Department of Informatics, University of Zürich

BSc Thesis

Implementation and Evaluation of a Key-Value Store for Flash-based Storage

Jonas Schmid

Place of birth: Zug, Switzerland

Matriculation number: 06-908-677

Email: jonas.schmid@uzh.ch

3. Februrary 2012

supervised by Prof. Dr. Michael Böhlen and Anton Dignös

University of Zurich[UZH]

Department of Informatics

Abstract

In the last decade solid state drives (SSD) gained more and more importance in the field of databases, due to their fast access time compared to traditional hard disk drives (HDD). The flash translation layer (FTL), an abstraction layer of SSDs, provides the same API as traditional HDDs and makes their use transparent. On top of FTL, traditional access methods and algorithms operate acceptably without any modification. The asymmetry of access time of read and write operations and the requirement to perform an expensive erase operation prior to an in-place update, raises the need for specialized access methods. This thesis shows an implementation and an evaluation of an approach for a key-value store, called in-page logging. It reduces the number of erase operations due to in-place updates on data pages by using logs.

Abstract

Solid State Drives (SSD) haben im letzten Jahrzehnt dank ihrer schnellen Zugriffszeit gegenüber traditionellen Festplatten mehr und mehr an Bedeutung im Datenbankumfeld gewonnen. Der Flash Translation Layer (FTL), eine Abstraktionsschickt von der SSD, stellt das gleiche API zur Verfügung wie traditionelle Festplatten und machen dessen Gebrauch transparent. Traditionelle Zugriffmethoden und Algorithmen oberhalb des FTLs funktionieren deshalb auch ohne Veränderungen akzeptabel. Die asymmetrische Lese- und Schreibzugriffszeit und die Anforderung, eine teure Löschoperation vor einer in-situ Aktualisierung zu machen, verlangt nach dem Bedürfnis spezialisierter Zugriffsfunktionen. Diese Arbeit zeigt eine Implementation und eine Evaluation von einem Ansatz für ein Key-Value Store namens in-page logging. Dieser Ansatz reduziert die Anzahl der von einer in-situ Aktualisierung einer Datenpage ausgelösten Löschoperationen mit der Verwendung von Logs.

Contents

List of Figures

1 Introduction

In the last few years, flash based storage, namely solid state drives (SSD), gained increasingly importance in the database field due to their fast access time compared to traditional hard disk drives (HDD). The increase in capacity and the decrease of price made them more and more interesting for the storage of large amounts of data. The asymmetry of read and write access time and expensive in-place updates poses new challenges and raises the need for specialized access methods and algorithms.

The solution, that is presented in this thesis, is called in-page logging (IPL) and was proposed by Sang-Wong Lee et. al. in [15]. The idea of IPL is to make usage of efficient write mechanism to reduce the number of erase cycles. The in-page logging approach wins over with its small and simple implementation. This gives the advantage that this approach can be applied to an existing database solution only by changing the implementation of the storage layer and keeping the rest of the database solution as it is.

Combined with the implementation of a key-value store, I show that the proposed solution has about 45% less read access, about 50% less write access and erase cycles, and about 50% faster estimated runtime than a regular non-SSD implementation of a key-value store on a SSD in a read-write workload.

The purpose of this thesis is to develop and implement an SSD specific key-value store and comparing it against a non-SSD implementation of a key-value store, both running on a SSD. The SSD implementation is based on the in-page logging approach presented in [15]. The output shows, that the overall speed can be increased with small changes on the database environment, namely on the storage layer.

The rest of this paper is organized as follows. Chapter 2 gives an overview on key-value stores and flash memory. Chapter 3 describes the problem. Chapter 4 introduces a possible solution called "in-page logging". Chapter 5 gives an understanding of the implementation, whereas in Chapter 6 the work is evaluated. Chapter 7 gives a conclusion and points out future work.

2 Overview of Key-Value Store and Flash Memory

2.1 Key-Value Store

Key-value stores [9] are similar to NoSQL [4, 5, 18] databases. A NoSQL database is, like the name says, a database where it is not possible to execute a SQL query directly on it. Some commercial key-value stores are Oracles Berkeley DB [6], Googles BigTable [10] or Amazons Dynamo [13].

A relation of a key-value store consists of two columns, one for the key and one for the data, the latter is also called value. There are no limitations, whether the value contains a simple string or an object or any other data types provided by the client application, since the value is stored as a binary large object (BLOB). Thus, the value in key-value stores has no schema, so the client application is responsible for the semantics of the data and how it is organized. Additionally, key-value stores can be used as a column store, where the value is the column and the key is used for the index.

An advantage of key-value stores is their simple API. To manipulate relations, three functions *put(key, data)*, *get(key)* and *remove(key)* are provided. Search operations are only possible on keys, thus access can be optimized, e.g. with indexes. Further the API has no SQL interface. If the caller wants to run a SQL query, it has to manage its SQL query and schema on its own.

Key-value stores are often considered for update and lookup intensive online transaction processing (OLTP) workloads or specialized workloads as document repositories, where they score with its lightweight design.

2.2 Flash Memory Overview

Despite flash memory [2] was invented around 1980, it made its break through not till the mid nineties in memory sticks and sd-cards. Even though the first SSDs [8, 16] came into production at the same time, it took about 10 years until a consumer friendly SSD was established.

Flash memory is a type of non-volatile memory that can be electrically erased and reprogrammed. Compared to HDDs [3], there are no mechanical parts inside flash memory. SSDs are based on NAND chips, which must be read or written block wise. Further, NAND chips are built with either single level cells (SLC) or multi level cells (MLC). Whereas a single level cell can only store one bit, a multi level cell can store several bits. The advantage of SLCs

are its writing speed, lower power consumption and longer life span comparing to a MLC. Since they are more expensive than MLC, SLC flash memory is used in areas where high-performance is needed. The advantage of a MLC is lower cost per unit of storage. Despite there exists also some enterprise MLC, they are mostly used in consumer flash storage.

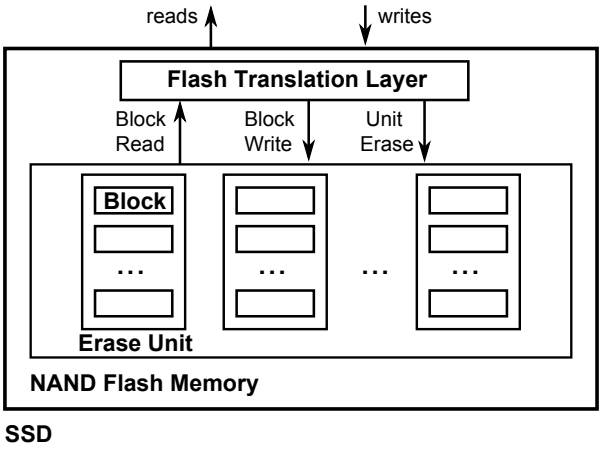


Figure 2.1: Simple Design of a Solid State Drive

Figure 2.1 shows the design of a SSD. A SSD consists of its flash memory and the corresponding flash translation layer (FTL). More about FTL in chapter 3. The flash memory is divided into erase units, whereas every erase unit is divided into blocks.

Flash memory has an important limitation. Although it can be read or written block wise, a block can only be updated or deleted by erasing the corresponding erase unit. An erase sets all bits of the erase unit to 1. After an erase, it is possible to write on the erase unit again at any position. Once a bit has been set to 0, it cannot be changed to 1 except with an erase operation. It follows directly that an in-place update is only possible, if the data bits switching from 1 to 0 or stays at 0. For example, a value 1111 can be updated to 1101. Continuous updates to 1001, 1000 and finally 0000 are possible. Further updates to this value will result in an erase operation. However, this thesis does not consider such updates, but consider a block to erase whenever it has been written once.

In Table 3.1, we see a comparison between the access times of a HDD and a SSD [15]. Despite the values are some years old, they show the time difference for read, write and erase operations. Further, we will refer flash memory as NAND based flash memory, because NOR based flash memory, which is not mentioned in this overview, is not relevant in this thesis.

3 Problem Statement

Due to the Flash Translation Layer (FTL) [14], a SSD appears to upper layers like a conventional disk drive. Hence an FTL provides the same API as a HDD. Because of the FTL, conventional disk-base database algorithms and access methods operate acceptably without any modification. But without any modification, every in-place update on a SSD will result in an erase operation, as explained in Chapter 2.2.

Despite the expensive update, a SSD is still faster than a conventional HDD, as following example shows: Assume a block size of *4 KByte* and the size of an erase unit of *128 KByte*. According to Table 3.1, an in-place update of a block on a conventional disk costs $27.4ms$. Since an erase unit contains 32 blocks, an in-place update of a block on a SSD will cost 32 read operations to read the erase unit, one erase operation, and 32 write operations to write the erase unit. This takes $10.46ms$.

Nevertheless, if an in-place update on a SSD could be replaced by writing the additional update into an empty place, it would be much faster, i.e. 400μs (4KB), than a complete erase and rewrite, despite these updates need to be read and merged with the page when accessing it. The idea is to take advantage of the asymmetry of read and write access time to postpone an erase by writing logs and sacrifice fast read operations. The approach in the next chapter realizes exactly this idea.

Media	Access time		
	Read	Write	Erase
Magnetic Disk[1]	12.7 ms (2KB)	13.7 ms (2KB)	N/A
NAND Flash[2]	80μs (2KB)	200μs (2KB)	1.5ms (128KB)

[1] Disk: Seagate Barracuda 7200.7 ST380011A, average access times including seek and rotational delay;
[2] NAND Flash: Samsung K9WAG08U1A 16 Gbits SLC NAND

Table 3.1: Access Speed: Magnetic disk vs. NAND Flash

4 In-Page Logging

To find an accurate approach, I have read several papers.

