Indexing on Solid State Drives based on Flash Memory

Florian Keusch

MASTER’S THESIS

Systems Group
Department of Computer Science
ETH Zurich
http://www.systems.ethz.ch/

September 2008 - March 2009

Supervised by:
Dr. Jens Dittrich
Lukas Blunschi
Dr. Marcos Antonio Vaz Salles
Abstract

Indexing is a very important concept in database systems. Index structures are maintained to organize huge data collections in an efficient way. Many of these structures are tuned for the use on conventional hard drives.

In this thesis we focus on indexing on solid state drives based on flash memory. Solid state drives have other characteristics than conventional hard drives. They do not contain any mechanical parts and the underlying NAND flash memory confers them other performance characteristics. The access latency is much lower and the write performance is slowed by the flash memory limitations. We built two different index structures: a conventional B⁺Tree and an index structure based on CSS-trees with an in-memory delta. On the B⁺Tree both drives had a constant performance. On the CSS-tree hierarchy the throughput varied depending on the number of trees in the list. In the experiments our solid state drive reached a much higher throughput (at least by a factor of 3) on both index structures. Comparing the two indexes the solid state drive favored the delta indexing approach. Additionally one observes the huge progress of this new technology. The density grows, the costs fall and the performance increases. This shows that solid state drives have the potential to compete against high performance hard drives used in todays server architectures.

Keywords: Flash Memory, SSD, Solid State Drive, Solid State Disk, SSD vs HD, Hard Drive, Flash Memory Drive, NAND Flash, External Sorting, Index Structures, Performance Testing
# Contents

1 **Introduction** .................................................. 9  
   1.1 Market Development ........................................ 9  
   1.2 Why use a SSD? ............................................. 9  
   1.3 Questions .................................................. 11  
   1.4 Goals and outline ......................................... 12  

2 **Solid State Drive** .............................................. 13  
   2.1 Introduction ................................................ 13  
   2.2 Flash Memory ............................................... 15  
   2.3 Flash Translation Layer (FTL) ......................... 17  
      2.3.1 Controller ............................................ 17  
      2.3.2 Wear Leveling ....................................... 19  
   2.4 Performance Experiments ............................... 20  
      2.4.1 Test Environment ................................... 20  
      2.4.2 Sequential I/O ...................................... 21  
      2.4.3 Random I/O .......................................... 24  
      2.4.4 Access Latency ...................................... 28  
      2.4.5 Conclusions ......................................... 28  
   2.5 Storage Hierarchy ......................................... 29  
   2.6 Energy Efficiency ......................................... 29  
   2.7 File Systems and Data Structures .................... 30  
   2.8 Connecting Flash ......................................... 31  
   2.9 Summary ................................................... 32  

3 **External Sorting** ........................................... 33  
   3.1 Motivation ................................................ 33  
   3.2 Algorithm ............................................... 33  
      3.2.1 Description ......................................... 34  
      3.2.2 Pseudo-code ......................................... 37  
   3.3 Implementation ........................................... 38  
      3.3.1 Kernel Buffering .................................... 38  
      3.3.2 Java Object Overhead ............................... 38  
      3.3.3 Raw Device ......................................... 38  
      3.3.4 Class Diagram ....................................... 39  
      3.3.5 Implementation Description ....................... 39  
   3.4 Test Results ................................................ 41  
      3.4.1 Solid State Drive .................................... 43  
      3.4.2 Hard Drive .......................................... 44  
   3.5 SSD vs HD Strategy ....................................... 46
## C Source Code Overview

- C.1 Performance Experiments .................................................. 89
- C.2 External Sorting ............................................................... 89
- C.3 Index Structure ............................................................... 91
  - C.3.1 Event Generation ....................................................... 91
  - C.3.2 CSS-tree hierarchy ..................................................... 92
  - C.3.3 DiskBTree ................................................................. 93
  - C.3.4 Helper Classes .......................................................... 94
1 Introduction

In the past few years flash memory became more and more important. In many mobile devices like mobile phones, digital cameras, USB memory sticks and mp3 players, flash memory is used in small amounts for years. But as the price for flash memory is rapidly decreasing and the storage density of flash memory chips is growing, it becomes feasible to use flash memory even in notebooks, desktop computers and servers. It is now possible to construct devices containing an array of single flash chips to build a device such that its amount of memory is sufficient to use for main storage. Such a device is called a Solid State Drive (SSD).

1.1 Market Development

The market of flash memory is changing. The density of NAND Flash is increasing drastically. While flash memory chips can be made much smaller, their capacity doubles approximately every year (see Figure 1.1).

This development leads to a widespread usage of flash memory. The current maximum capacity of the newest solid state drives available on the market is 250GB. Today solid state drives are mainly used in notebooks. In the future they might even be used in personal computers and server architectures as the standard configuration.

Another interesting development is the cost of flash memory (see Figure 1.2). The price of flash memory is rapidly dropping. Every month flash memory devices or flash memory storage cards get cheaper. New products with larger capacity emerge on the market.

1.2 Why use a SSD?

Solid state drives appeared on the market just recently. What are the differences of this new technology compared to conventional hard drives? A solid state drive has no mechanical parts. It contains only flash memory chips (for detailed information see Chapter 2). This offers several advantages over a conventional hard drive:

- A solid state drive survives rougher conditions, it is more resistant to shock and can operate in a wider temperature range.
- Since there are no mechanical parts and no cooling fans a solid state drive does not make noise.
- The density of the flash memory chips can be very high. Therefore it is possible to produce solid state drives which are generally smaller and lighter than common hard drives.
10 Introduction

Figure 1.1: The evolution of the density of NAND flash memory [9].

- Solid state drives usually have a very low access latency which speeds up the start of applications and the overall booting time.

- Flash memory consumes less power than hard drives. That is one of the reasons why flash memory is often assembled in mobile devices.

On the other side solid state drives also have some flaws:

- One problem is their asynchronous read and write performance. Reading is fast but writing is slow.

- The problem with writing can be described as follows: internally, before a write (called programming) can occur, there has to be a free block. A block has to be erased to be free for writing. The erase operation clears the flash memory, but the clearing can only be executed with the granularity of a large block (erase unit). If the amount of data being written is small compared to the erase unit, then a correspondingly large penalty is incurred in writing the data.

- Another problem is that flash memory cells wear out after a certain number of erasures. Then this cell can no more be used.
To keep the device fully functional, a solid state drive has a controller which adds techniques (e.g. wear leveling) to manage the internal block usage.

Usage of flash memory has been discussed in many papers. In one famous talk *Tape is Dead, Disk is Tape, Flash is Disk* [18] it was discussed that flash memory could eventually emerge as a replacement of traditional hard drives. Therefore various people tried to adapt databases, algorithms and data structures to the new properties of flash memory. Using such tuned algorithms one can achieve better performance on flash devices.

### 1.3 Questions

Technology of flash memory is still improving. Its limits are not reached yet. We may ask one of the following questions:

- What are the limits of this fast emerging technology?
- Will flash memory be used in personal computers and server architectures?
- Could flash memory devices even replace mechanical disks completely?
- What are the full performance characteristics of flash memory devices?
- How does flash memory affect the performance of algorithms and index structures?
- Do we have to redesign software to take the maximum advantage of flash memory devices?
1.4 Goals and outline

While we can not answer many of the above questions yet, a goal of this thesis is to get a better understanding of flash memory in general and of solid state drives in particular. Then this knowledge can be used to build algorithms and index structures which better support the characteristics of solid state drives.

Solid state drives are relatively new storage devices. Their market is rapidly growing and their performance is improving every few months. Can they replace hard drives as the main storage medium? Currently the I/O performance of databases, data structures and many algorithms are optimized for hard drives, optimized to use sequential I/O patterns. Having a new storage media, that has new performance characteristics, many of these algorithms need to be adapted to perform better on a solid state drive.

Chapter 2, Solid State Drive, explores the technical details of a typical solid state drive and explains their assembling and functionality. As we will see, solid state drives are quite different compared to common hard drives. We will expose their advantages but also handicaps running some experiments.

In chapter 3, External Sorting, we implement external sorting and run the algorithm directly on top of the solid state drive and the hard drive. We will see how this algorithm performs on either device.

Now it is time to build a more complex index structure and let it run on both testing drives. Chapter 4, Index Structure, describes more complex index structures. The data is stored on the drives. We let it run on both devices to see the differences and gain a better understanding of solid state drives.

In the final chapter 5, Conclusions, we summarize the knowledge we gained in this thesis and describe how one could exploit the new characteristics of a solid state drive to build a faster index or data structure.
2 Solid State Drive

2.1 Introduction

Comparing a solid state drive with a traditional hard drive there is a big difference. A solid state drive does not contain any mechanical parts, it is a completely electronically device (see Figure 2.1). Traditional hard drives, which are used as the main storage devices since the eighties, have platters to store the data magnetically. An arm is mechanically moved to allow the drive to position the head to the right point on the platter. Then the head can read and write data from the spinning disk platters. The simplified cost model of an I/O request from a hard drive is:

\[ time_{I/O} = t_{\text{seek}} + t_{\text{rotation}} + t_{\text{transfer}} \]

\( t_{\text{seek}} \) is the seek time, the time to move the head to the correct position on the platter. Full seeks last about 8-9ms on average on traditional hard drives. One tries to avoid seeks whenever possible. When the new head position is near the old one, many hard drives can do faster mini seeks instead.

\( t_{\text{rotation}} \) is the rotational delay. After the head is moved to the correct position it has to wait until the platter has rotated to the start position. This delay depends on the rotational speed of the hard drive. Nowadays most standard hard drives have a rotational speed of 7200 RPM which results in an average rotational delay of 4.17ms.

\( t_{\text{transfer}} \) is the time to transfer the data from the platter to the hard drive cache. With a read performance of 120MB/s the drive can for example transfer 32KB data in 0.26ms.

Figure 2.1: The internal life of a hard drive (left): mechanical parts / solid state disk (right): flash memory chips [39].
On the contrary solid state drives do not need any mechanical parts. They are fully electronically devices and use solid state memory to store the data persistently. Two different storage chips are used: flash memory or SDRAM memory chips. In this thesis we only consider flash memory based solid state drives which are mostly used today.

Flash memory suffers from two limitations:

1. Before writing to a solid state drive the flash memory has to be erased. This flash memory can only be cleared by erasing a large block of memory called erase unit (for example 512KB depending on the model).

2. Each erase unit can only be erased a limited number of times (for example 100'000 times).

Flash memory can offer advantages over the traditional hard drive, but also has some flaws. A small comparison of advantages (+) and disadvantages (-) is listed below:

+ No noise: neither mechanical parts nor cooling fan.
+ Very fast read operation.
+ Faster startup: no spin-up required.
+ Nearly constant I/O performance over the whole drive. Hard drives have slightly different speeds depending on the platter region.
+ Smaller size and weight: solid state drives can be constructed much smaller and with a smaller weight than hard drives.
+ Low power consumption.
+ Rougher usage conditions: shock resistance, temperature range.
+ Very low read latency: faster boot and application start.
  - High price: much higher costs per GB than hard drives.
  - Much lower capacity than hard drives at the moment.
  - Asynchronous read/write speeds: write speed is much slower than read.
  - Erase operation can only erase a whole unit: slow write performance.
  - Flash memory cells wear out: special techniques (wear leveling) to ensure reliable functionality are needed.

Due to the limitations of flash memory a solid state drive needs special data structures and algorithms to achieve a reliable functionality. Today most flash memory devices contain a controller which optimizes pending operations and manages the flash memory. This makes it possible to address a flash memory device like any other block device.
This chapter provides more detailed information about flash memory. Then in multiple sections we discuss the different internal parts of a solid state drive. In section 2.3 we describe the flash translation layer and techniques like wear leveling which ensure the functionality of the solid state drive. In section 2.4 we present performance experiments. We test one solid state drive against a traditional hard drive to find out their characteristics. We shortly discuss the storage hierarchy 2.5 and the power consumption myth 2.6 of a solid state drive.

