maps, there is one metadata map that contains names and positions of user maps, and chunk metadata. The very last page of a chunk contains the root page of that metadata map. The exact position of this root page is stored in the chunk header. This page (directly or indirectly) points to the root pages of all other maps. The metadata map of a store with a map named "data", and one chunk, contains the following entries:

- chunk.1: The metadata of chunk 1. This is the same data as the chunk header, plus the number of live pages, and the maximum live length.
- map.1: The metadata of map 1. The entries are: name, createVersion, and type.
- name.data: The map id of the map named "data". The value is "1".
- root.1: The root position of map 1.
- setting.storeVersion: The store version (a user defined value).

Similar Projects and Differences to Other Storage Engines

Unlike similar storage engines like LevelDB and Kyoto Cabinet, the MVStore is written in Java and can easily be embedded in a Java and Android application.

The MVStore is somewhat similar to the Berkeley DB Java Edition because it is also written in Java, and is also a log structured storage, but the H2 license is more liberal.

Like SQLite 3, the MVStore keeps all data in one file. Unlike SQLite 3, the MVStore uses is a log structured storage. The plan is to make the MVStore both easier to use as well as faster than SQLite 3. In a recent (very simple) test, the MVStore was about twice as fast as SQLite 3 on Android.

The API of the MVStore is similar to MapDB (previously known as JDBM) from Jan Kotek, and some code is shared between MVStore and MapDB. However, unlike MapDB, the MVStore uses is a log structured storage. The MVStore does not have a record size limit.

Current State

The code is still experimental at this stage. The API as well as the behavior may partially change. Features may be added and removed (even though the main features will stay).

Requirements

The MVStore is included in the latest H2 jar file.

There are no special requirements to use it. The MVStore should run on any JVM as well as on Android.

To build just the MVStore (without the database engine), run:

```
./build.sh jarMVStore
```

This will create the file `bin/h2mvstore-2.2.224.jar` (about 200 KB).