- [17] proposes a buffer manager for DBS running on flash based disks. They developed a new replacement policy in which they separate modified and unmodified pages into two buffer pools. They take account of the read-write asymmetry and achieve an improvement of the overall performance up to 33%.
- [11] introduces FlashStore, a high throughput persistent key-value store, that uses flash memory as a non-volatile *cache* between RAM and hard disk. One of the design goals of FlashStore is to use flash memory in an FTL friendly manner.
- [12] introduces SkimpyStash, a RAM space skimpy key-value store on flash-based storage, designed for high throughput, low latency server applications. The distinguishing feature of SkimpyStash is the design goal of extremely low RAM footprint at about 1($\pm$0.5) byte per key-value pair, which is more aggressive than earlier designs like FlashStore [11].

The solution I have chosen is the "in-page logging" approach, proposed in [15]. I have chosen this approach, because I'd like to show that with minimal changes to the storage layer of a key-value store, the overall performance can already be improved.

Besides taking advantage of the characteristics of flash memory, such as uniform access speed due to no mechanical latency and asymmetric read and write access speed, in-page logging has the aim to minimize the changes made to the database system. Hence the design changes will be limited mainly to the buffer manager and file manager, as this chapter will show.

To avoid confusion about the terms block and page, we define a block as a contiguous sequence of bytes on disk, whereas a page is a structured block in the implementation. A block is the main disk storage unit, whereas a sector is the minimal IO unit. Figure 4.1 illustrates the design of in-page logging. On the upper half, it shows a buffer page, whereas the lower half shows the layout of an erase unit. Following enumeration points out the main structures of the design:

- A buffer page consists of a data page and the corresponding log.
- The erase unit has a size of *128 KByte*.
- Every erase unit has 28 data pages of *4 KByte* each and 32 log sectors of *512 byte* each.
- Every page has one corresponding log. With this decision, there are 4 logs per erase unit left for future extensions.

- A page and its log sector have the same size in the buffer as their equivalent in the flash memory.

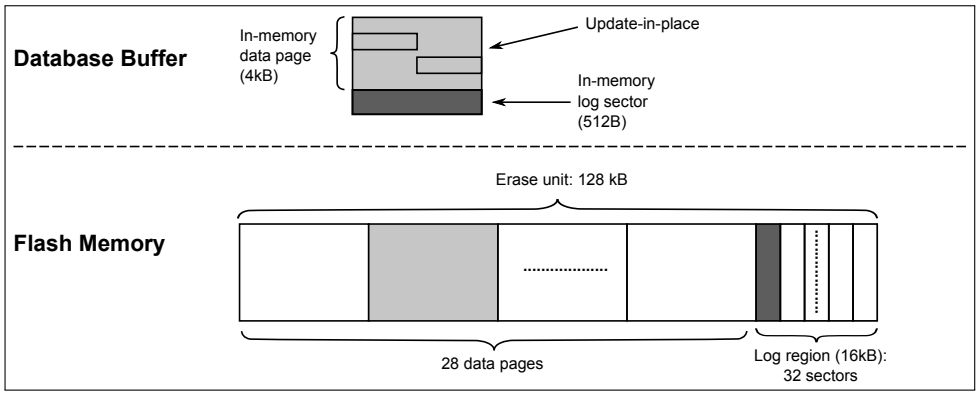


Figure 4.1: The Design of In-Page Logging

The idea of in-page logging is as follows: A relation consists of a file, which in turn contains several erase units with several pages in it. Without in-page logging, it was necessary to rewrite the whole erase unit for an update operation, even if only a single record from a page was affected. This leads to frequent write and erase operations. To avoid this, in-page logging writes only the changes made to a page to the database on per-page basis, instead of writing the entire page. These change requests can be written into the corresponding log. Figure 4.2 shows an example of an update operation: The relation has a page with two tuple T1 and T2. To update T2, the page needs to be read into memory in order to perform the update. Before the non-SSD implementation can perform a the write operation on the file, the erase unit needs to be erased, which implies to read the whole erase unit, erase it, update the page, and write the erase unit with the updated page back. On the contrary, in-page logging writes the change request into the log, and writes only the log to disk. That makes a difference of one write operation instead of 32 write-, 32 read- and one erase operation.

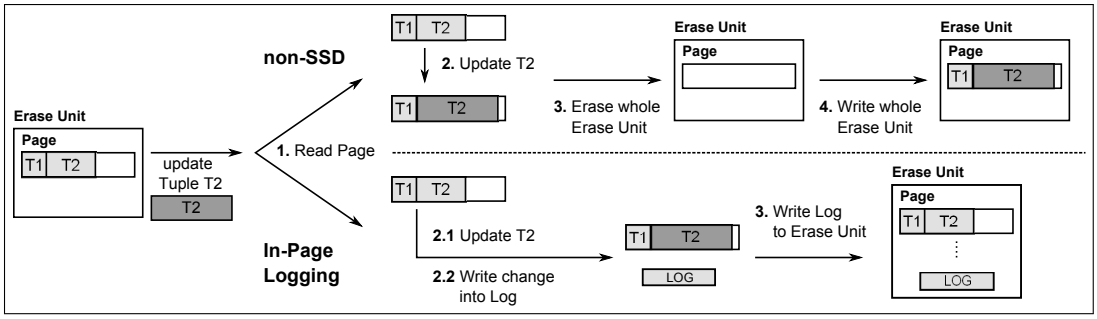


Figure 4.2: Example of an Update Operation

A read operation in in-page logging always results in reading the page and the corresponding log. Figure 4.3 shows an example by reading page two of an erase unit. After reading the page and its log into memory, the application needs to merge the page with the corresponding log.

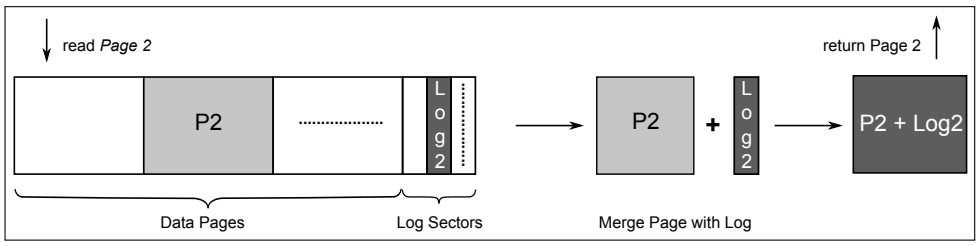


Figure 4.3: Read a Page with corresponding Log

This means to go through the log and apply every log entry to the page. For example, if the entry states a deletion of key 45, the tuple with key 45 has to be removed from the page. Despite the higher read effort due to the log, this won't be regarded as a disadvantage, since read operations on a SSD are highly efficient.

To optimize an erase operation, in-page logging colocates the log of a page within the same erase unit, as Figure 4.1 shows. If the log sectors were placed in another erase unit, an erase operation would have to erase two erase units, one for the pages, and one where the log sectors are placed.

If a write or update operation occurs, the affected page will be read from disk and loaded into the buffer, if it's not already there. Then the in-memory page is written / updated in place and handled like a traditional page (see Figure 4.1). In addition, the buffer adds a corresponding log to the buffer page. An in-memory log sector will automatically be created if needed, and deleted when the log entries are written to the SSD. To keep the read effort within a limit in my implementation, every page has exactly one corresponding log sector, as described before.

A page in the buffer is called dirty, if it has been changed comparing to its copy on the disk. If a dirty page has to be swapped out from the buffer to disk, it is not always necessary to write the whole page, since every change request was also written into the log. As long as its log was not already written into the flash memory, it is enough to "swap out" the log. After the log has been written to disk, it will be deleted in the buffer. The page itself stays in the buffer and is not dirty anymore, since the page in the buffer is equal to the page on the disk merged with its corresponding log.

Whenever both, a page and its log have been written to disk, the erase unit needs to be erased and rewritten for the next write operation affecting this page. When rewriting the erase unit, the pages will be merged with its corresponding logs. Algorithm 1 shows the rewriting and associated merging of the pages with their logs. The Algorithm is as follows:

1. The function takes two erase units as input. B_0 is the old erase unit to merge, whereas B is a new, unused erase unit.

2. Then the algorithm goes through every page of B_0. If a log for a corresponding page exists, then every log entry is applied to the page, as described before with the example of deleting the tuple with key 45.

3. If the page is merged or the page did not have any log, it will be written into the new erase unit B.

4. At the end, the old erase unit B_0 will be erased and freed.

After merging due to step 2., all log sectors in B are empty.

Algorithm 1 Merge Operation

Input: B_0: an old erase unit to merge
Output: B: a new erase unit with merged content

```
1:  function MERGE(B_0, B)
2:      allocate a free erase unit B
3:      for each data page p in B_0 do
4:          if any log entry for p exists then
5:              p' ← apply the log entry to p
6:              write p' to B
7:          else
8:              write p to B
9:          end if
10:     end for
11:     erase and free B_0
12: end function
```

5 Implementation

This chapter is subdivided into two sections. The first section describes the implementation of a non-SSD key-value store, the second section explains the changes to be made to optimize the key-value store for SSDs, according to the in-page logging approach. Further is to mention that the implementation is a functional simulation and not yet tested on a real SSD. Because an FTL [14] abstracts structures like erase units, we simulate the rewriting of erase units.