2.2 Flash Memory

Flash memory is a completely electronic device, it does not need any mechanically moving parts. It is a specific type of EEPROM that can be electrically erased and programmed in blocks. Flash memory is non-volatile memory. There are two different types of flash memory cells:

- NOR flash memory cells.
- NAND flash memory cells.

At the beginning of flash memory NOR flash memory was often used. It can be addressed by the processor directly and is handy small amounts of storage.

![Figure 2.2: A (NAND) flash memory chip [43].](image)

Today usually NAND flash memory is used to store the data. It offers a much higher density which is more suitable for large data amounts. The costs are lower and the endurance is much longer than NOR flash. NAND flash can only be addressed at the page level.

Flash memory can either come with single level cells (SLC) or multi level cells (MLC). The difference in these two cell models is that a SLC can only store 1 bit per cell (1 or 0), whereas MLC can store multiple bits (e.g. 00 or 01 or 10 or 11). Internally these values are managed by holding a different voltage level. Both flash memory cells are similar in their design. MLC
flash devices cost less and allow a higher storage density. Therefore in most mass productions MLC cells are used. SLC flash devices provide faster write performance and greater reliability. SLC flash cells are usually used in high performance storage solutions. Table 2.1 compares the two cell models.

<table>
<thead>
<tr>
<th></th>
<th>SLC</th>
<th>MLC</th>
</tr>
</thead>
<tbody>
<tr>
<td>High Density</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Low Cost per Bit</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Endurance</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Operating Temperature Range</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Low Power Consumption</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Write/Erase Speeds</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Write/Erase Endurance</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

Table 2.1: Short comparison: SLC vs MLC facts from [46].

Flash memory only allows two possible states:

- erased
- programmed

When a flash memory cell is in the erased status then its bits are all set to zero (or one, depending on the flash device). Only when a flash cell is in the erased mode, the controller can write to that cell. In this example this means the 0 can be set to 1. Now the cell is programmed and kind of frozen. It is not possible to simply change back the 1 to a 0 and write again. The flash memory cell has to be erased first (see example Figure 2.3). The even worse fact is that not only a small couple of cells can be erased. The erase operation has to be done on a much larger scale. It can only be done in the granularity of erase units which are e.g. 512KB. If the amount of data being written is small compared to the erase unit, then a correspondingly large penalty is incurred in writing the data.

Another point to consider is the write position (see Figure 2.4). The flash memory architecture is divided in blocks of flash memory. The smallest erasable block is called an erase unit. If the position of the written data overlaps two blocks, then both blocks have to be erased. However this erase operation must not be necessarily executed right before or after the write. The controller of the device might choose just a new block for the write request and update the internal addressing map.

The smallest addressable memory unit on a solid state drive (using NAND flash) is a page. Typically a page can be of 4KB flash memory. Having 128 pages, the next greater unit in the flash memory hierarchy is the erase unit with 512KB (Note: This can vary from drive to drive).
2.3 Flash Translation Layer (FTL)

Solid state drives are addressable like common block devices. But behind the scenes many different processes like wear leveling or erase operations occur. The Flash Translation Layer [25] is the interface that makes the flash memory appear like a conventional drive. Therefore it can be used as a normal block device. The concept of the FTL is implemented by the controller of the solid state drive. The layer tries to efficiently manage the read and write access to the underlying flash memory chips. It hides all the details from the user. So when writing to the solid state drive the user does not have to worry about free blocks and the erase operation. All the managing is done internally by the FTL. It provides a mechanism to ensure that writes are distributed uniformly across the media. This process is called wear leveling and prevents flash memory cells from wearing out.

2.3.1 Controller

The controller of a solid state drive manages all the internal processes to make the FTL work. It contains a mapping table that does the logical physical mapping. The logical address that comes from the request is mapped to the the physical address which points to the flash memory block where the data is in fact stored. Whenever a read or write request arrives in the solid state drive the logical block address (LBA) first has to be translated into the physical block address(PBA) (see Figure 2.5). The LBA is the block address used by the operating system to read or write a block of data on the flash drive. The PBA is the physical address of a block of data on the flash drive. Note that over time the PBA corresponding to one and the same LBA can change often.
The controller handles the wear leveling process (see next section). When a write request arrives at the solid state drive then a free block is selected, the data is written and the address translation table is updated. Internally the old block has not to be erased immediately. The controller could also choose to wait with the erasure and do a kind of garbage collection when the amount of free blocks falls below a certain limit. Or the controller may also wait until the drive is not busy. Certainly some data structures are used to maintain a free block list and to store the used blocks. In a flash memory block there is a little overhead memory where meta-data can be stored to help managing these structures. For example a counter stores how many times each block has already been erased.

Inside a solid state drive we would most certainly find a DRAM cache. The controller has access to this cache and can optimize requests. For example when many write requests are arriving it can try to avoid multiple writes to the same block and just write them in one step instead of each time using a new block. The question remains how the controller can reorganize certain requests to achieve the best or a good performance.
There are no real specifications of the internal life of a solid state drive. One can list structures and mechanisms that are really needed to do the work. But solid state drives are a relatively new technology. Every vendor tries to keep their knowledge top secret and hide the techniques that make their own drives better and faster than others. Therefore a solid state drive can also be seen as a black box. It just does the right thing, but it is not possible to clearly see all the internal details and intelligence. It is hidden from users and developers.

For such a SSD controller one can think of many optimizations. Using and increasing the DRAM cache and add more intelligent techniques to organize requests could make a huge difference. Prefetching data when sequential read patterns occur (like a conventional hard drive could fill its whole buffer) might speed up the reading process. A controller could also write to different flash chips in parallel (see Figure 2.7). Since all the parts are electronically in flash memory, parallelization might not be very hard to add. Flash memory can also be seen as many memory cells that are ordered in parallel. Using parallelization, the I/O requests, the erase process and the internal maintaining of the data structures get more complicated, but a much higher performance can be accomplished. One could even think of constructing a SSD containing several drives combined together as a RAID configuration inside.

![Figure 2.7: A sketch of parallelization in a solid state drive.](image)

### 2.3.2 Wear Leveling

A big limitation of flash memory is that it can only be erased a limited number of times. When an erase unit exceeds this number it can wear out. This means that this flash memory unit can no longer store data reliable. Internally it is marked as a bad block and is no longer used in future requests to store data. A typical mean time before failures (MTBF) is currently around 100000 erase cycles per erase unit (depending on the flash memory type).

But can we still use flash memory for reliable data storage? Yes. The FTL implements a wear leveling mechanism to overcome this problem. It ensures that the lifetime of the solid state
drive is prolonged. The flash memory units are shuffled. When data is updated and written back to the same logical memory address, this is not written to the same physical memory block. A new free memory block is taken from the free list and used to store the new data. Internally in the meta-data a counter is available that holds the number of erasures for each individual block. This gives the controller the possibility to uniformly use the memory blocks and prevent memory from wearing out. There are different methods of wear leveling. Static and dynamic wear leveling. Whereas static wear leveling applies its technique over the whole flash memory, dynamic wear leveling gives the possibilities to only wear-level over a specified region of flash memory. More information about wear leveling can be read in [8].

Some new flash memory devices even contain additional memory blocks that are kept as a reserve. The flash device as a whole will not see the extra memory it contains. Many calculations have shown that, using a wear leveling technique, a solid state drive can be used for many years without having problems, even when writing a lot of data. Example: writing each day 100GB sequentially on a 32GB solid state drive still ensures its lifetime for 75 years (calculation in [24] table 25).

2.4 Performance Experiments

In this section we do some basic performance measurements with our test drives. We measure sequential I/O and random I/O with different read/write block sizes. We use our own test software. The tested devices are one solid state drive from OCZ and a conventional hard drive from Western Digital. All the tests are performed on the same computer. With these tests we can better analyze the performance and characteristics of the solid state drive.

2.4.1 Test Environment

The test software is written in Java. Time is measured as the duration of a test run to complete the defined sequence of read/write requests. The measurement is done with the System.nanoTime() method. There is no file system in between. The data is read and written directly to the drive using the Java new I/O classes. Through a file system the characteristics might be a bit blurred. We wanted to access the drives as direct as possible with preferably no layers and no buffers in between.

The two devices we have to test and compare:

**HD**: Western Digital Hard Drive, 320GB, 7200RPM, 16MB Cache

**SSD**: OCZ Solid State Drive, 32GB, core series v1

In further tests we address the conventional hard drive with the name HD and the solid state drive with SSD.
2.4 Performance Experiments

The tests are running on a Debian Linux system. This computer uses 2GB Ram and two Intel Cores with 2.13GHz. The operating system is running on a separate hard drive. All the devices are connected via the SATA 2 computer bus interface.

For more information see the Appendix A, Test Environment, and C.1, Source Code Documentation.

2.4.2 Sequential I/O

This I/O pattern is typically used for conventional hard drives. The hard drive is very fast in sequentially reading and writing. Therefore many data structures and algorithms have been designed to support this pattern. In a solid state drive the data does not lie in a sequential order. Due to the wear leveling mechanism and the address translation a block with the logical id 1 and a block with the logical id 2 must not even lie on the same flash memory chip. The physical memory address is looked up in the address translation table the controller maintains.

A sequential I/O pattern starts at a specific position and then a lot of I/O requests are executed sequentially (see Figure 2.8).

![Figure 2.8: An example sequential I/O pattern.](image)

In our test we read/write 2GB of data sequentially. The time measurement is started at the beginning and stopped at the end of the test execution. At the end the average read or write performance is calculated in MB/s. With this measure we can compare our drives. We execute this test using different read/write chunk sizes to find out how the performance of the drives changes. Example: having a chunk size of 16KB. When we do the read test this means we read 16KB from the disk, then sequentially the next 16KB at the updated position. The application loops until the whole test size of 2GB is read.

On the HD the read speed seems to be constantly at 115MB/s. The read speed of the solid state drive also reaches this performance, but can even reach a higher speed up to 136MB/s choosing the right chunk-size of 8-16KB (see Figure 2.9(a)).

The write speed analysis seems a little bit more complicated. In Figure 2.9(b) we can clearly see that the hard drive outperforms the solid state drive when sequential writing. We discover the asynchronous performance when we compare the read and write of the solid state drive. Using a very small chunk-size below 4KB, both drives performance is very slow. In this experiment the hard drive reaches write speeds of 137MB/s, even a little bit higher than
its read speeds. On contrary on the solid state drive we can reach around 46MB/s at the maximum. In this graph some test values seem completely lost. Only the green line seems to show feasible results. We also chose chunk-sizes that are not aligned to integers. For example we test the following 3 test cases: (4KB-1B), 4KB, (4KB+1B). In the read graph this seems not to be of any influence. But the writing clearly suffers from unaligned write chunk sizes. In general we can say that a multiple of 4KB will have a good write performance on both drives.

We were a bit disappointed by the low write performance of the solid state drive although the vendor states that it can write 80-93MB/s. So we looked a bit deeper into their testing methods. To measure the performance they use the ATTO disk benchmark testing tool for Windows. This test is commonly used to benchmark hard drives. In their configuration they test sequential read and write performance using the following chunk-sizes: 16KB, 32KB, 64KB, 128KB, 256K, 512KB, 1MB, 2MB, 4MB, 8MB. And they also activated the flag ‘overlapped I/O’. So we reconstructed this ATTO test with our own software and let it run on our test drives. We could not exactly find out how they defined ‘overlapped I/O’ and how much I/O is overlapped (the manual is not open). So we let the experiments run with an overlapped I/O of factor 2 (ATTO(2)). This means that in the sequential read/write the second block will overlap the half data of the first block. Comparing normal I/O with overlapped I/O did not have any differences in the performance for the hard drive. It amazingly had only an impact on the solid state drive write performance (see Figure 2.10). We can see that the write performance sometimes is almost doubled. The performance of ATTO(2) varies from experiment run to experiment run, but ATTO(2) is at least 50% faster although the same amount of data has been written. Well in fact the same data has been written, but not in the same range on the drive. What does the solid state drive different in the ATTO(2) test? Since the write operation is a performance crucial operation on a solid state drive, the controller is looking for optimizations. In an overlapped writing scenario many flash block addresses would have been written and erased multiple times. We think that in this scenario the controller of the solid state drive can do some optimizations when writing and in fact it may not really write all the data.

Usually we will not gain any performance out of this characteristics. It looks only better. Many benchmark tools that exist have been developed to test the performance of conventional hard drives. Just applying them to a solid state drive may not show the whole truth about solid state drives.

Table 2.4.2 shows a short performance summary. We measured these values for our two devices by running the sequential I/O experiments.

<table>
<thead>
<tr>
<th></th>
<th>read</th>
<th>write</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSD</td>
<td>136MB/s</td>
<td>46MB/s</td>
</tr>
<tr>
<td>HD</td>
<td>115MB/s</td>
<td>137MB/s</td>
</tr>
</tbody>
</table>

Table 2.2: Short summary of the sequential I/O performance.
Figure 2.9: The sequential I/O experiment results of the read and write performance.
2.4.3 Random I/O

In the random I/O pattern (see Figure 2.11) the requests are done at random positions on the drive. Two requests may have a completely different logical address. The difference from sequential to random I/O lies in the access latency. In the sequential pattern the head of the hard drive is already at the right position and the data can just be read. On the contrary we have a complete new position in a random pattern. In a hard drive the head has to be moved to a new position. With every request a seek is done which accounts to the transfer time. Such random I/O patterns often occur when looking up single values in an index structure or when loading many configuration values at the system or application start.

Figure 2.11: An example random I/O pattern.
In this experiment we access 1000 times over the whole device and read/write a unit of different size. At the beginning of the experiment an array stores the positions where to read/write the units with a sufficiently large offset to overcome the hard drive buffer (16MB). Elements are taken randomly from that array to generate the random position array. Then in a loop the random position array is traversed and the units are read/written. Each time a new unit is read/written at a new random position, a seek has to be done. This takes some time (see Section 2.4.4 for the access latency measurements). At the end of the experiment the overall speed in MB/s is calculated. This value is taken to compare the two drives at different unit sizes.

In Figure 2.12 we can see the experiment results. In Figure 2.13 are the same results in log scale. In random reading the solid state drive is very fast. This is also because of its very small access latency. Figure 2.14(a) shows the direct ratio. For small read unit sizes the solid state drive clearly outperforms the hard drive. The hard drive does not have a bad read speed in general, but the much higher access latency, that comes with each random access, makes the hard drive slow. The ratio decreases as the read unit size increases. Then the reading of the unit sizes gets more and more sequential and the access latency does not have such a huge weight. This can be seen in Figure 2.13 on the top right.
When we look at the write speed then we have another situation. The solid state drive still has a very low access latency, but it also has a much lower write speed. Why? The explanation lies in the earlier mentioned erase problem. The solid state drive suffers from erasing many large blocks while writing only small data. Let us consider the following example: to write 4KB of data to an erase unit the solid state drive needs to have a free unit. Then it reads the old unit, updates its data in the cache and writes the whole data of the unit to the new unit. Then internally it has to update the data structures. The random write performance for small data is therefore very bad compared to the hard drive. Update in place will cost a lot. Figure 2.14(b) shows this direct comparison.
Figure 2.14: The ratio SSD vs HD for reading and writing.
2.4.4 Access Latency

The access latency or the seek time is the delay that occurs until the device is ready. Having a traditional hard drive this means moving the arm to the right position so that the head is on the track where the desired data is stored. In flash memory there also exists a such delay. The controller has to wait until the flash memory block is ready to read or write.

In an experiment we measured this access latency. We randomly do seeks over the whole device. The access latencies we measured were consistent with the vendor specification. The HD has a seek time of nearly 9ms and the SSD a latency of 0.3 ms. This is in fact a huge difference. Accessing the data a hard drive needs around 25 times longer than a solid state drive would need! (Comparing our two test devices, but in general you will measure similar differences testing other drives.)

Note that the solid state drive has the same constant latency over the whole flash memory. In contrary the latency of a conventional hard drive varies very much. The hard drive has to move its head to the correct position before doing an I/O operation. Depending on where the head currently lies it may not have to do a long way. Instead it could do a mini seek, which would be much faster than the average seek time. To improve their performance hard drives often use an internal cache (our model has a 16MB cache). When reading data from near positions the hard drive may already have loaded the data into the cache and read it directly from there. Then the costs are only the time to transfer the data from the hard drive cache to the system over the SATA bus interface, which is very fast.

2.4.5 Conclusions

In our experiments we have seen that there are differences between conventional hard drives and the new solid state drives. The big difference lies in the access latency. The solid state drive has a very small access latency compared to a conventional hard drive. This speeds up random I/O. Solid state drives are very fast compared to hard drives doing small random read requests. The more sequential the requests, the more the hard drive catches up. When doing a full sequential reading our solid state drive performance is comparable to the hard drive performance.

Regarding write performance our solid state drive cannot compete with the hard drive. Even for small chunk sizes the hard drive outperforms the solid state drive although the SSD has a very small access latency. This is due to the limitations of flash memory. When randomly writing small chunks a large penalty is incurred in writing the data due to the erase problem. The more sequential data to read/write the better its speed gets. The hard drive has a very constant performance and a simple scheme. First it moves the head and then it reads/writes with full speed. The controller of the solid state drive instead has to overcome the limitations of the flash memory as mentioned earlier and can thus gain a varying performance.

Having these measurements we can see the strategies to use with a solid state drive. Access to the drive is very cheap compared to the hard drive. Small reads or index lookups will not cost very much. Instead we definitely have to avoid small writes. This is where the solid state drive suffers most. When we write something to the solid state drive we should write large chunks. We should collect some data and then write it to the drive.
2.5 Storage Hierarchy

When flash memory was relatively young and there were no solid state drives on the market, there was a discussion of how flash memory could be used in the storage hierarchy (see for example [1]). The traditional storage hierarchy of personal computers and server architectures contains the components: Cache, RAM and External Memory (see Figure 2.15).

As flash memory was more and more used in mobile devices, the idea came up to use it also in common computers. Because of its low access latency some people proposed to use flash memory in between the RAM and the external storage device (hard drive). Some vendors started producing hybrid drives (storage drives where a conventional hard drive is enhanced with a small flash memory as a buffer). In the meantime there is no serious market for those devices. The picture of the storage hierarchy did not really change. Nowadays a flash memory drive can be realized with a feasible amount of data in the form of a solid state drive. Solid state drives currently coexist with hard drives in the external memory layer of the storage hierarchy. Some notebooks already use solid state drives, but the market for personal computers and server architectures is not there yet. This could certainly change in the future as the development of flash memory drives makes serious progress in performance.

![Diagram of storage hierarchy](image)

Figure 2.15: Will flash memory change the storage hierarchy?

2.6 Energy Efficiency

Power consumption becomes more and more an interesting discussion point. Nowadays there are many huge server architectures that run 24 hours a day and consume quite an amount of power. Flash memory is said to have a low power consumption. Many mobile devices rely on flash memory to store their data. As flash memory can be used nowadays in the form of solid
state drives, is there a new era in terms of energy efficiency when using the new solid state drives?

Yes and no. This point is not very clear. An advantage of flash memory is that it has a low power consumption. All the vendors of solid state drives, which also grow in numbers, present their products as low power consumption solutions and even state that one could save a lot of energy using solid state drives instead of conventional hard drives. There are so many different drives, it is not that easy to find a general conclusion. Yes, compared to a standard 3.5 inch server hard drive, that is tuned for pure performance, a solid state drive may consume much less power. But compared to the state of the art notebook 2.5 inch hard drive the solid state drive may not win this battle. In the past years notebook hard drives have made a lot of progress in terms of power consumption. The one advantage of a solid state drive over a good hard drive in terms of power consumption is that the operation power consumption is lower. On the contrary several measures showed that the idle power consumption can be much lower in a hard drive. So we could surely find a workload where the hard drive wins and others where the solid state drive can show a better energy efficiency. Newer high performance solid state drives often contain an additional amount of DRAM cache which will also use some additional power. Another point to add is that for hard drives there were many improvements and research in terms of power consumptions in the last 20 years. But solid state drives are a relatively new technology. In future years we might also have their power consumption lowered by many tricks. The general power consumption comparison is not that easy to do. No general statement can be found. Every hard drive or solid state drive has a slightly different power consumption. See the critical article from Tom’s Hardware [47] for more information.

It always depends on what we compare. There are so many different drives. Comparing the energy efficiency of solid state drives with notebook hard drives might not be what we want. For most server architectures and personal computers a good performance is the first constraint. Consuming less energy would be nice. Replacing the high performance hard drives by new solid state drives may reduce the power consumption by an amazing factor. And still the performance is good. There are not much studies and tests in this direction. But as the performance and density of solid state drives rise one might consider replacing the conventional hard drives.

2.7 File Systems and Data Structures

In the past years a lot of research has been done to build file systems and data structures that are aware of their underlying flash memory.

There are many different file systems that were optimized for the use on top of flash memory. JFFS [48], YAFFS, TrueFFS, exFAT just to name some famous ones. These file systems came up because flash memory has other properties than conventional data storage. Using a traditional file system over a flash memory block device is not optimal. They are not aware of the erasure problem of flash memory. Traditional file systems are highly optimized for conventional hard drives avoiding disk seeks. But flash memory has a very low access latency, so it does not suffer doing many seeks. Flash memory cells wear out after a certain number
of erasures. A flash file system tries to overcome all these problems. It uses log-structured approaches to implement the desirable properties of a flash file system. As a fact such flash file system cannot be efficiently used on a solid state drive. The solid state drive already contains a flash translation layer, the controller, that manages the flash memory. Such flash file systems would be used for Memory Technology Devices (MTD). These are flash memory storage devices which do not have a controller. All flash memory cards, USB flash memory sticks and solid state drives already have a controller which manages the wear leveling, caching, error correction and maintaining the data structures. Therefore they would not benefit from the usage of a specific flash file system.

Many papers try to find log based data structures and algorithms to improve the flash memory performance. "Algorithms and Data Structures for Flash Memories" [15] give a good overview of flash file systems and describes structures that support not-in-place updates on which log-based data structures rely. "FlashDB" [38] describes a database that uses NAND flash memory on sensor networks. In [50, 49, 26] different tree structures are presented to reach better performance on underlying flash memory storage. All these data structures could be applied again to a MTD or inside a solid state disk controller.

2.8 Connecting Flash

Actually most solid state drives on the market use the SATA computer bus interface. But there are other approaches, see for example the ioDrive [14] (Figure 2.16). It uses the PCI-Express bus interface. Its specification sheet announces a very fast performance compared to nowadays available solid state drives. The ioDrive is not on the market yet and may be not feasible to use for mainstream computers in the near future.

Figure 2.16: The ioDrive [14] from Iofusion uses the PCI-Express computer bus interface.
Another possibility would be to integrate flash memory chips directly on the main board. The flash memory would not be exchangeable that easy. But still when this setup would be reliable and fast, then most users would not care. They could still add additional drives using the common bus interfaces.

Clearly when the technology advances it may come a time when the drives are so fast that normal SATA bus interfaces are not fast enough to exploit the full speed of a flash memory drive. In some years these limits may be reached. We will have to think about other setups or construct faster bus interfaces.