5.1 Magnetic Hard Disk Drive Implementation

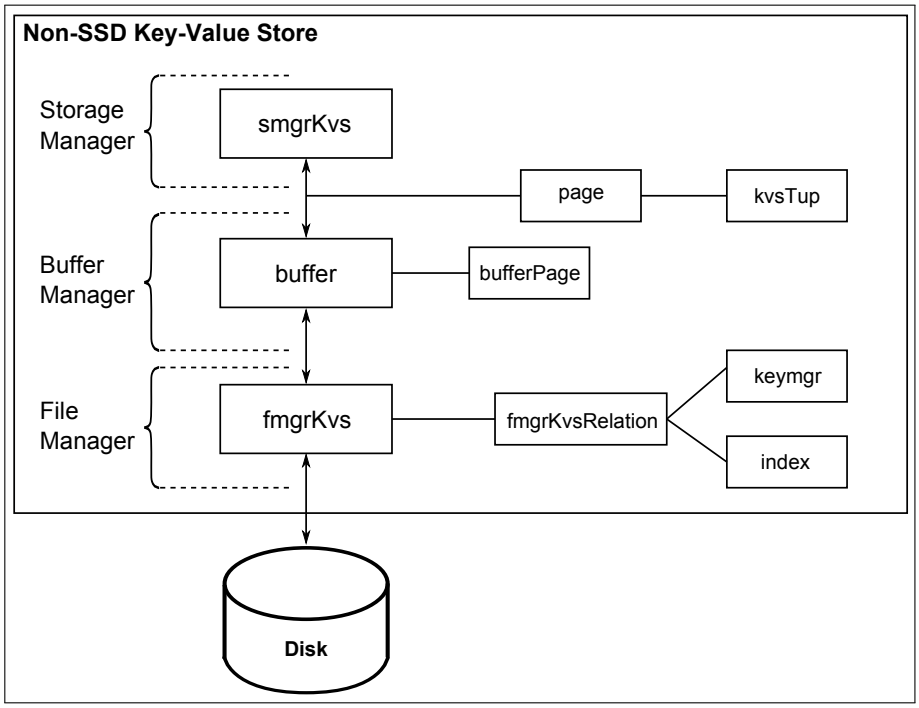


Figure 5.1: Non-SSD Key-Value Store Implementation

Figure 5.1 shows an overview of the non-SSD implementation.

- The file manager *fmgrKvs* deals with relations called *fmgrKvsRelation*. Every relation has a corresponding key manager *keymgr* and an *index*. A key manager provides unique keys within the relation. An index stores the key and its corresponding block number of the page which the tuple contains. Further an index provides efficient lookup access

when looking for a tuple by its key. The file manager is responsible to communicate with its underlying disk thus dealing with data files organized into blocks.

- The buffer manager *buffer* is the link between the storage manager and file manager. Every block transfer goes through the buffer, which caches the traffic in its *bufferPages* to reduce block IOs. The buffer fetches the blocks from the file manager.
- The storage manager *smgrKvs* on top acts as API towards other applications. It structures the receiving data into *kvsTup* tuple and uses the buffer manager to store and retrieve them.

I created a header file named *ctypes.h* (Listing 5.1), which contains a macro to define the *blocksize* to 4 KByte, and a typedef for the *key*.

Listing 5.1: ctypes.h

```
#define BLKSIZE 4096

typedef size_t key;
```

The blocksize defines the size of a block on disk. As we will see in the next subsection, the file manager manages a file using several blocks, since a block is the unit for IO transfers in this non-SSD implementation. Figure 5.2 shows an example with a file consisting of three blocks and a buffer with one buffer page. In the example, Block 1 is already in the Buffer. If a query needs access to tuple T1 or tuple T2, the buffer can immediately return these two without any disk access. If a query needs access to tuple T3 or T4, the buffer manager has to read and load the corresponding block from the hard disk / file into the buffer through the file manager.

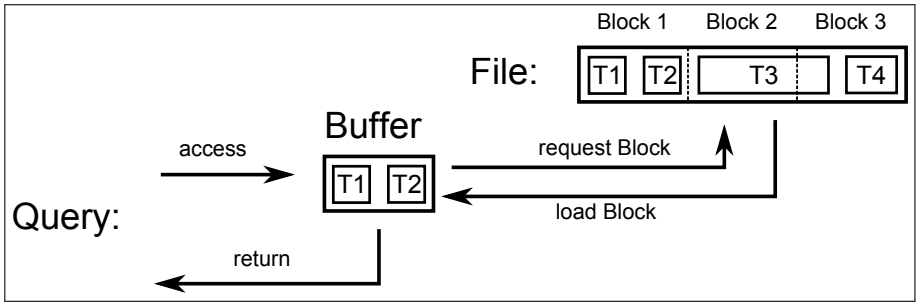


Figure 5.2: Example of a Buffer with one BufferPage

5.1.1 Structures

Page

As the file manager deals with blocks, we define the structure of a block and call a formatted block a page. Therefore the size of a page corresponds to the size of a block, which is *4 KByte*. The structure of a page is designed to have as less overhead as possible, so that the implementation stays lightweight. Figure 5.3 shows an example of a structured page that

contains 3 tuples. It consists of a header, that includes the number of tuples the page contains and the size of free space. After the header we store the offsets of the tuples in the page. The corresponding tuple is then added at the end of the available freespace.

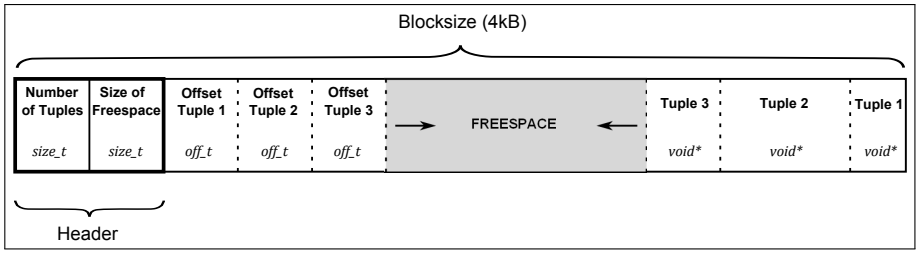


Figure 5.3: Example of a Page with 3 Tuples

The implementation of a page follows the idea to have as less overhead as possible (Listing 5.2). Thus a page is represented as a void pointer **page*, because the size of a page is always constant. To manipulate pages we have the following functions:

- A page can be created by calling *makeEmtyPage()*.
- *pAddTup()* provides the functionality to add a tuple to the page, whereas *pRemoveTup()* removes a tuple from a page.
- As a last function implemented, the caller can search after a tuple in the page by key with *pGetTupFromKey()*.

Listing 5.2: page.h

```
typedef void* page;

#define SIZEOFPAGE BLKSIZE

/* Create an empty Page */
page makeEmtyPage();

/* Add a Tuple to the Page */
int pAddTup(page p, const kvsTup tup);

/* Removes a Tuple from the Page */
int pRemoveTup(page p, const kvsTup tup);

/* Returns the Tuple with Key k */
kvsTup pGetTupFromKey(page p, key k);
```

Tuple

A key and its corresponding value is stored in a key-value store tuple. The structure of a tuple is held simple and lightweight, so that the implementation contains no unnecessary overhead. Figure 5.4 shows the structure of a tuple. A tuple consists of the value and a header, whereas a header contains the key and the total size of the tuple. The size of a tuple can vary, but it's limited by the size of a page minus its header and offset. The implementation remains small and well-arranged, as Listing 5.3 shows. Because a tuple is represented as a void pointer

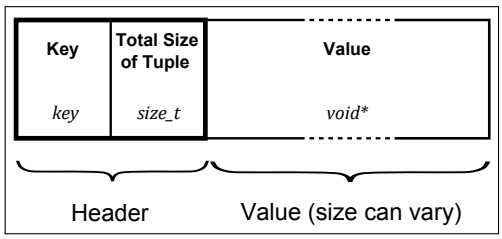


Figure 5.4: Structure of a Key-Value Store Tuple

**kvsTup*, we need to keep the overall size of the tuple. A tuple can be created by calling *makeKvsTup()*

Listing 5.3: kvsTup.h

```
typedef void* kvsTup;

/* Creates a Kvs Tuple */
kvsTup makeKvsTup(key k, size_t datasize, const void *data);
```

5.1.2 File Manager

A relation is represented as a struct, defined in the file manager (Listing 5.4). The client application comes with the name of the tablespace, database and relation and creates a *fmgrKvsRelInfo*. From this, the file manager converts a *fmgrKvsRelInfo* into a *fmgrKvsRelation*, that contains the filename and the corresponding file of the relation.

Listing 5.4: Structs of fmgrKvs.h

```
typedef struct fmgrKvsRelation
{
  char * fileName;
  FILE * file;
} fmgrKvsRelation;

typedef struct fmgrKvsRelInfo
{
  char tblSp[MAX_FILENAMESIZE + 1];
  char db[MAX_FILENAMESIZE + 1];
  char rel[MAX_FILENAMESIZE + 1];
} fmgrKvsRelInfo;
```

Further the file manager provides a couple of functions (Listing 5.5). First it has to be able to create a file to the corresponding relation where to write the data. In addition it has to provide a function to remove a relation. As a second pair of functions it needs to provide the ability to write and read data. As a third it provides functionalities to either extend or truncate a file. The following enumeration describes the order in which the functions have to be called:

1. As the first and last functions, *fmgrKvsInit()* and *fmgrKvsShutdown()* have to be called at the startup and the shutdown respectively of the file manager.

2. Before any call to other functions can be performed, *fmgrKvsCreate()* have to be called, that creates a *fmgrKvsRelation* from a *fmgrKvsRelInfo*.

3. To open the file of a relation, *fmgrKvsOpen()* have to be called. With a parameter the caller can decide whether the file will be created, if it does not exist, or just opened. *fmgrKvsClose()* provides the functionality to close the file. *fmgrKvsExists()* can be used to test, if a specific file of a relation already exists.

4. When the file is open, one calls *fmgrKvsRead()* and *fmgrKvsWrite()* to read and write from / to the file.

5. *fmgrKvsExtend()* and *fmgrKvsTruncate()* provide the functionality to extend or truncate a file.

6. With *fmgrKvsNrBlocks()* it can be checked how many blocks of data the file contains.

7. *fmgrKvsUnlink()* allows the caller to remove the file from the relation, whereas *fmgrKvsFlush()* take care of writing the content of the file permanently to the disk.

Listing 5.5: fmgrKvs.h

```
/* Initialize File Manager */
void fmgrKvsInit();

/* Shutdown File Manager */
void fmgrKvsShutdown();

/* Create a Relation */
fmgrKvsRelation *fmgrKvsCreate(const fmgrKvsRelInfo *relInfo);

/* Close a Relation */
int fmgrKvsClose(fmgrKvsRelation *rel);

/* Open the Relation */
int fmgrKvsOpen(fmgrKvsRelation *rel, bool create);

/* Read one Block from the Relation */
size_t fmgrKvsRead(fmgrKvsRelation *rel, size_t blckNr, void *buffer);

/* Write one Block to the Relation */
size_t fmgrKvsWrite(fmgrKvsRelation *rel, size_t blckNr, void *buffer);

/* Drop the File from the Relation */
int fmgrKvsUnlink(fmgrKvsRelation *rel);

/* Check if the File from the Relation exist */
bool fmgrKvsExists(fmgrKvsRelation *rel);

/* Number of Blocks from the File of the Relation  */
size_t fmgrKvsNrBlocks(fmgrKvsRelation *rel);

/* Extend the File from the Relation */
bool fmgrKvsExtend(fmgrKvsRelation *rel, int blckNr, void *buffer);

/* Truncate the File from the Relation */
int fmgrKvsTruncate(fmgrKvsRelation *rel, int blckNr);

/* Write the Content of the File permanently to disk */
int fmgrKvsFlush(fmgrKvsRelation *rel);
```

5.1.3 Buffer Manager

The buffer manager *Buffer* is represented as a struct, defined in Listing 5.6. It consists of an array of *bufferPage*s and a counter variable clock. A *bufferPage* itself consists of:

- The corresponding relation the data page belongs to,
- the page number,
- the data page,
- a variable to keep its clock and
- a boolean to notice if the page has been changed ($= dirty$) comparing to its version on the disk or not.

The clock is used to realize a *Least Recently Used* (LRU) [1, 7] page replacement strategy. Every time a *bufferpage* is used, it gets the highest clock. If the clock can't be incremented anymore due to overflow, it will be reset internally. The *bufferPage*s will be adapted analogous. The number of *bufferPage*s is defined by the macro *BUF_NUM_PAGES*.

Listing 5.6: Structs of buffer.h

```
#define BUF_NUM_PAGES 10

typedef struct Buffer
{
  void *data;           /* array of bufferPage of length BUF_NUM_PAGES */
  size_t clock;         /* next clock to give to a page */
} Buffer;

typedef struct bufferPage
{
  fmgrKvsRelation *rel;
  size_t pageNum;       /* pageNum of Page in Relation */
  page p;
  size_t clock;         /* clock time for buffer strategy */
  bool isDirty;         /* has page been written */
} bufferPage;
```

Further the buffer manager provides functions to ensure its functionality (Listing 5.7). The following enumeration summarizes them:

1. As the first and last functions, *bufferKvsInit()* and *bufferKvsShutdown()* have to be called at the startup and the shutdown respectively of the buffer manager.
2. Before any call to other functions can be performed, *bufOpenRel()* needs to be called, which opens and, if necessary, creates the relation in the file manager. At the end, the relation needs to be closed with *bufCloseRel()*.
3. *bufferKvsRead()* and *bufferKvsWrite()* provide the functionality to read and write a page from and to the file by the buffer. They use the internal functions *swapIn()* and *swapPage-Out()* to get a page into or out of the buffer respectively.
4. Finally *flushBuffer()* is used to clean the buffer by writing every dirty page to its file. The data pages however stay in the buffer.

Listing 5.7: buffer.h

```
/* Initialize Buffer Manager */
void bufferKvsInit();

/* Shutdown Buffer Manager */
void bufferKvsShutdown();

/* Write the Content of the Buffer permanently to disk */
void flushBuffer();

/* Read corresponding Block either from Buffer or File */
size_t bufferKvsRead(fmgrKvsRelation *rel, int blockNr, page p);

/* Write corresponding Block into Buffer */
void bufferKvsWrite(fmgrKvsRelation *rel, int blockNr, page p);

/* Open the Relation */
fmgrKvsRelation *bufOpenRel(char *tblSpc, char *db, char *rel);

/* Close the Relation */
void bufCloseRel(fmgrKvsRelation *rel);
```

5.1.4 Storage Manager

The storage manager is the API of the key-value store. As described in Section 2.1, a key-value store provides the three functions *put()*, *get()* and *remove()*. Beside these three function, the storage manager needs additional functions to ensure the functionality of the whole storage layer. These functions are shown in Listing 5.9. The following enumeration describes the order in which the functions need to be called:

1. To start and to terminate the storage manager, the client has to call *smgrKvsInit()* and *smgrKvsShutdown()* respectively. Both functions call internally the related initial- and shutdown functions from the buffer- and file manager.

2. Before any call to other functions can be performed, the client application has to open and create the relation with *openRel()* first. On contrast *closeRel()* needs to be called to close a relation.

3. Beside the three initial functions *insertKeyVal()* to insert a value, *searchKey()* to get a tuple and *removeKey()* to remove a value, the storage manager provides a fourth function *updateKey()* which allows to update the value of a specific key an. An update is realized as a delete and followed by insert.

A call to this modification functions requires an index and a key manager as parameter. We consider the index as a main memory index, in which the key and the corresponding page number of the page the tuple is inside are stored. A key manager is defined as follows:

Key Manager

Every relation has a key manager, which provides unique keys. As Listing 5.8 shows, a key manager is implemented as a struct. This contains only a counter variable *nextVal* up to now. A key manager provides following functions:

- With *makeKeymgr()* the caller has to create a key manager.
- *getNewKey()* is a getter function which returns a unique key.
- *storeKeymgr()* provides the functionality of saving a key manager to a file. This can be restored with *makeKeymgrFromFile()*.

Listing 5.8: keymgr.h

```
typedef struct keymgr
{
  key nextVal;
} keymgr;

/* Create a new Key Manager */
keymgr *makeKeymgr();

/* Open an existing Key Manager from File */
keymgr *makeKeymgrFromFile(char *filepath);

/* Store the Key Manager into a File */
size_t storeKeymgr(char *filepath, keymgr *mgr);

/* Get a unique key */
key getNewKey(keymgr *mgr);
```

Listing 5.9: smgrKvs.h

```
/* Initialize Storage Manager */
void smgrKvsInit();

/* Shutdown Storage Manager */
void smgrKvsShutdown();

/* Open a Relation */
fmgrKvsRelation *openRel(char *tblSpc, char *db, char *rel);

/* Close a Relation */
void closeRel(fmgrKvsRelation *rel);

/* Insert a new Value with a unique Key from the Key Manager */
int insertKeyVal(key k, void *value, size_t sizeOfValue, fmgrKvsRelation *rel,
                                                                    kvalIdx ix);
/* Search Tuple according to its Key */
kvsTup searchKey(key k, fmgrKvsRelation *rel, kvalIdx ix);

/* Remove a Tuple according to its Key */
int removeKey(key k, fmgrKvsRelation *rel, kvalIdx ix);

/* Updates a Tuple according to its Key */
int updateKey(const key k, void *value, size_t datasize, fmgrKvsRelation *rel,
                                                                    kvalIdx ix);
```

5.2 Flash Memory Specific Implementation

In the next subsections only those sublayers are described which have been changed or added to the implementation due to the SSD specific implementation. Figure 5.5 shows the overview of the whole implementation with highlighted parts that have changed. Beside the three managers have changed, the log comes along as a new structure.

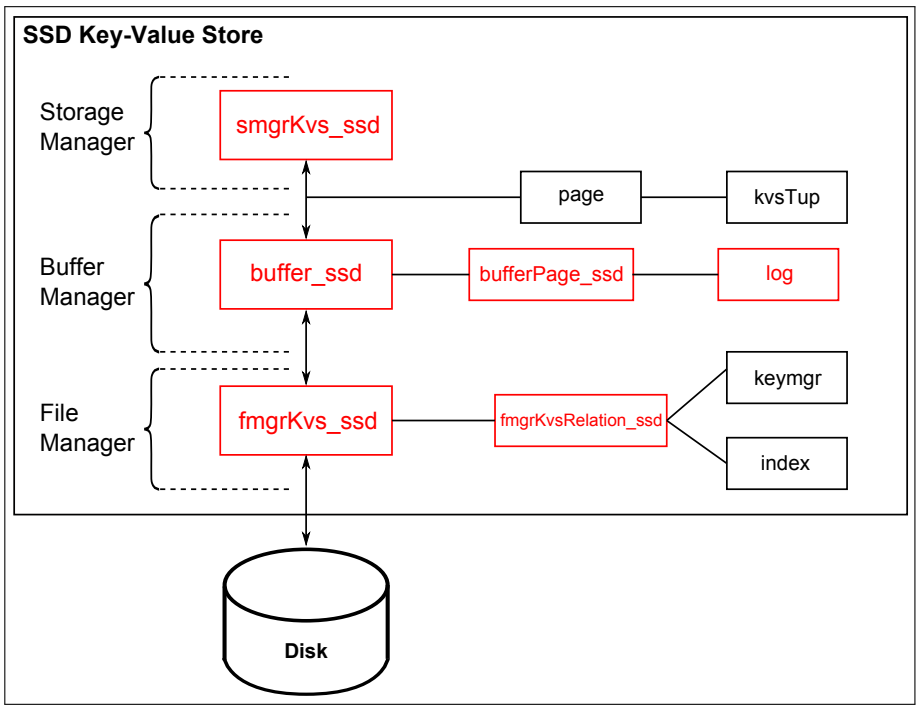


Figure 5.5: SSD Key-Value Store Implementation

The header file *ctypes.h*, which has been renamed to *ctypes_ssd.h*, has been extended, as Listing 5.10 shows. New are the definitions of a sector size (which is equal to the size of a Log) to *512 byte* and the size of an erase unit to *128 KByte*.