2.9 Summary

In this chapter we have seen how flash memory can be used as a storage device. We described the problems that occur when writing to a flash device. Wear leveling techniques have to be applied because the number of erasures of a flash memory cell is limited. These erasures can only be executed in a relatively high granularity (size of an erase unit). In our performance experiments we have seen that a solid state drive has a very low access latency compared to a conventional drive. Therefore the solid state drive has an enormous advantage when doing small random read requests. The performance of a solid state drive is asynchronous. The write speed is much slower. Even in small random writes the solid state drive can not gain advantage of its low access latency due to the erase problem.

Using flash memory packed in a solid state drive is a new technology. Nowadays the performance of these drives is nice, but as this technology grows it might still increase a lot in the next years.

For our testing we had a main stream solid state drive from OCZ. In the mean time there are a lot of high performance solid state drives from different vendors that have performance specifications high above ours. The technology grows fast, many progress is done improving flash memory. And as the density increases also the cost of flash memory per GB decreases every month. Nowadays it is not possible to build very large solid state drives containing terabytes of storage. But still solid state drives gain more and more interest to use in mobile devices or even in personal computers.
3 External Sorting

3.1 Motivation

External sorting is an important algorithm for database servers. It is used whenever the data to sort exceeds the available main memory capacity. Many query processing algorithms and database servers rely on an external sorting algorithm. For example 77GB of data needs to be sorted on a machine with 1.5GB available main memory. An external sorting algorithm typically divides the input data into smaller runs (such that one run fits into main memory), sorts the runs and merges them iteratively into one single sorted run. In the end there is only 1 run left which contains all the data sorted. External sorting can sort a huge amount of data. Its sorting time is limited by the I/O performance of the drive that is used for the sorting. The sorting and merging in memory is very fast. In this algorithm there occur two typical I/O patterns:

- sequential writes of sorted runs
- random reads when merging the runs

Solid state drive have other characteristics than conventional hard drives. In this chapter we optimize the external sorting algorithm for solid state drives. Can we use the same strategy or is there a faster method to do the external sorting process?

In Section 3.2 we describe the general external sort algorithm. Then in 3.3 we describe our implementation and problems that occurred. We present the test results in 3.4 and describe the strategies for an external sorting algorithm on both drives.

3.2 Algorithm

In this section we describe the general external sorting algorithm [16]. It is quite simple to understand. Since external sorting is an important algorithm there was a lot of research in this area. Many people tried to optimize this algorithm in many ways. We use the simple standard algorithm for our purpose. Optimizations could be added later. Our goal is to see the differences of the sorting on both drives, solid state drive vs hard drive.
In the external sorting algorithm we define the following vocabulary:

**run**: A run is sequential data on the drive. It can be sorted.

**runSize**: is the size of a run.

**queue**: A queue wraps a run using a buffer. It allows to access data element-wise. When all the data of the underlying run was processed the queue is empty.

**fanIn**: is the maximum number of runs that the algorithm is allowed to merge in one step into a larger run.

**writeBuffer**: is the buffer the algorithm uses to write to the drive.

**readBuffer**: is a buffer the algorithm uses to read from the drive. Each queue can have a readBuffer. The sizes of readBuffer and writeBuffer are not necessarily equal and could also vary in each merge phase.

**merge phase**: is the process called which merges runs into one larger run. In a merge phase at most number of fanIn runs are merged. At the end of the merge phase there is one larger run (with the size of all the merged runs) lying on the drive.

**online merge**: The last merge may be done in an online way. Then in the last merge phase the data is not written to the drive but directly passed on to the next operator. At the end of the external sort algorithm the data is not lying sorted on the physical drive anywhere. It was used online.

**element**: An element is a record usually consisting of a key and a value. Since the values are not relevant in our experiment we only have keys to sort, which are realized using the primitive type int. One could easily add a value (or more) to the key defining a tupleSize (we use a tupleSize of 1), but it is not necessary for our experiments.

**rawPath**: is the path where the raw device is located (e.g /dev/raw/raw0).

**merge level**: The merge level is the true fanIn that was used for the merge phase. For example when 64 runs are sorted with a fanIn of 16 this results in the merge levels 16 4. This means there are two merge levels.

Note: when we talk about sizes in our algorithm this can either be in elements (records) or in bytes.

### 3.2.1 Description

We assume that the data to sort lies somewhere on the drive written in one large sequential unsorted run (our test software first generates random data and writes it to the drive). The position of this run on the drive is known. A simple external sorting algorithm consists of the following three phases (see Figure 3.1):
3.2 Algorithm

1. run generation phase
2. multilevel merge phase
3. final merge phase

These 3 phases contain the following steps:

run generation

Figure 3.2 describes the first phase of the external sorting, the run generation. At the beginning of the external sorting all the data lies sequentially unsorted on the drive. In this phase data is divided into pieces. Data in the amount of runSize is read from the drive, sorted in-memory and written back to the same location on the drive. The sorting in-memory is very fast. Mostly a quick-sort algorithm is taken to do this in-memory sorting. The algorithm loops until all the data is on the drive in sorted runs. The last run may be smaller because the data is not dividable by the runSize without a remainder. To the overall sorting algorithm this does not matter. This phase simply consists of: read, then sort, then write back. For the read- and writeBuffer the same storage can be used. A run is defined by its position (where its data lies on the drive) and its size. The run generation phase has only sequential I/O patterns. A large sequential data block is read, sorted in main memory and written back to the same location.
**multilevel merge**

The multilevel merge phase is sketched in Figure 3.3. It merges runs into bigger sorted runs until the specified fanIn is achieved and the final merge can be done. Let $n$ be the fanIn. So in this phase always the smallest $n$ runs are taken and merged. This process is stopped when there are at most $n$ runs left. Then the algorithm is ready for the final merge. Each run is represented as a queue using a buffer to access the data element-wise. For every queue to merge there is a readBuffer, for the merged queue a writeBuffer is needed. In this phase many disk accesses are random. Whenever the readBuffer of a queue is empty a random access is performed and it is filled again. When the writeBuffer of the merged queue is full, it is flushed to the drive. The merged queue needs to be written in a new position. It cannot overwrite the data from the runs to merge since it could overwrite yet unmerged data.

![Figure 3.3: In this example one multilevel merge is performed. The fanIn is 8.](image)

**final merge**

Figure 3.4 describes the final merge step which simply merges the runs that are left. Depending on the implementation this final run can be online or the data can be written back to the drive. An online final merge might pass the sorted data to an operator for further processing. In the second case the data lies on the drive as a sorted run at a known position.

![Figure 3.4: The final merge: merge all the runs that are left.](image)
3.2.2 Pseudo-code

The sorting algorithm sketched in pseudo-code is shown in the Listing 3.1.

```java
// the data to sort
Run data;
// parameters
int fanIn;
int runSize;

// the actual available runs
Run[] runs;

// 1. run generation phase
while (data.elements != 0) {
    tmp = data.remove(runSize);  // get new data
    tmp.sort();  // sort the data
    Run newRun = new Run(newPosition);  // create new run
    newRun.add(tmp);  // write back
    runs.add(newRun);
}

// 2. multilevel merge phase
while (runs.size() > fanIn) {
    Run[] runsToMerge = runs.get(fanIn);  // get the runs
    Run mergedRun = new Run(newPosition);  // create new run
    while (runsToMerge.hasNext()) {  // merge the runs
        mergedRun.add(runsToMerge.getNext());
    }
    runs.add(mergedRun);  // add the new run
}

// 3. final merge phase
while (runs.hasNext()) {
    runs.getNext();  // get next sorted element
    // store element or use it online
}
```

Listing 3.1: Pseudo-code of the external sorting algorithm.
3.3 Implementation

All our software is implemented in Java. For more information about the test machine see A.1. In our implementation we had to face different issues:

3.3.1 Kernel Buffering

While testing our drives we had to avoid the kernel buffering. This was very annoying and not straightforward. As the Linux I/O kernel buffer would fill the RAM and do its reads directly from there measuring I/O performance would be impossible. We finally managed to avoid the kernel buffering by mounting the devices with the Linux raw interface (see B.2) as block devices. Using the Java new I/O classes we can write directly to these without going over the Linux I/O kernel buffer. Our solution is working but it is not very nice. The Linux raw interface is deprecated and might be removed in future kernel versions. We also tried the Java NIO classes to overcome the kernel buffering (see B.1) and directly allocate a ByteBuffer, but we did not succeed. There was still buffering between our application and the drives. We found many documentations that described to use the O_DIRECT flag when using the C programming language, but for Java there is no simple and clear way to do this.

3.3.2 Java Object Overhead

The Java object orientation can have quite a big overhead. First we tried to implement a generic external sorting algorithm. But having these Integer objects instead of primitive type int and the conversion between Integer[] and int[] was just too much overhead. According to the specifications an Integer object uses at least 16 bytes on a 32 bit machine (and even more on a 64 bit machine). Instead the primitive data type int uses only 4 byte for its representation. Hence the used memory space can be reduced by a factor 4 using the primitive type instead of objects. Creating objects can be a big overhead and additionally uses much CPU. Our first implementation was just too much CPU-bound, the I/O did not really have an impact. To measure differences for our drives the implementation has to be I/O-bound. This means that the system is waiting for I/O the most of the time. So we switched the implementation to use primitive data types and reused arrays wherever possible. The implementation gets more specialized but the Java object overhead can be reduced.

3.3.3 Raw Device

Our implementation is based on raw devices. We directly do the I/O operation on the drive. This means we also have the limitations of a block device:

- The I/O can only be performed in multiples of 512 bytes.
- The I/O has to be aligned to 512 bytes.

(see Section B.2 for more information).
3.3 Implementation

3.3.4 Class Diagram

Figure 3.5 sketches a simplified class diagram of our implementation. For a description of the classes see the next subsection or the Appendix Source Code Documentation C.

![Class Diagram](image)

Figure 3.5: The simplified class diagram of our external sorting algorithm.

3.3.5 Implementation Description

Beginning at the top of the implementation we can define a certain test case. In the class diagram this test case has the name TestSorting.java. All these test cases extend the abstract class AbstractTest.java which holds all the necessary parameters. When our test case is executed with a shell script it reads some parameters (rawPath, runSize[KB], fanIn, testSize[MB], (readBufferSize[KB], writeBufferSize[KB])) from the arguments and initializes the test settings. Then it instantiates a tester class ExternalIntArrayMemorySortTester.java which will execute all the tests. This tester is implemented in different versions:
**ExternalIntArrayMemorySortTester.java:** Standard tester. Does an online final merge.

**ExternalIntArrayMemorySortTesterWithFinalWrite.java:** This tester writes the result of the final merge back to the drive. Sometimes it makes more sense to do this or the sorted data is needed on the drive. Then this tester can be used.

**ExternalIntArrayMemorySortTesterMultiDisk.java:** This tester uses multiple disks to do the sorting. More description can be found in the Section 3.7.

The tester also creates the test data. It randomly generates the data and writes it to the specified device. This data is then available in one large run for further processing. Then it creates an instance of the ExternalIntArrayMemorySorter. This class manages all the sorting process. First it executes the run generation. The large run is divided into smaller runs according to the defined runSize. Then it creates an instance of the multilevel merger and starts it. When the multilevel merge is finished, it has the last remaining runs ready for the final merge. It performs the final merge and gives back the integers element-wise through its public methods hasNext() and getNext().

The IntArrayMultiLevelMerger does the multilevel merge phase. It collects 'fanIn' number of runs and merges them until we have less or equal than 'fanIn' number of runs left. In the merge step some helper classes are needed.