Listing 5.10: ctypes_ssd.h

```
#include "ctypes.h"

#define SECTORSIZE 512
#define ERASEUNSIZ (128*1024)
```

5.2.1 Structures

Log

The size of a log is equal a sector, which is *512 byte*. The structure of a log is similar to the structure of a page. Figure 5.6 shows a log with three entries. The log consists of a header, which includes the number of entries the log contains and the size of free space. After the header we store the offsets of the entries in the log. The corresponding log entry is then added at the end of the available freespace.
A log entry contains its type (add, update or delete), the key and if the entry is for adding or updating, the corresponding value. Figure 5.7 shows the structure of both an add or update entry and delete entry.
The implementation of a log follows the idea to have as less overhead as possible (Listing

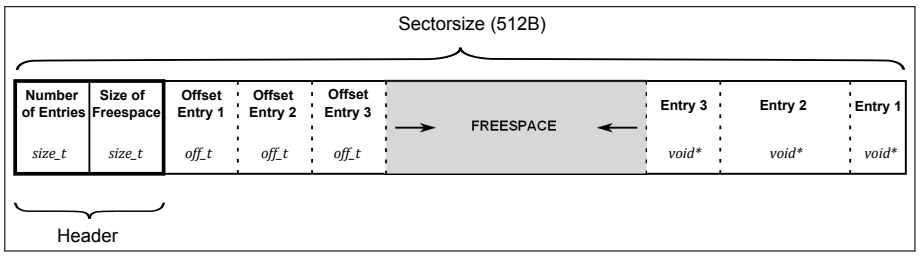


Figure 5.6: Example of a Log with 3 Entries

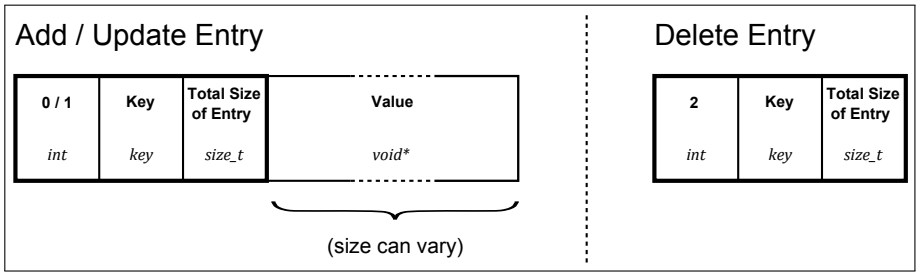


Figure 5.7: Structure of a Log Entry

5.11). Thus a log is represented as a void pointer **logIPS*, because its size is always constant. To manipulate logs we have the following functions:

- A log can be created by calling *makeEmtylogIPS()*.
- *makeLogIPSentry()* will create an entry.
- With *addEntryTologIPS()* the entry can be added to a log.

Listing 5.11: Log Listing of log_ssd.h

```
#define LOGIPSSIZE SECTORSIZE

#define ADDENTRY 0
#define UPDATEENTRY 1
#define DELETEENTRY 2

typedef void *logIPS;

/* Create an emtpy Log */
logIPS makeEmtylogIPS();

typedef void *logIPSEntry;

/* Create a Log Entry */
logIPSEntry makeLogIPSentry(int lgEntryTyp, key k, size_t datasize, void *data);

/* Add a Log Entry to a Log */
int addEntryTologIPS(logIPS l, logIPSEntry ent);
```

5.2.2 File Manager

The representation of a relation has not been changed and is the same struct as in the non-SSD implementation in section 5.4. The requirements for the file manager of the SSD based

implementation are similar as for the non-SSD one. Beside the functions from Listing 5.5, the SSD based file manager needs additional functions to handle the new introduced erase unit and log. Listing 5.12 gives a full overview of all functions. The following enumeration pretends the order the functions have to be called, whereas only 4., 5. and 7. have been changed compared to enumeration in section 5.1.2:

1. As the first and last functions, *fmgrKvsInit_ssd()* and *fmgrKvsShutdown_ssd()* have to be called at the startup and the shutdown respectively of the file manager.

2. Before call any other functions, *fmgrKvsCreate_ssd()* have to be called, that creates a *fmgrKvsRelation_ssd* from a *fmgrKvsRelInfo*.

3. To open the file of a relation, *fmgrKvsOpen_ssd()* needs to be called. With a parameter the caller can decide whether the file will be created if it does not exists, or just opened. *fmgrKvsClose_ssd()* provides the functionality to close the file. *fmgrKvsExists_ssd()* can be used to test, if a specific file of a relation already exists.

4. If the file is open, one calls
   - *fmgrKvsReadBlock_ssd()* and *fmgrKvsWriteBlock_ssd()*,
   - *fmgrKvsReadLog_ssd()* and *fmgrKvsWriteLog_ssd()*,
   - *fmgrKvsReadErUn_ssd()* and *fmgrKvsWriteErUn_ssd()*

   to read and write the corresponding block, log or erase unit from / to the file.

5. If a block is already written on the disk and needs to be updated, *fmgrKvsEraseUnit_ssd()* clears the corresponding erase unit.

6. *fmgrKvsExtend_ssd()* and *fmgrKvsTruncate_ssd()* provide the functionality to extend or truncate a file.

7. With *fmgrKvsNrEraseUnits_ssd()* it can be checked how many erase units the file contains.

8. *fmgrKvsUnlink_ssd()* allows the caller to remove the file from the relation, whereas *fmgrKvsFlush_ssd()* takes care of writing the content of the file permanently to the SSD.

Listing 5.12: fmgrKvs_ssd.h

```
/* Initialize File Manager */
void fmgrKvsInit_ssd();

/* Shutdown File Manager */
void fmgrKvsShutdown_ssd();

/* Create a Relation */
fmgrKvsRelation_ssd *fmgrKvsCreate_ssd(const fmgrKvsRelInfo *relInfo);

/* Close a Relation */
int fmgrKvsClose_ssd(fmgrKvsRelation_ssd *rel);

/* Open the Relation */
int fmgrKvsOpen_ssd(fmgrKvsRelation_ssd *rel, bool create);
```

```
/* Read one Block from the Relation */
size_t fmgrKvsReadBlock_ssd(fmgrKvsRelation_ssd *rel, int erUnit, int blckNr,
                                                                    void *buffer);
/* Write one Block to the Relation */
size_t fmgrKvsWriteBlock_ssd(fmgrKvsRelation_ssd *rel, int erUnit, int blckNr,
                                                                    void *buffer);
/* Read one Log from the Relation */
size_t fmgrKvsReadLog_ssd(fmgrKvsRelation_ssd *rel, int erUnit, int logNr,
                                                                    void *buffer);
/* Write one Log to the Relation */
size_t fmgrKvsWriteLog_ssd(fmgrKvsRelation_ssd *rel, int erUnit, int logNr,
                                                                    void *buffer);
/* Read one Erase Unit from the Relation */
size_t fmgrKvsReadErUn_ssd(fmgrKvsRelation_ssd *rel, int erUnit, void *buffer);

/* Write one Erase Unit to the Relation */
size_t fmgrKvsWriteErUn_ssd(fmgrKvsRelation_ssd *rel, int erUnit, void *buffer);

/* Clear one Erase Unit */
void fmgrKvsEraseUnit_ssd(fmgrKvsRelation_ssd *rel, int erUnit);

/* Drop the File from the Relation */
int fmgrKvsUnlink_ssd(fmgrKvsRelation_ssd *rel);

/* Check if the File from the Relation exist */
bool fmgrKvsExists_ssd(fmgrKvsRelation_ssd *rel);

/* Number of Erase units from the File of the Relation  */
size_t fmgrKvsNrEraseUnits_ssd(fmgrKvsRelation_ssd *rel);

/* Extend the File from the Relation */
bool fmgrKvsExtend_ssd(fmgrKvsRelation_ssd *rel, int erUnitNr, void *buffer);

/* Truncate the File from the Relation */
int fmgrKvsTruncate_ssd(fmgrKvsRelation_ssd *rel, int erUnitNr);

/* Write the Content of the File permanently to disk */
int fmgrKvsFlush_ssd(fmgrKvsRelation_ssd *rel);
```

5.2.3 Buffer Manager

The buffer manager *buffer_ssd* is represented as a struct, defined in the buffer manager (Listing 5.13). It consists of an array of *bufferPages_ssd* and a counter variable clock. A *bufferPage_ssd* itself consists of:

- The corresponding relation the data page belongs to,
- the page number,
- the data page,
- a variable to keep its clock and
- a boolean to notice if the page has been changed ($= dirty$) comparing to its version on the disk or not,
- a boolean to notice if the log has been changed ($= dirty$) comparing to its version on the disk or not,
- the corresponding log to the page,

- a boolean to notice if the page on the disk is erase and only in the buffer or not,
- a boolean which indicates if the page is a *bomb* or not.

Bomb is defined as the condition when the log is full or too small for an entry. If the *bombBit* is set to true, the log of this bufferPage will then be disabled and changes are made directly to the page without any entry in the log. When the page needs to be swapped out, the process has to write the page, even though if the corresponding log on the file is empty. With the bombBit the size of a inserted tuple is not limited to the size of a log. Further it allows unlimited accessing (adding, updating and removing) to the page in a row. Without the bomBit the log would be full after a certain number of access, which would then result in swapping out either the page or the log.

The clock is used to realize a LRU strategy, as described in section 5.1.3. The number of *bufferPages_ssd* is defined by the macro *BUF_NUM_PAGES_SSD*.

Listing 5.13: Structs of buffer_ssd.h

```
#define BUF_NUM_PAGES_SSD 10

typedef struct Buffer_ssd
{
  void *data;          /* array of bufferPage_ssd of length BUF_NUM_PAGES */
  size_t clock;        /* next clock to give to a page */
} Buffer_ssd;

typedef struct bufferPage_ssd
{
  fmgrKvsRelation_ssd *rel;
  size_t pageNum;
  page p;
  size_t clock;        /* clock time for buffer strategy */
  bool pagDirty;       /* has page been written */
  bool logDirty;       /* has logIPS been written */
  logIPS l;
  bool isErasedOnDisk;
  bool bombBit;        /* if a log is full or too small, the page becomes a "bomb".
                        * as long as page is in buffer, changes are made directly
                        * to the page, log is not needed anymore.
                        */
} bufferPage_ssd;
```

Further the buffer manager provides, similar to its non-SSD based equivalent, functions to ensure its functionality (Listing 5.14). The following enumeration summarizes them, whereas only 3. has been changed compared to enumeration in section 5.1.3:

1. As the first and last functions, *bufferKvsInit_ssd()* and *bufferKvsShutdown_ssd()* have to be called at the startup and the shutdown respectively of the buffer manager.

2. Before call any other functions, *bufOpenRel_ssd()* needs to be called, which opens and, if necessary, creates the relation in the file manager. At the end, the relation needs to be closed with *bufCloseRel_ssd()*.

3. *bufferKvsRead_ssd()* and *bufferKvsWrite_ssd()* provide the functionality to read and write a page from and to the file by the buffer. They use the internal functions *swapIn_ssd()* and *swapPageOut_ssd()*, which are responsible to get a page into or out of the buffer respectively. *mergePageWithlogIPS_ssd()* provides the functionality of merging a page with its log.

4. Finally *flushBuffer_ssd()* is used to clean the buffer by writing every dirty page to its file. The data pages however stay in the buffer.

If a swap out of a page results in an erase and rewriting the erase unit, the buffer manager will swap out every page of the corresponding erase unit. In detail, an erase is proceeded in the following way:

1. The whole erase unit is loaded into the buffer.
2. Every page will be merged with its log.
3. The merged pages will be written to the disk into erase unit, whereas their logs will be deleted.

Listing 5.14: buffer_ssd.h

```
/* Initialize Buffer Manager */
void bufferKvsInit_ssd();

/* Shutdown Buffer Manager */
void bufferKvsShutdown_ssd();

/* Open the Relation */
fmgrKvsRelation_ssd *bufOpenRel_ssd(char *tblSpc, char *db, char *rel);

/* Close the Relation */
void bufCloseRel_ssd(fmgrKvsRelation_ssd *rel);

/* Read corresponding Block either from Buffer or File */
size_t bufferKvsRead_ssd(fmgrKvsRelation_ssd *rel, int blockNr, page p);

/* Write corresponding Block into Buffer */
void bufferKvsWrite_ssd(fmgrKvsRelation_ssd *rel, int blockNr, page p,
                                                          logIPSEntry entry);
/* Merge a Page with its Log */
page mergePageWithlogIPS_ssd(page p, logIPS l);

/* Write the Content of the Buffer permanently to disk */
void flushBuffer_ssd();
```

5.2.4 Storage Manager

The interface of the storage manager is not allowed to change with the SSD based implementation. The functions have not been changed either, as Listing 5.15 shows. However inside the implementation, the functions needed to be adapted since a log come along with a page. For an order the functions need to be called, I refer to the enumeration in section 5.1.4.

Listing 5.15: smgrKvsS_ssd.h

```
/* Initialize Storage Manager */
void smgrKvsInit_ssd();

/* Shutdown Storage Manager */
void smgrKvsShutdown_ssd();

/* Open a Relation */
fmgrKvsRelation_ssd *openRel_ssd(char *tblSpc, char *db, char *rel);
```

```
/* Close a Relation */
void closeRel_ssd(fmgrKvsRelation_ssd *rel);

/* Insert a new Value with a unique Key from the Key Manager */
int insertKeyVal_ssd(key k, void *value, size_t sizeOfValue,
                                          fmgrKvsRelation_ssd *rel, kvalIdx ix);
/* Search Tuple according to its Key */
kvsTup searchKey_ssd(key k, fmgrKvsRelation_ssd *rel, kvalIdx ix);

/* Remove a Tuple according to its Key */
int removeKey_ssd(key k, fmgrKvsRelation_ssd *rel, kvalIdx ix);

/* Updates a Tuple according to its Key */
int updateKey_ssd(const key k, void *value, size_t datasize,
                                          fmgrKvsRelation_ssd *rel, kvalIdx ix);
```

6 Evaluation

The evaluation has been run on a IBM Thinkpad T60p with a mobile Core2Duo T7400 2.16GHz processor, 2GB RAM and Windows 7 Professional 32bit.

The implementation ran on a conventional HDD, hence the evaluation is only simulated. The number of read- and write access and the number of rewriting an erase unit has been measured. To get an estimated runtime, the measured results have been multiplied with the values of Table 3.1. Further the assumption is that the non-SSD implementation triggers an erase every time a write occurs, except the page on the disk is empty.

Three main hypotheses have been established. The first handles inserts only, the second handles reads only and the third handles a random case. The first two hypotheses have sub-hypotheses each, where I distinguish between the fact that the requested page is already in the buffer or not. Despite it is very unlikely that the first two hypotheses occur on an ordinary database system, they give an interesting insight in their performance.

To set up the test environment, two relations have been created. Insert and read operations are always performed on the first relation, whereas the second relation only contains one tuple. Further the size of the buffer has been limited to one page. The second relation is used to evaluate the two sub-hypotheses, where the requested page is not in the buffer. After every access to relation one, the accessed page needs to be swapped out of the buffer. By reading the tuple from relation two, the page in the buffer is forced to be swapped out. Hence the next time a page from relation one is accessed, it won't be in the buffer anymore.

Every hypothesis has been run 10 times. I started with performing 1000 insert or read-access respectively and incremented the number of access by 1000 every run. The read, write and erase count will be shown each in separate graphs, whereas a fourth graph estimates the overall runtime.

6.1 Hypothesis 1 - Insert only

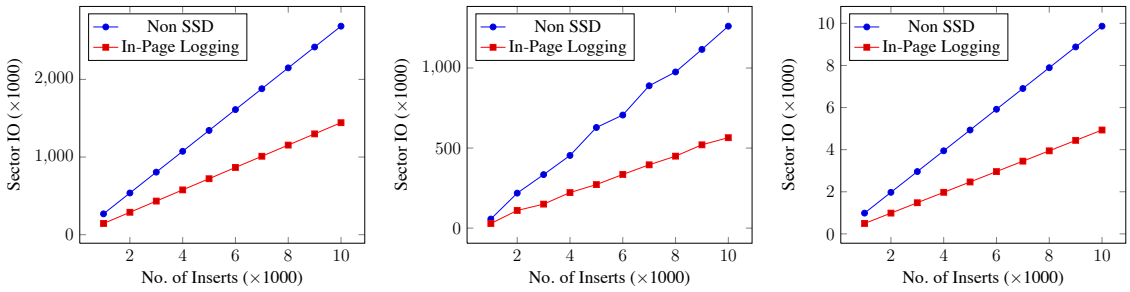
6.1.1 Hypothesis 1.1 - Sequential Insert

This hypothesis evaluates how the implementations perform in case of inserts only, whereas I assume that the page, where the tuple is inserted, is not in the buffer. This results in swapping out a page from the buffer and swapping in the requested page every time a tuple is inserted. I call this sequential insert.

The expectation is, that the in-page logging approach performs at least as good or better as the non-SSD implementation. The read- as well as write- and erase counts from the in-page logging approach should be about 50% smaller, since almost every insert in the non-SSD

implementation triggers an erase operation. Whereas with in-page logging, about only every second insert should trigger an erase.

Figure 6.1 shows the read, write and erase count graphs of hypothesis 1.1, whereas Figure 6.2 shows the estimated runtime:



(a) Read Count (b) Write Count (c) Erase Count

Figure 6.1: Sequential Insert: Read, Write and Erase Count

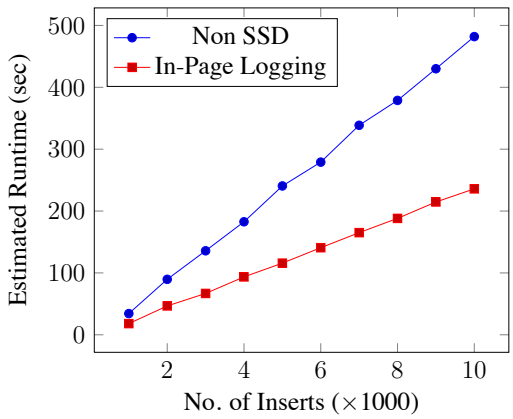


Figure 6.2: Sequential Insert: Estimated Runtime

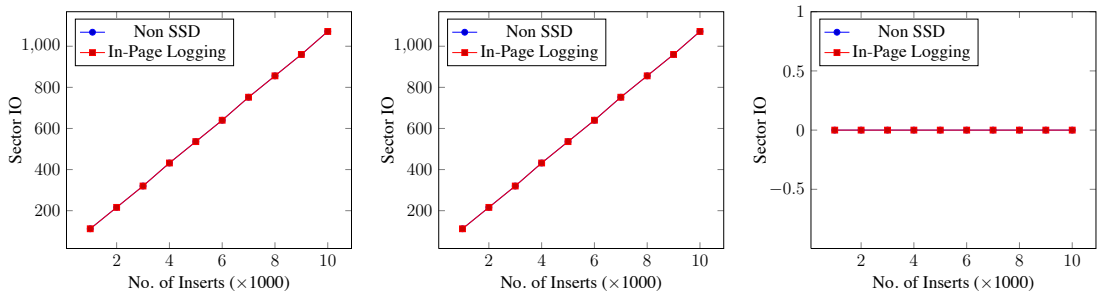
According to the results the hypothesis has been fulfilled. In particular Figure 6.1 shows the improvement due to in-page logging compared to the non-SSD implementation. Figure 6.2 is the logical conclusion of the previous figure. This result of nearly 50% improvement validates the described effect of the in-page logging approach, that only about every second insert triggers an erase operation.