The class ExternalIntArrayQueue represents one run on the drive. It contains a position that indicates where the run is lying on the drive and the number of elements. Furthermore the rawPath is needed and a persister that is given when the run is created. The persister is implemented in the NIOIntArrayPersister.java class. It allows only to read and write in arrays of the primitive type int. Every run uses the same instance of this persister, but has its own file channel. The file channel is given to the persister each time a read or a write operation is performed. An integer array is passed down to avoid creating it many times.

The ExternalIntArrayQueue class only offers to read and write in int[]. Therefore we need another wrapper class is needed, IntArrayWrapperQueue. From the underlying run it can read a whole array of the data type int and store it in a buffer. Then the data can be consumed element-wise. This is needed for the merging process. When the buffer is empty, the next elements can be fetched until there are none left. This class also implements the comparable interface. The one object that has the smaller element in the actual buffer position is considered smaller.

The class IntMerger.java is used to do the merging process. All queues are stored in a PriorityQueue data structure. The merger can just take the top queue, take the actual element out, advise the queue to calculate the next element and put the queue again into the PriorityQueue list. The IntMerger gives back the elements of the queues sorted with its methods hasNext() and getNext().

Many of the involved classes are implemented as operators. Before they can be used they have to be initiated with the method open(). Then buffer parameters are set and buffers may be filled to be ready for further processing. When the operator is finished and fully used the client has to call the method close(). Then remaining buffers may be flushed and the files are
3.4 Test Results

The implementation of our external sorting algorithm was not straight forward. In many stages of a computer system buffering mechanisms are implemented. It needed some time to find a solution to overcome the kernel buffering. Still the hard drive contains an internal buffer of 16MB. The solid state drive might also have a buffer, but it is not specified. In some experiments such buffers may distort the results.

First three different experiments are executed. The data size, the runSize and the fanIn are varied. These are three parameters that can be set in the application. We tested both drives and wanted to find out how these three parameters influence the overall sorting process.

In Figure 3.6(a) the data size was varied from 256KB to 2GB. The other parameters are fixed to their standard configuration (fanIn = 16, runSize = 256KB). Varying the data size does not have any influence on the external sorting process. The time is constantly growing the more data to be sorted.

In Figure 3.6(b) the runSize was varied from 32KB to 8MB. We sort 256MB of data and have a standard fanIn of 16. In this setup the hard drive is slightly faster. Increasing the runSize with the same fanIn and data size means that another multilevel merge configuration may result. With a larger runSize more data can be presorted in main memory.

The third experiment (see Figure 3.6(c)) varied the fanIn with constant data size 256MB and runSize 256KB. In this picture a big difference can be observed. The hard drive suffers from a too big fanIn, whereas the solid state drive does not seem to have problems with a large fanIn.

In the next section we try to find the optimal parameters for each drive by varying one parameter and analyzing how the overall external sorting process is influenced by its change. Then we choose this parameter fixed and try to find better configurations for the next parameter. Some of the test results are listed in tables.

The choice of the fanIn influences the multilevel merge process. Let’s assume 64 runs are on the drive to be sorted. With a fanIn of 64 or higher, no multilevel merge is performed. Hard drives need the multilevel merge step to avoid too much randomness. It is cheaper to have a smaller fanIn of 8 and do a multilevel merge, write the larger runs to the drive, compared
Figure 3.6: Basic testing. Vary different parameters and see their influence.
3.4 Test Results

to the fanIn 64. In all these tests the data is written back to the drive at the end. Then the data lies on the drive as one large sorted run.

3.4.1 Solid State Drive

We vary the parameters on the solid state drive to see the behavior of the overall sorting process and find a good parameter set.

**FanIn**

512MB data, fanIn X, readBuffer 32KB, writeBuffer 8MB, runSize 8MB

<table>
<thead>
<tr>
<th>fanIn</th>
<th>time</th>
<th>2 multilevel merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2m25</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1m54</td>
<td>1 multilevel merge</td>
</tr>
<tr>
<td>16</td>
<td>1m55</td>
<td>“</td>
</tr>
<tr>
<td>32</td>
<td>1m54</td>
<td>“</td>
</tr>
<tr>
<td>64</td>
<td>1m27</td>
<td>no multilevel merge</td>
</tr>
<tr>
<td>128</td>
<td>1m27</td>
<td>“</td>
</tr>
</tbody>
</table>

For the solid state drive it seems that no multilevel merge step is needed to provide the best sort results. Instead the fanIn can be chosen high to directly merge all the runs at once. Here only the number of multilevel merges counts. Two test cases with the same multilevel merge achieve the same execution time. Skipping the multilevel merge step results in the best performance.

**ReadBuffer**

512MB data, fanIn X, readBuffer X, writeBuffer 8MB, runSize 8MB

<table>
<thead>
<tr>
<th>readBuffer[KB]</th>
<th>sorting time</th>
<th>fanIn 64</th>
<th>fanIn 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1m58</td>
<td>2m57</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1m40</td>
<td>2m23</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>1m32</td>
<td>2m08</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1m27</td>
<td>1m54</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>1m25</td>
<td>1m49</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>1m20</td>
<td>1m45</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>1m30</td>
<td>1m46</td>
<td></td>
</tr>
<tr>
<td>512</td>
<td>1m30</td>
<td>1m43</td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>1m29</td>
<td>1m41</td>
<td></td>
</tr>
</tbody>
</table>

The readBuffer should be too small. If it is below 32KB then the overall sorting process gets much slower. When the fanIn is 64 and no multilevel merge is done a readBuffer larger
than 128KB also makes the sorting process slower. But having a fanIn of 16 and a multilevel merge, then a larger readBuffer still decreases the sorting time even if it is not that much.

**WriteBuffer**

512MB data, fanIn X, readBuffer 32KB, writeBuffer X, runSize 8MB

<table>
<thead>
<tr>
<th>writeBuffer[KB]</th>
<th>sorting time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>fanIn 64</td>
</tr>
<tr>
<td>4</td>
<td>2m17</td>
</tr>
<tr>
<td>8</td>
<td>1m58</td>
</tr>
<tr>
<td>16</td>
<td>1m47</td>
</tr>
<tr>
<td>32</td>
<td>1m39</td>
</tr>
<tr>
<td>64</td>
<td>1m31</td>
</tr>
<tr>
<td>128</td>
<td>1m29</td>
</tr>
<tr>
<td>256</td>
<td>1m27</td>
</tr>
<tr>
<td>512</td>
<td>1m27</td>
</tr>
<tr>
<td>1025</td>
<td>1m28</td>
</tr>
<tr>
<td>2048</td>
<td>1m27</td>
</tr>
<tr>
<td>4096</td>
<td>1m27</td>
</tr>
<tr>
<td>8192</td>
<td>1m26</td>
</tr>
<tr>
<td>16384</td>
<td>1m26</td>
</tr>
<tr>
<td>32768</td>
<td>1m26</td>
</tr>
</tbody>
</table>

When the parameters are set to do no multilevel merge, then the writeBuffer does not make a big difference. When there is a multilevel merge then the writeBuffer should be chosen higher. The higher, the faster the sorting process, more data can be written sequentially. But it is not such a huge difference, but more memory is consumed.

**RunSize**

512MB data, fanIn 64, readBuffer 32, writeBuffer 8MB, runSize X

<table>
<thead>
<tr>
<th>runSize[MB]</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1m27</td>
</tr>
<tr>
<td>16</td>
<td>1m22</td>
</tr>
<tr>
<td>32</td>
<td>1m22</td>
</tr>
</tbody>
</table>

The larger the runSize, the more can be presorted in main memory, the faster the overall sorting process? But here it does not have a lot of influence.

### 3.4.2 Hard Drive

We vary the parameters on the hard drive to see the behavior of the overall sorting process and find a good parameter set.
3.4 Test Results

**FanIn**

512MB data, fanIn X, readBuffer 32KB, writeBuffer 8MB, runSize 8MB

<table>
<thead>
<tr>
<th>fanIn</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1m51</td>
</tr>
<tr>
<td>8</td>
<td>1m39</td>
</tr>
<tr>
<td>16</td>
<td>1m37</td>
</tr>
<tr>
<td>32</td>
<td>2m08</td>
</tr>
<tr>
<td>64</td>
<td>2m22</td>
</tr>
<tr>
<td>128</td>
<td>2m24</td>
</tr>
</tbody>
</table>

Here there are two points to consider:

1. Many merge levels (because of a very low fanIn) cause too much reading/writing and slow the sorting process.

2. Having a too high fanIn slows the hard drive. Many random accesses have to be done. The best result is reached with the fanIn 16, but fanIn 8 also shows a good result.

The sorting time does not only depend on the number of merge levels it also kind of depends on the fanIn configurations. With the levels 32 2 the sorting takes more time than with the levels 16 4. This is because in the final merge step of the levels 32 2 there is more randomness. The final merge step only contains two queues. To explain the merge levels in more details consider the merge levels 16 4. These numbers correspond to the fanIn that was used in a multilevel merge step. In the experiment at the beginning there are 64 sorted runs with a size of 8MB. Then one multilevel merge step is performed with the fanIn 16. At this point there are 4 larger runs of the size 128MB. Then the final merge is done with the fanIn 4.

**ReadBuffer**

512MB data, fanIn X, readBuffer X, writeBuffer 8MB, runSize 8MB

<table>
<thead>
<tr>
<th>readBuffer[KB]</th>
<th>sorting time</th>
<th>fanIn 64</th>
<th>fanIn 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>3m22</td>
<td>1m43</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>2m23</td>
<td>1m37</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>1m53</td>
<td>1m34</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>1m43</td>
<td>1m35</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>1m35</td>
<td>1m34</td>
<td></td>
</tr>
<tr>
<td>512</td>
<td>1m29</td>
<td>1m37</td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>1m31</td>
<td>1m34</td>
<td></td>
</tr>
</tbody>
</table>

The readBuffer plays a huge role for the overall sorting time. The larger the readBuffer the faster the execution time? But we also need much more memory. The total size of the memory for all the readBuffers is 'readBufferSize * fanIn'. Having a larger readBuffer means that more data can be stored and that the drive does have to do less seeks which are very expensive on a hard drive.
**WriteBuffer**

512MB data, fanIn 16, readBuffer 32KB, writeBuffer X, runSize 8MB

<table>
<thead>
<tr>
<th>writeBuffer[KB]</th>
<th>fanIn 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>1m51</td>
</tr>
<tr>
<td>64</td>
<td>1m43</td>
</tr>
<tr>
<td>128</td>
<td>1m39</td>
</tr>
<tr>
<td>256</td>
<td>1m45</td>
</tr>
<tr>
<td>512</td>
<td>1m38</td>
</tr>
<tr>
<td>1024</td>
<td>1m41</td>
</tr>
<tr>
<td>2048</td>
<td>1m38</td>
</tr>
<tr>
<td>4096</td>
<td>1m38</td>
</tr>
<tr>
<td>8192</td>
<td>1m37</td>
</tr>
<tr>
<td>16384</td>
<td>1m37</td>
</tr>
<tr>
<td>32768</td>
<td>1m37</td>
</tr>
</tbody>
</table>

The higher the writeBuffer the faster the sorting, the write is more sequential. But the used memory is also higher.

**RunSize**

512MB data, fanIn 64, readBuffer 32, writeBuffer 8MB, runSize X

<table>
<thead>
<tr>
<th>runSize[MB]</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>2m23</td>
</tr>
<tr>
<td>16</td>
<td>1m55</td>
</tr>
<tr>
<td>32</td>
<td>1m14</td>
</tr>
</tbody>
</table>

The higher the runSize, the more is sorted in main memory, the lower the overall sorting process time. On the hard drive this is a much bigger performance difference. The question remains: is it faster taking a lower runSize and having a better fanIn instead?

### 3.5 SSD vs HD Strategy

In this section we describe the different strategies for both drives that apply to the external sorting algorithm.