6.1.2 Hypothesis 1.2 - Bulk Insert

This hypothesis evaluates how the implementations perform in case of inserts only, whereas the page, where the tuple is inserted, this time is in the buffer. I call this bulk insert.

I expect identical results from both implementations. Since the actual page is always in the buffer, they will only read the empty page once from the file into the buffer and will swap the page out, if it is full. This implies, that the page will never be updated on disk. Hence there should be no erase operations.

Figure 6.3 shows the read, write and erase count graphs of hypothesis 1.2, whereas Figure 6.4 shows the estimated runtime:



(a) Read Count (b) Write Count (c) Erase Count

Figure 6.3: Bulk Insert: Read, Write and Erase Count

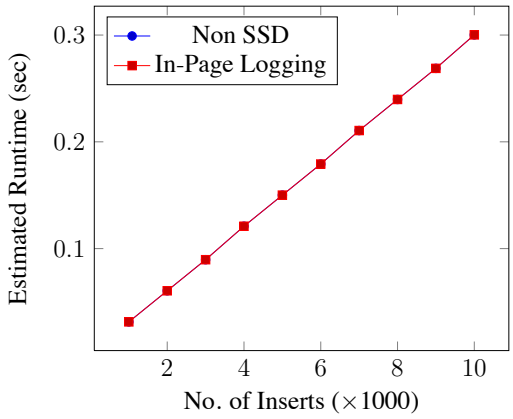


Figure 6.4: Bulk Insert: Estimated Runtime

As the results illustrate, the hypothesis was fulfilled. In both figures, Figure 6.3 and 6.4, the graphs show identical results for both implementations. As expected, there are no erase counts (Figure 6.3(c)).

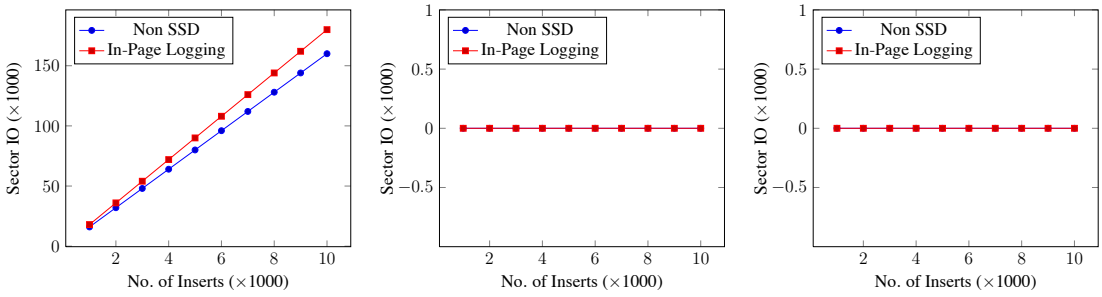
6.2 Hypothesis 2 - Read only

6.2.1 Hypothesis 2.1 - Random Read

This hypothesis evaluates how the implementations perform in case of reads only, where I assume that the page, that contains the tuple, is not in the buffer. That results in swapping out a page from the buffer and swapping in the requested page every time we want to read a tuple. I call this random read.

Because only read operations are performed, there should be no write and erase counts, in neither of the implementations. According to the fact, that a read in the in-page logging approach results in reading the page and additionally its corresponding log, I expect that it will result in more read count, and thus have worse write performance as the non-SSD implementation.

Figure 6.5 shows the read, write and erase count graphs of hypothesis 2.1. Figure 6.6 shows the estimated runtime:



(a) Read Count (b) Write Count (c) Erase Count

Figure 6.5: Random Read: Read, Write and Erase Count

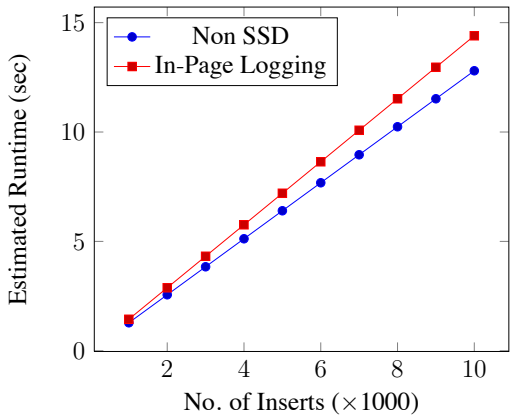


Figure 6.6: Random Read: Estimated Runtime

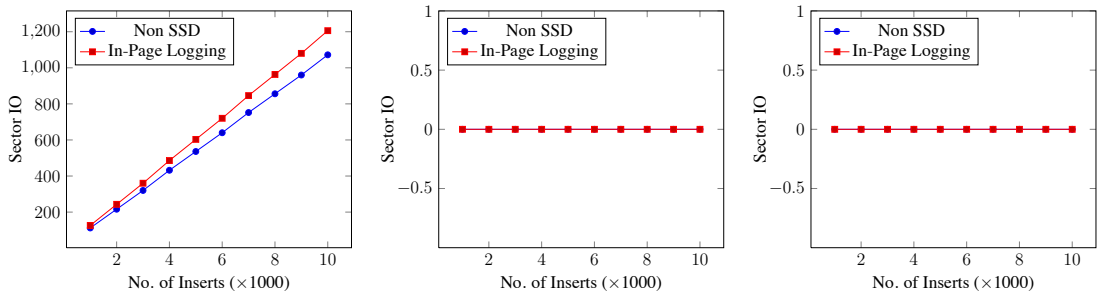
As the results illustrate, the hypothesis was approved. In both figures, Figure 6.5(a) and 6.6, the graphs show the expected difference, which is the additional log that the in-page logging implementation has to read from disk. As expected, there are no write and erase counts (Figure 6.5(b) and 6.5(c)).

6.2.2 Hypothesis 2.2 - Sequential Scan

This hypothesis evaluates how the implementations perform in case of reads only, where the page, that contains the tuple, is always in the buffer. I call this hypothesis sequential scan.

I expect that the outcome of this hypothesis is similar to the previous hypothesis 2.1 (random read), and that the outcome has less read access than in hypothesis 2.1 (random read), since the required page stays in the buffer. Nevertheless, the in-page logging approach still should have more read counts, and thus a worse overall runtime.

Figure 6.7 shows the read, write and erase count graphs of hypothesis 2.2, whereas Figure 6.8 shows the estimated runtime:



(a) Read Count (b) Write Count (c) Erase Count

Figure 6.7: Sequential Scan: Read, Write and Erase Count

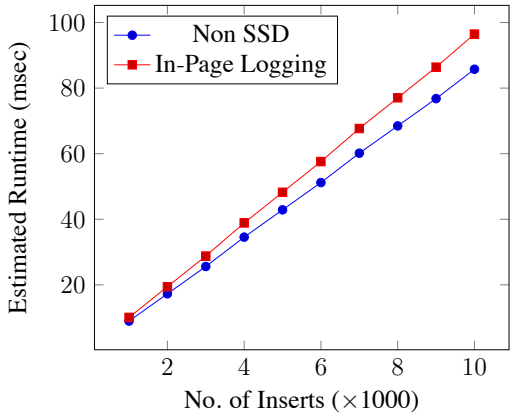


Figure 6.8: Sequential Scan: Estimated Runtime

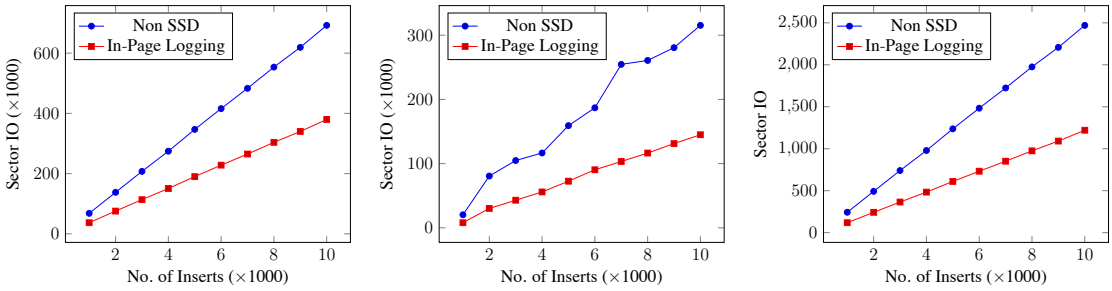
The outcome is, as expected, almost identical with the previous hypothesis 2.1 (random read). As predicted there are no write and erase counts (Figure 6.7(b) and 6.7(c)). There is also the slight gap in the read counts and runtime of the two implementations, because of the additional log the in-page logging has to read. The only difference to hypothesis 2.1 (random read) is the lower number of read counts due to buffering, hence also the estimated runtime.

6.3 Hypothesis 3 - Random Read and Write

This hypothesis evaluates how the implementations perform for random read and write access. At the starting point, the relation is half full. This prevents the first couple of read operations of reading nothing.

I expect that the in-page logging approach performs for every measurement at least as good or better as the non-SSD implementation. The write as well as the erase counts from the in-page logging should be about 50% lower, since almost every insert in the non-SSD implementation triggers an erase operation. Whereas for the in-page logging, only every second insert on an existing page triggers an erase. Due to the higher read effort in the in-page logging approach, I expect that the improvement in read counts does not fall out as high as expected in the write- and erase counts. Nevertheless, the number of read operations in the in-page logging approach should be clearly lower than in the non-SSD implementation.

Figure 6.9 shows the read, write and erase count graphs of hypothesis 3, whereas Figure 6.10 shows the estimated runtime:



(a) Read Count (b) Write Count (c) Erase Count

Figure 6.9: Random Read and Write: Read, Write and Erase Count

The outcome covers my expectations of the hypothesis, as Figure 6.9 and 6.10 show. In particular Figure 6.9(b) and 6.9(c) with an improvement of about 50% show the advantage of the in-page logging compared to the non-SSD implementation. The improvement of 45% in Figure 6.9(a) falls out fairly high too.

The difference of about 50% in Figure 6.10 shows clearly, that the additional write effort to read the logs does not weight much, since read operations are highly efficient on a SSD. Rather the reduction of almost 50% in erase operations is the main improvement. Less erase operations imply also fewer write- and read operations.

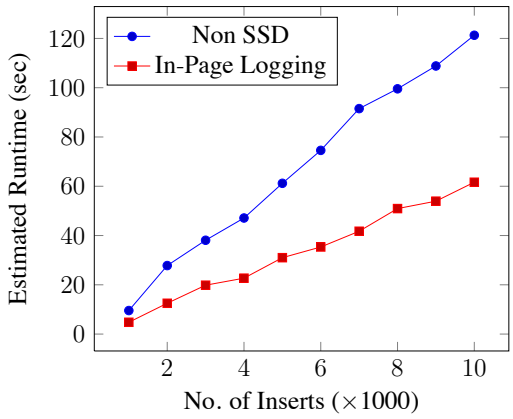


Figure 6.10: Random Read and Write: Estimated Runtime

6.4 Impact of Buffer Size

Further I wanted to know how the impact of increasing the buffer size turns out at both implementations. This evaluation was done with 100'000 access operations.

Figure 6.11 shows clearly that both implementations profit equally up to a certain point. The block I/O gets static when the buffer has the capacity of keeping the whole relation into it.

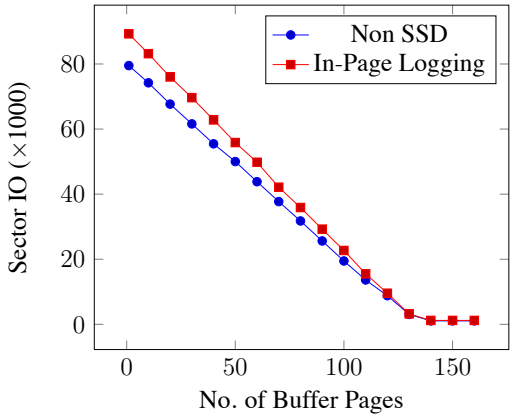


Figure 6.11: Random Block IO by Varying Buffer Size

6.5 Summary of the Evaluation

The idea of in-page logging is mainly to use fast read access to reduce expensive erase operations. Less erase operations implies also less read- and write operations, because an erase consists of first reading the whole erase unit, merging the pages and rewrite the erase unit.

Therefore it is a win-win situation concerning the read and write as well as erase counts. This phenomenon is shown mostly in hypothesis 1.1 (sequential insert) and 3 (random read/write), where we have about 45% less read-, and about 50% less write and erase counts. Due to the in-page logging approach cuts the erase counts in half, the estimated runtime gain of in-page logging comparing to a non-SSD approach is about 50%.

7 Summary / Conclusion

Conventional access methods and algorithms can be used without any modification due to the flash translation layer (FTL). Despite this possibility, this thesis showed, that the overall performance of a key-value store can significantly be increased with small changes in the implementation.

To show this increase, I implemented a key-value store and applied an approach called in-page logging (IPL). The approach reduces the number of expensive erase operations by introducing a log based system. Despite in-page logging utilizes only one positive ability of flash memory and disregards the software layers of the SSD, the evaluation showed that the increase of the overall performance is up to 50%.

As a future work, it would be interesting to investigate on how the approach performs with the use of more log sectors per page. It should be possible to reduce the number of erase operations even more by using more reads. Further, it could be analyzed, how the implementation performs on a real SSD.

Bibliography

[1] *Cache Algorithms - Least Recently Used*. http://en.wikipedia.org/wiki/Cache_algorithms#Least_Recently_Used, [15.01.2012]

[2] *Flash memory*. http://en.wikipedia.org/wiki/Flash_memory, [15.01.2012]

[3] *Hard Disk Drive*. http://en.wikipedia.org/wiki/Hard_disk_drive, [15.01.2012]

[4] *NoSQL*. http://en.wikipedia.org/wiki/NoSQL, [15.01.2012]

[5] *NoSQL Databases*. http://nosql-database.org/, [15.01.2012]

[6] *Oracle Berkeley DB*. http://www.oracle.com/technetwork/database/berkeleydb/overview/index.html, [15.01.2012]

[7] *Page Replacement Algorithm - Least Recently Used*. http://en.wikipedia.org/wiki/Page_replacement_algorithm#Least_recently_used, [15.01.2012]

[8] *Solid-State-Drive*. http://de.wikipedia.org/wiki/Solid-State-Drive, [15.01.2012]

[9] BREKLE, Jonas: *Seminararbeit über Key Value Stores, BigTable, Hadoop, CouchDB*. http://dbs.uni-leipzig.de/file/seminar0910_Brekle_Ausarbeitung.pdf, [15.01.2012]

[10] CHANG, Fay ; DEAN, Jeffrey ; GHEMAWAT, Sanjay ; HSIEH, Wilson C. ; WALLACH, Deborah A. ; BURROWS, Michael ; CHANDRA, Tushar ; FIKES, Andrew ; GRUBER, Robert: Bigtable: A Distributed Storage System for Structured Data (Awarded Best Paper!). In: *OSDI*, USENIX Association, 2006, S. 205–218

[11] DEBNATH, Biplob K. ; SENGUPTA, Sudipta ; LI, Jin: FlashStore: High Throughput Persistent Key-Value Store. In: *PVLDB* 3 (2010), Nr. 2, S. 1414–1425

[12] DEBNATH, Biplob K. ; SENGUPTA, Sudipta ; LI, Jin: SkimpyStash: RAM space skimpy key-value store on flash-based storage. In: SELLIS, Timos K. (Hrsg.) ; MILLER, Renée J. (Hrsg.) ; KEMENTSIETSIDIS, Anastasios (Hrsg.) ; VELEGRAKIS, Yannis (Hrsg.): *SIGMOD Conference*, ACM, 2011. – ISBN 978–1–4503–0661–4, S. 25–36

[13] DECANDIA, Giuseppe ; HASTORUN, Deniz ; JAMPANI, Madan ; KAKULAPATI, Gunavardhan ; LAKSHMAN, Avinash ; PILCHIN, Alex ; SIVASUBRAMANIAN, Swaminathan ; VOSSHALL, Peter ; VOGELS, Werner: Dynamo: amazon's highly available key-value store. In: *SIGOPS Oper. Syst. Rev.* 41 (2007), Oktober, 205–220. `http://dx.doi.org/http://doi.acm.org/10.1145/1323293.1294281`. – DOI http://doi.acm.org/10.1145/1323293.1294281. – ISSN 0163–5980

[14] INTEL: *Understanding the Flash Translation Layer (FTL) Specification*. Application Note AP-684. Intel Corporation, [December 1998]

[15] LEE, Sang-Won ; MOON, Bongki: Design of flash-based DBMS: an in-page logging approach. In: CHAN, Chee Y. (Hrsg.) ; OOI, Beng C. (Hrsg.) ; ZHOU, Aoying (Hrsg.): *SIGMOD Conference*, ACM, 2007. – ISBN 978–1–59593–686–8, S. 55–66

[16] NEAL EKKER, Jim H. Tom Coughlin C. Tom Coughlin: *An Introduction to Solid State Storage*. `http://www.snia.org.au/assets/documents/SSSI_Wht_Paper_Final.pdf`, [15.01.2012]

[17] ON, Sai T. ; LI, Yinan ; HE, Bingsheng ; WU, Ming ; LUO, Qiong ; XU, Jianliang: FD-buffer: a buffer manager for databases on flash disks. In: HUANG, Jimmy (Hrsg.) ; KOUDAS, Nick (Hrsg.) ; JONES, Gareth J. F. (Hrsg.) ; WU, Xindong (Hrsg.) ; COLLINS-THOMPSON, Kevyn (Hrsg.) ; AN, Aijun (Hrsg.): *CIKM*, ACM, 2010. – ISBN 978–1–4503–0099–5, S. 1297–1300

[18] STONEBRAKER, Michael: SQL databases v. NoSQL databases. In: *Commun. ACM* 53 (2010), April, 10–11. `http://dx.doi.org/http://doi.acm.org/10.1145/1721654.1721659`. – DOI http://doi.acm.org/10.1145/1721654.1721659. – ISSN 0001–0782