#### 3.5.1 Parameter Influence

What is the influence of the different parameters on the overall external sorting process? They must not necessarily have the same impact on each different drive.
fanIn

For the fanIn there is the biggest difference:

**SSD:** On a solid state drive use a high fanIn and merge all the runs just in one step.

**HD:** On a hard drive a too small, but also a too high fanIn, is bad. The test results showed that a reasonable fanIn is 8-32 (but try to use a low fanIn, 32 is very high for the hard drive). When choosing a fanIn higher than 16, there might be a better solution with a lower fanIn and one more merge level. Also consider the fanIn in the end level (when all data is processed in the final merge). A very small fanIn at the end is bad. Then take a a higher fanIn instead and eliminate a whole merge level.

readBuffer

The readBuffer should be higher on the hard drive:

**SSD:** The solid state drive does not need a huge readBuffer. Random access is fast on a solid state drive. A readBuffer of 32-128KB seems reasonable.

**HD:** The hard drive should have a higher readBuffer, 128-512KB, depending on the memory that can be spared.

The readBuffer is used for each run in the merge step. A larger readBuffer causes more memory usage.

writeBuffer

For the online solution the writeBuffer is not used (in the wrapperQueue) when doing no multilevel merge. In general we can say: a large writeBuffer decreases the overall sorting time, more data will be written sequential. This is good for both drives. But the difference is not that much. One interesting observation is that the writeBuffer may written sequential anyway. For example choose the writeBuffer and the readBuffer of the same size and a fanIn of 16. Then in the optimal case the writeBuffer writes in the merge step 16 times before the readBuffers are empty and have to be filled again. A small writeBuffer could eventually have some influence on the solid state drive. The erase problem could have some effects on the performance depending on the drive’s behavior.

Strategy Difference

In our experiments we observed that the solid state drive uses another configuration than the hard drive to achieve the best performance. The solid state drive does not suffer from a high fanIn, but instead writing is not very fast (see previous performance measurements). This results in a new strategy (see Figure 3.7). For the solid state drive it is mostly faster to merge all the runs in just one step, skipping the multilevel merge step. The patterns that occur in this strategy are many random reads, when fetching the data from the runs, and large sequential writing. These patterns are optimal for solid state drives. The best strategy for the hard drive instead is the old strategy using multilevel merge steps. These steps are done to avoid too much randomness in the reading of the runs. Since the write speed of the hard drive is much faster this strategy pays off for the hard drive.
3.5.2 How to Choose Parameters

This subsection explains how to choose the parameters to get a good solution for each drive. It is not easy to find the best parameters that are available, but following these rules a good parameter set can be found easily.

SSD

For the solid state drive, try to skip the multilevel merge step.

1. make as huge runs as possible → \( n_{\text{runs}} \).
2. check if direct merging is possible: \( n_{\text{runs}} \times 128\text{KB} < \text{MEMORY} \).
3. check if direct merging is possible: \( n_{\text{runs}} \times 32\text{KB} < \text{MEMORY} \).
4. if yes, then use the parameters: \( \text{fanIn} = n_{\text{runs}} \). Use a readBuffer of 32KB (or 128KB if possible), give the rest of the memory to the writeBuffer.
5. if no, then no direct merging is possible. Try \( \text{fanIn} = \text{ceil}(\sqrt{n_{\text{runs}}}) \)

If the memory is low consider choosing a lower \( \text{fanIn} \) but having a larger readBuffer: (sorting 512MB data. with small \( \text{runSize}(128\text{KB}) \), and 16MB memory).

<table>
<thead>
<tr>
<th>disk</th>
<th>runSize[KB]</th>
<th>fanIn</th>
<th>readBuffer[KB]</th>
<th>writeBuffer[KB]</th>
<th>time</th>
<th>levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 SSD</td>
<td>128</td>
<td>4096</td>
<td>4</td>
<td>32</td>
<td>3m22</td>
<td>4096</td>
</tr>
<tr>
<td>2 SSD</td>
<td>128</td>
<td>64</td>
<td>128</td>
<td>8192</td>
<td>2m53</td>
<td>64 64</td>
</tr>
</tbody>
</table>

This experiment is somehow scaled down (with the small \( \text{runSize} \)). With 16MB memory there would be at most 496 runs with a 32KB buffer and a writeBuffer of 512KB. 496 runs could nearly store 8GB of data. With more data the readBuffer would be very small having only 16MB main memory. So here an additional merge phase can be added to let the readBuffer at a good size. This leads to a better performance than merging all the runs at once with a very small readBuffer.
3.5 SSD vs HD Strategy

The last column 'levels' in the tables denotes the merge levels that correspond to this experiment. For example the levels '64 64' mean that in a multilevel merge step with the fanIn of 64 the runs are merged. Then the final merge can be done with a fanIn of 64. '21p' means that directly a final merge is done with 21 runs, but the last one is only a partial run (when dividing the data into runs a partial run might be left).

**HD**

The parameters of the hard drive can be chosen:

1. make as huge runs as possible → nruns.
2. check if direct merging is possible: nruns <= 20?
3. if yes, then use the parameters: fanIn = nruns. Use a readBuffer of 32KB (or 128KB if possible), give the rest of the memory to the writeBuffer.
4. if no, then find fanIn with at least 1 merge level. fanIn between (8,32) and does not have a very low final merge level (not less than 4)

Where should the sorting process now take place, having our two drives? Always sort on the solid state drive. The experiments showed that in most cases the sorting is faster on the solid state drive. The difference is not big. For small data the hard drive can still take some advantage of its internal buffer. Sorting more data favors the solid state drive.

### 3.5.3 Test Cases

Some arbitrarily chosen test cases show how the parameters could be selected. The difference is not that big, but choosing the correct parameters the solid state drive is faster. Why is there not a bigger difference? The solid state drive has a much lower access latency. When merging runs it can gain advantage from this property. But the write speed of the solid state drive is only about a third compared to the hard drive. There the hard drive will gain time. So in the end the difference is not that big. But the more data to sort, the more the hard drive will suffer from random accesses in the merge step. Sorting more data favors the solid state drive.

In the following test cases the parameters are selected with the rules we presented in the last section. Certain parameters are slightly varied to show how this influences the sorting process compared to the good parameter set.
Test case 1: sort 321 MB data, 16MB memory

<table>
<thead>
<tr>
<th>disk</th>
<th>runSize[MB]</th>
<th>fanIn</th>
<th>readBuffer[KB]</th>
<th>writeBuffer[KB]</th>
<th>time</th>
<th>levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSD</td>
<td>16</td>
<td>21</td>
<td>32</td>
<td>15712</td>
<td>47s</td>
<td>21p</td>
</tr>
<tr>
<td>SSD</td>
<td>16</td>
<td>21</td>
<td>128</td>
<td>13696</td>
<td>48s</td>
<td>21p</td>
</tr>
<tr>
<td>SSD</td>
<td>16</td>
<td>21</td>
<td>512</td>
<td>5632</td>
<td>49s</td>
<td>21p</td>
</tr>
<tr>
<td>HD</td>
<td>16</td>
<td>21</td>
<td>128</td>
<td>13696</td>
<td>47s</td>
<td>21p</td>
</tr>
<tr>
<td>HD</td>
<td>16</td>
<td>8</td>
<td>512</td>
<td>12288</td>
<td>53s</td>
<td>8p 3</td>
</tr>
<tr>
<td>HD</td>
<td>16</td>
<td>5</td>
<td>512</td>
<td>13824</td>
<td>55s</td>
<td>5p 5</td>
</tr>
</tbody>
</table>

→ same time for SSD and HD

Test case 2: sort 917 MB data, 16MB memory

<table>
<thead>
<tr>
<th>disk</th>
<th>runSize[MB]</th>
<th>fanIn</th>
<th>readBuffer[KB]</th>
<th>writeBuffer[KB]</th>
<th>time</th>
<th>levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSD</td>
<td>16384</td>
<td>58</td>
<td>32</td>
<td>14528</td>
<td>2m43</td>
<td>58p</td>
</tr>
<tr>
<td>SSD</td>
<td>16384</td>
<td>58</td>
<td>128</td>
<td>8900</td>
<td>2m39</td>
<td>58p</td>
</tr>
<tr>
<td>SSD</td>
<td>16384</td>
<td>58</td>
<td>256</td>
<td>1536</td>
<td>2m49</td>
<td>58p</td>
</tr>
<tr>
<td>HD</td>
<td>16384</td>
<td>8</td>
<td>512</td>
<td>12288</td>
<td>3m16</td>
<td>8p 8</td>
</tr>
<tr>
<td>HD</td>
<td>16384</td>
<td>16</td>
<td>512</td>
<td>8192</td>
<td>2m58</td>
<td>16p 4</td>
</tr>
<tr>
<td>HD</td>
<td>4048</td>
<td>16</td>
<td>512</td>
<td>8192</td>
<td>3m35s</td>
<td>16p 16</td>
</tr>
</tbody>
</table>

→ 1,2: SSD is slightly better with a readBuffer of 128KB instead of 32 KB
→ 6: use not all memory for the runSize, but try another fanIn

Test case 3: sort 211 MB data, 128MB memory

<table>
<thead>
<tr>
<th>disk</th>
<th>runSize[MB]</th>
<th>fanIn</th>
<th>readBuffer[KB]</th>
<th>writeBuffer[KB]</th>
<th>time</th>
<th>levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSD</td>
<td>128</td>
<td>2</td>
<td>32768</td>
<td>65536</td>
<td>23s</td>
<td>2p</td>
</tr>
<tr>
<td>SSD</td>
<td>16</td>
<td>16</td>
<td>6114</td>
<td>32768</td>
<td>28s</td>
<td>16p</td>
</tr>
<tr>
<td>SSD</td>
<td>128</td>
<td>2</td>
<td>32768</td>
<td>131008</td>
<td>26s</td>
<td>16p</td>
</tr>
<tr>
<td>HD</td>
<td>128</td>
<td>2</td>
<td>32768</td>
<td>65536</td>
<td>24s</td>
<td>2p</td>
</tr>
<tr>
<td>HD</td>
<td>16</td>
<td>16</td>
<td>6114</td>
<td>32768</td>
<td>26s</td>
<td>16p</td>
</tr>
</tbody>
</table>

→ use all memory for the runSize
→ use more memory for the readBuffer if there is much memory

3.6 Optimizations

There was a lot of research to optimize the external sorting algorithm. Our solution implements the basic easy understandable algorithm. Our goal was not to implement a highly optimized algorithm. We wanted to find differences between the solid state drive and the hard drive.

To optimize the external sorting one could add additional buffers to do more reading at one time. Parallelization can improve the overall sorting process. While one thread is merging data another could already fetch data or write data. Many tasks could be done in parallel. For more information see for example [16, 11, 32].
3.7 Multiple Disks

Multiple drives could be used to to the sorting (see Figure 3.8). Assume the full data lies on the drive 1, then the runs could be read from this drive, sorted in main memory and written to the drive 2. Then all runs could be merged from drive 2 in main memory to the drive 1. At the end the data lies on the same position as before but sorted.

![Diagram of external sorting with multiple disks](image)

Figure 3.8: External sorting with multiple disks.

Applying this scheme to use a hard drive and a solid state drive:

1. Data is stored on the hard drive.
2. Read runs from hard drive, write sorted runs to the SSD.
3. Merge all runs from SSD, write final run to the hard drive.

With this combination sometimes the solid state drive is beaten. But the time difference is not that big.

This strategy might have the following advantages:

- Read first in huge chunks from the hard drive.
- When merging the runs take advantage of the small access latency of the solid state drive.
- Write the final merged data to the hard drive.
- At the end the data lies on the same position, but sorted.

3.7.1 Test Cases

Applying the test cases to this new multi disk sorting:
Test case 1: sort 321 MB data, 16MB memory

<table>
<thead>
<tr>
<th>disk</th>
<th>runSize[MB]</th>
<th>fanIn</th>
<th>readBuffer[KB]</th>
<th>writeBuffer[KB]</th>
<th>time</th>
<th>levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>MultiDisk</td>
<td>16</td>
<td>21</td>
<td>32</td>
<td>15712</td>
<td>50s</td>
<td>21p</td>
</tr>
<tr>
<td>MultiDisk</td>
<td>16</td>
<td>21</td>
<td>128</td>
<td>13696</td>
<td>49s</td>
<td>21p</td>
</tr>
<tr>
<td>MultiDisk</td>
<td>16</td>
<td>21</td>
<td>512</td>
<td>5632</td>
<td>48s</td>
<td>21p</td>
</tr>
</tbody>
</table>

Test case 2: sort 917 MB data, 16MB memory

<table>
<thead>
<tr>
<th>disk</th>
<th>runSize[KB]</th>
<th>fanIn</th>
<th>readBuffer[KB]</th>
<th>writeBuffer[KB]</th>
<th>time</th>
<th>levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>MultiDisk</td>
<td>16384</td>
<td>58</td>
<td>128</td>
<td>8900</td>
<td>2m38</td>
<td>58p</td>
</tr>
<tr>
<td>SSD</td>
<td>16384</td>
<td>58</td>
<td>32</td>
<td>14528</td>
<td>2m43</td>
<td>58p</td>
</tr>
</tbody>
</table>

Test case 3: sort 211 MB data, 128MB memory

<table>
<thead>
<tr>
<th>disk</th>
<th>runSize[MB]</th>
<th>fanIn</th>
<th>readBuffer[KB]</th>
<th>writeBuffer[KB]</th>
<th>time</th>
<th>levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>MultiDisk</td>
<td>128</td>
<td>2</td>
<td>32768</td>
<td>65536</td>
<td>23s</td>
<td>2p</td>
</tr>
</tbody>
</table>

In these test the multiDisk configuration is sometimes even faster than the solid state drive.

### 3.8 Conclusions

Chapter 2 showed that solid state drives have other performance characteristics than conventional hard drives. In this chapter we looked at the important external sorting algorithm. Many algorithms have been tuned to better support the characteristics of a conventional hard drive. In external sorting multilevel merges avoid too much randomness in the access patterns. But a solid state drive does not suffer from random accesses. In fact random accesses are very cheap for a solid state drive. In this section we discovered that a solid state drive needs another strategy to achieve its best performance. Always do direct merge of the runs and skip the multilevel merge. This results in a faster sorting result for the solid state drive. It can gain advantage of the very small access latency and it does not have to write that much.

The time difference solid state drive versus hard drive is not always that much. Especially for small data they are both comparable. For larger data our solid state drive is slightly faster than our hard drive.

We also proposed another sorting method using multiple disks.

We have seen that using a solid state drive it can matter changing the implementations of traditional algorithms. Using different strategies can speed up your algorithm on a solid state drive. Newer high performance solid state drives promise faster performance and even a smaller access latency. Therefore running these experiments with a newer solid state drive would show larger differences.
4 Index Structures

Computer systems often maintain so much data that not everything can be stored in main memory. Main memory is limited. Conventional hard drives have been used for years as the main storage device. With a large data collection, indexes are needed to quickly access this data on a drive. Many indexes have been tuned to get the highest possible performance. In this chapter we build simple index structures. We build an index structure using CSS-trees which are partially persisted on the drive. Another index structure we try is a commonly used B+Tree that stores the data on the drive. We run a defined workload on these index structures. The overall throughput of the index structures is measured on both drives. Here we will see differences on the solid state drive versus the conventional hard drive. Since many years index structures were tuned for the usage on traditional hard drives. With these experiments we hope to see how the solid state drive can advance in common index structures. We try to find out what favors a solid state drive and how an index structure can be chosen to support its new characteristics to achieve a better performance. Our goal is not to make high performance and tuned indexes. We aim to see the differences between solid state drives and the conventional hard drives on different indexes.

4.1 Data Structures

In this section we shortly describe the important data structures that are used as parts of our index structures.

4.1.1 B+ Tree

A B+Tree is a commonly known tree data structure. This is not the binary tree. It is often used in databases and file systems. Internal nodes store values and pointers to the leaf nodes which store in fact the data. All leaves are at the same level. The B+Tree is defined by two parameters:

\[ k: \] the size of an internal node.

\[ k^*: \] the size of a leaf node.

In a B+Tree (see Figure 4.1) all the data (in the form of tuples) is stored in leaf nodes. By increasing the size of the internal nodes one can decrease the height of the tree. Optimized for conventional hard drives the data lies sequentially on the hard drive. For more information on B+Trees see any database architecture book.
4.1.2 CSS-trees

Cache-Sensitive Search Trees (CSS-trees) have been proposed in 'Cache Conscious Indexing for Decision-Support in Main Memory' [41] (also see [40]). The CSS-tree stores an index over a sorted array. In an additional array the index is stored as a balanced search tree. The size of the nodes in a CSS-tree is designed to match the cache-line size of the machine. A CSS-tree is an immutable index. Updating an entry in a CSS-tree results into the creation of a new CSS-tree. When searching an entry in a CSS-tree, the internal nodes are traversed with the binary search algorithm. The result is the index where the corresponding leaf starts. With this index the data of the leaf can be found and processed using again the binary search algorithm.

In our first index structure 4.3 we used CSS-trees to build a CSS-tree hierarchy. The data in the leaves is persisted on a drive. Each CSS-tree stores its level and a list contains all the actual CSS-trees in the hierarchy. A parameter defines the k-way merge strategy. Whenever there are k CSS-trees of the same level they are merged and a new CSS-tree of the higher level is built.
4.2 Implementation Overview

Our testing machine has a CPU with two cores. So we decided to use two different threads for:

- Event generation (workload thread).
- Index structure management (index thread).

The event generation thread, described in the next section, generates all the events that arrive in our index structure.

The index structure manager processes the incoming events and manages the index structures. The underlying index structure is maintained and updated when needed. For example a merge is performed when there are enough updates.

![Diagram](image)

Figure 4.3: Thread usage in our test application.

4.2.1 Event Generation

The event generation is implemented in a separate thread. The generator contains a queue where the events are stored for further processing.

The implementation is specific and uses the primitive type int for the records. If a tuple (key-value pair) has the size 2, then it consists of a key (int) and value (int). If the tupleSize is more than 2, then the value is an int[]. A key-value pair is also called record.

Events

Assume to have a tupleSize is of 2 (one key maps to one value). In our test scenario we defined the following events that can occur:
**READ:** search a given key in the index structure. The result is a value.

**READ_FAIL:** search a given key in the index structure. The key is not available in the index structure therefore the result is an integer constant (KEY_NOT_FOUND) which indicates that the key was not found.

**INSERT:** insert a new tuple into the index structure. Key and value are given.

**UPDATE:** update an existing tuple in the index structure. The key is given and a new value is chosen.

**DELETE:** remove a tuple completely from the index structure. This event is not supported yet.

To overcome the Java object creation overhead these events are stored in one large event block (e.g. containing 1024 events). Such a block contains 3 arrays:

- **eventType[] types:** contains the type of the events.
- **int[] keys:** contains the keys.
- **int[][] data:** contains the values that correspond to the keys.

With the appropriate index we can easily get the current event out of such a block. The values are of type int[][][]. When the tupleSize is 2 then each array has the length 1 and contains only one value.

**Strategy**

For our event generator we can implement different strategies for the creation of new events. We built a strategy that randomly creates new events. Parameters can be set to divide the event generation in percentages of the available events **READ, READ_FAIL, INSERT, UPDATE** and **DELETE**. Another parameter specifies how many events are first filled into the index structure (using the event **INSERT**) until the real workload starts (e.g. fill 100MB data first).

In Table 4.2.1 reasonable workloads are listed:

<table>
<thead>
<tr>
<th>event</th>
<th>read workload [%]</th>
<th>write workload [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ</td>
<td>70</td>
<td>10</td>
</tr>
<tr>
<td>READ_FAIL</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>INSERT</td>
<td>10</td>
<td>40</td>
</tr>
<tr>
<td>UPDATE</td>
<td>10</td>
<td>40</td>
</tr>
<tr>
<td>DELETE</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.1: Two example workloads for the experiments.
4.2 Implementation Overview

Implementation

Figure 4.4 shows how the classes of the implementations are linked together. The queue manager is a separate thread. Its purpose is to ask for new events and to manage the queue. The queue is a synchronized list. It contains events that have not been processed yet. The queueSize had to be bounded to a specific limit because the event generation is much faster than the processing of events and the maintaining of the index structure.

![Diagram](image)

Figure 4.4: An overview of the event generation implementation.

The queue manager has an instance of the event generator. This generator creates new events (which come in whole blocks). This generation is implemented by a specific strategy. A method can be called to get a new event block. The strategy decides how to generate new events. This could be done completely random. In the strategy we use for the event generation we have defined a certain workload. At the beginning the index is just filled with randomly chosen tuples with the INSERT event. After a specified number is reached the real workload generation starts. Of course our strategy needs data structures to generate the events. We need for example data structures to hold all the existing keys. So the strategy generates a new key and can be sure it is not existing. These data structures are used only for the event generation and do not have any influence to the index structure. The event generation consumes a lot of memory, but somehow this has to be done. Another possibility would be to read the whole workload from a file. This could easily be implemented by another strategy.

The generator contains a pool to recycle the event block objects. Two different threads are operating on these structures. So this access to the pool and the methods that change the queue with the new events have to be synchronized.

The queue manager writes a log file to store some information about the events that are generated during the runtime of the application.
The queue manager does the following work sketched in Listing 4.1.

```java
// initialize parameters and log files
init();

// after thread is started
while (true) {
    if (!queue.isFull()) {
        EventBlock e = generator.getNewBlock();  // get a new block
        queue.add(e);                             // add the new block
        log(e);                                   // write the logs
    }
}
```

Listing 4.1: Pseudo-code of the event generation.

For a short class description see the Appendix Source Code Overview C.3.1.

### 4.3 Index 1: CSS-tree hierarchy

The CSS-tree hierarchy (see Figure 4.5) is the first index structure to be built. It uses CSS-trees and a B+ Tree. The CSS-trees store the data on a physical drive. The overall index structure contains a whole hierarchy of CSS-trees. The internal nodes are stored in main memory. The data in the leaves of such a CSS-tree is persisted on the drive. They contain a diskPosition that indicates where the data is stored on the drive. With the internal nodes and the diskPosition the needed data of a leaf can be fetched from the drive. In a first step the internal nodes are searched to get the index of the leaf. Then the diskPosition of the leaf can be calculated and fetched into main memory. Again this data can be searched to find the desired key.

The B+ Tree is a complete in-memory data structure. It represents a delta index. From time to time this delta has to be merged with the actual CSS-trees hierarchy. A parameter defines the k-way merge strategy. If k is 2, then CSS-trees are merged whenever there are two of the same level in the list. The result is one CSS-tree of the next higher level.

In this index structure a read operation can be one of the events READ or READ_FAIL. Listing 4.2 shows the pseudo-code of such an event being processed. In the index structure then first the B+ Tree is queried. If there is a result then the value was found in the B+ Tree. It can be returned to answer the event. Else the CSS-trees have to be queried. This means traversing the CSS-trees in a reverse order, from the end to the start. At the end of the list lies the newest CSS-tree. This tree also contains the newest data. A CSS-tree is an immutable index structure. Multiple occurring values are filtered in the merge step. The situation can occur that a CSS-tree of level 0 contains the key 42 which is also in a CSS-tree of level 2. This does not hurt. Because the list is traversed backwards always the newest value is queried. If all CSS-trees have been traversed and the key could not be found, the constant KEY_NOT_FOUND is returned (this corresponds to the event READ_FAIL).
Figure 4.5: The CSS-tree hierarchy index structure with a B+Tree delta.

Listing 4.2: Pseudo-code of querying a key.
A write operation can be one of the events INSERT or UPDATE. Listing 4.3 shows the pseudo-code of these events where the key-value pair is simply added to the B+Tree delta. If a key is already in this B+Tree, the value of this key can simply be overwritten (UPDATE) and the index structure is still correct. Then the size of the B+Tree has to be checked. A parameter defines the maximum size of the delta. If the B+Tree is full then this delta has to be persisted and merged with the CSS-tree hierarchy.

```java
// INSERT, UPDATE
btree.add(key, value);  // add the value
if (btree.isFull()) {
    // the btree is full
    treeManager.merge(btree);  // create CSS-tree. merge if necessary
    btree.clear();
}
```

Listing 4.3: Pseudo-code of updating or inserting a key.

The merge of a B+Tree with the CSS-tree hierarchy is the most complicated part of this whole index structure. This merge process can be divided into 3 phases:

1. Initialization phase: make a new CSSDiskTree of the B+Tree data.
2. Collect phase: collect the CSS-trees to merge.
3. Merge phase: merge if necessary.

Listing 4.4 describes these 3 phases in pseudo-code.

In the initialization phase all the data of the B+Tree is collected and a new CSSDiskTree is created. This new CSS-tree is written to the physical drive to a new diskPosition. It is a CSS-tree of the lowest level. Then it is added to CSS-tree hierarchy. A list contains all the currently available CSS-trees.

In the second phase the algorithm collects all the CSS-trees that have to be merged. A parameter here defines the k-way merge. We start with the smallest level and collect trees with (k-way - 1) occurrences of the same level. This means all the trees are collected which can be merged directly into a higher CSS-tree. The collecting is stopped when there are not enough trees for the actual level. For example the trees levels: \{0, 0, 1, 2\} can be merged directly into a new tree of level 3 (doing a 2-way merge). The sub-steps of this merging process would be \{1, 1\} → 2 and then \{2, 2\} → 3. This merge steps can be done in only 1 merge. When collecting the data a special check for the level 0 trees has to be done. To start the merging process there have to be (k-way) occurrences of level 0 trees.

The third phase is the actual merge step. If there are collected trees then these are merged here into a CSSDiskTree of the next higher level. After the merge the data is stored on the drive at a new diskPosition and the new tree is added to the collection. For the merge step
some helper classes are needed (see implementation for a detailed description). Each CSS-DiskTree has a unique age. The older this age, the newer the tree. This makes a CSS-tree comparable. For example it can occur that a tree of level 0 and a tree of level 2 contain the same key with a different value. In the merged tree of level 3 (merge example above) then only the key with the value of the level 0 tree must occur. This value is newer. The other entry has to be deleted. CSS-trees cannot be updated. Newer values are just written in new trees. In the merge steps then these multiple occurrences are filtered and only the newest tuples are written to the merged tree. This guarantees the correctness of the whole index structure.

```java
// 1. make a new CSS-tree of the btree data
long position = getNewDiskPosition(); // get a new position
int[] treeData = btree.getContent(); // get the btree data
CSSDiskTree newTree = new CSSDiskTree(treeData, position);
trees.add(newTree); // add new tree

// 2. collect all the CSS-trees to merge
int actualLevel = 0;
boolean collecting = true; // collect the trees of the same level
while (collecting) {
    selectedTrees = trees.collectByLevel(actualLevel);
    if (selectedTrees.size == (kway-1)) {
        // merge these trees
        treesToMerge.add(selectedTrees);
        trees.remove(selectedTrees);
    } else { // no merge at this level. stop
        collecting = false;
    }
    actualLevel++;
}

// 3. merge if necessary
if (treesToMerge.size() != 0) { // merge these trees
    position = getNewDiskPosition();
    CSSDiskTree newTree = treesToMerge.merge(getNewDiskPosition(), actualLevel);
    trees.add(newTree);
} else { // no trees to merge
}
```

Listing 4.4: Pseudo-code of the simplified merge algorithm.

Figure 4.6 sketches an example evolution of this index structure. At the beginning (‘t=0’) the B+ Tree is empty and no CSS-trees are stored on the drive. The index structure is empty and does not contain any data yet. As new events are processed by the manager, data is added into the B+ Tree. There is a time ‘t=1’ when the B+ Tree is full. Its maximum size is limited by a given parameter. The higher this parameter the higher the amount of main memory that is used for the delta index. So the B+ Tree has to be merged with the CSS-tree hierarchy. Now there is no CSSDiskTree on the drive. We can just persist the data of the
B+Tree to the drive and empty the old B+Tree. The actual configuration is shown at time ‘t=2’. When the next B+Tree is full (‘t=3’) the next merge step has to be executed. Again a new CSSDiskTree is created and added to the list (‘t=4’). But now there are two trees of the same level in the CSS-tree hierarchy. In a 2-way merge now these trees of level L0 have to be merged. This results in a new CSSDiskTree of the level L1 that persists its data on the drive (‘t=5’). The old B+Tree is cleared and the index structure is ready to process further events. After some time (‘t=6’), the B+Tree is full again and has to be merged with the CSS-tree hierarchy. The next step would be to create a new CSSDiskTree of the level L0. No merge step need to be done then.

Figure 4.6: An example evolution of the CSS-tree hierarchy index structure with a B+Tree delta (2-way merge).

4.3.1 Implementation

Figure 4.7 contains an overview of the classes that are used for the implementation of this index structure.

The index structure manager extends the Java class Thread. It is started as a separate thread. This class has an instance of an event generator where new events can be fetched. It has an instance of a BTree index structure and the manager of the CSS-trees. The BTree is acting as a delta index. A parameter defines the number of tuples it can hold.

The CSS-trees manager contains all the CSSDiskTree classes that store their data on a drive. Whenever the index structure manager processes a new event, this event is evaluated according to its definition. The necessary information is gathered from the BTree or from the
CSS-trees manager. In case of a write the data is simply added to the BTree. If the BTree is full then it is merged with the CSS-trees. This is done by calling the CSS-trees manager method merge(bTree). Then the CSS-trees manager does all the necessary described steps. It stores the data as a new CSSDiskTree and if necessary merges them.

The CSSDiskTree is an implementation of a CSS-tree, but it was adapted to store the leaf data on a drive. It contains a position and an instance of the class BlockIOService.java. With this information it can do all the I/O requests. Every CSS-tree has the same instance of the BlockIOService. The I/O is done directly on the Linux raw interface with no caching in between. The CSSDiskTree contains internal nodes. These are needed to find the leaf that fits to a key. Then the data can be loaded and traversed with binary search to find the value of the key. If the key is not found then KEY NOT FOUND is returned (if the tupleSize is more than 2 then NOT FOUND ARRAY is returned which is an array containing only the value KEY NOT FOUND). This and many other constants can be defined in the class IndexStructureConstants.java.

In the merge phase some helper classes are needed to do the merge. The class CSSDiskTreeWrapper wraps a CSSDiskTree. It takes some values of a CSSDiskTree and makes itself comparable. This is needed to eliminate duplicate keys that occur in multiple trees. Each CSSDiskTree has a unique age. The wrapper itself contains an internal buffer. It can fetch an array of elements and then give them back element-wise. When the buffer is consumed it can just get the next chunk until all the elements have been processed.

The class BlockIOService.java that does all the I/O operations must operate in blocks of 512 Byte size. The I/O requests have to be aligned since we use no kernel buffering and access the drive directly (see B.2).

The class CSSDiskTreeMerger then takes full control of the merging. It has all the wrappers stored in a PriorityQueue and since these wrappers implement the comparable interface (the smaller the age of the of the underlying CSS-tree the older the data) one can easily get the smallest element. The merger needs to store the previous key and so it can eliminate the keys occurring in multiple CSS-trees.

For a short class description see the Appendix Source Code Overview C.3.2.
4.3.2 Characteristics

This index structure contains a delta index, that is realized using a B+Tree.

Every write operation is stored in this delta index. When the B+Tree is full then the whole data can be written as a large sequential run. The overlying index structure of this data is realized in a CSS-tree. When a merge is done then all the runs from the CSS-trees are merged and the data can again be written in relatively large sequential chunks choosing the buffer size. Multiple occurrences of keys are filtered in the merge step. So only large sequential data is written when merging or adding a new CSS-tree. Inserting or updating a key into the B+Tree is considered as 0 costs. No I/O operation has to be done.

In every read operation the CSS-tree hierarchy is traversed. This is where the costs lie. Every read operation costs the access time and the transfer time of the data to the main memory. For every traversed tree these costs are added. If there are many trees in the hierarchy this will cost some time especially on a hard drive.

The overall throughput is mostly affected by the read accesses to the tree hierarchy. For the solid state drive a disk access is not that expensive, but hard drive has to pay much more for a random read. The more trees in the hierarchy, the higher the costs and the lower the overall throughput. The write operations are executed in large sequential chunks. Both drives can profit from collecting the data in a delta. The hard drive is fast in sequential writing, but this is also the best we can have for the solid state drive. Using this delta index we can avoid small random writes.

Note that the values stored in the B+Tree as a delta are not persisted on a drive. They remain in main memory until a merge occurs. If the system would crash in between the data would be lost.

4.3.3 Experiment

We let our index structure run on both drives with the same parameters. What is measured is the throughput of the index structures (how many events can be processed each second).

Setup

Parameters for the experiment:

- LeafSize: 32KB
- Maximum BTree size: 1MB
- Fill 100MB data (INSERT).
- Then workload: 70% READ, 10% READ_FAIL, 10% INSERT, 10% UPDATE
- 2-way merge
- Stop: 153th bTree.
4.3 Index 1: CSS-tree hierarchy

4.3.4 Description

Each leaf node on a CSS-tree contains 32KB of data. We first fill in 100 times a full B⁺Tree of the size 1MB into the index to have some data, then we let run the read oriented workload of 70% READ, 10% READ_FAIL, 10% INSERT, 10% UPDATE. The application is configured to do a 2-way merge. The throughput is measured. We want to know how many events the index structure can process every second. The index structure is stopped after the 153th B⁺Tree was added.

Model

The different events that occur during the runtime of the index structure have always the same cost depending on the number of trees that are actually in the CSS-tree hierarchy. The throughput will depend on these events. The workload is clearly defined. Some calculations can estimate this throughput depending on the performance measurements of each drive:

When 100 B⁺Trees are merged, then our CSS-tree structure has the following configuration: CSSTree(64) + CSSTree(32) + CSSTree(4). There are 3 trees in total. To answer a read query the smallest CSS-tree has to be traversed first (see Figure 4.8), it could contain the newest values. The important point is, that when a read lookup is done then mostly more than half of the data is located in the highest tree. All the lower level trees together could contain the same amount of data as the highest CSS-tree. Therefore the application has to look at many trees until the key is found.

![CSS Trees](image)

Figure 4.8: The read lookup order of the CSS-tree hierarchy. Check the youngest tree first. Configuration after 100 B⁺Tree deltas were merged.

So assume: if there is the index structure with 3 CSS-trees, then approximately 2.5 CSS-trees have to be traversed. The approximate overall levels to lookup is one half smaller than the
highest CSS-tree level (considering a 2-way merge. If there is only 1 tree, then the costs are 1).

The events produce the following actions in the index manager:

- **READ, READ_FAIL**: check the B+Tree. Check the CSS-tree hierarchy.
- **UPDATE, INSERT**: add to the B+Tree. Do a merge if necessary.

Parameters:
- \( T_{seek} \): access latency
- \( R_{seq} \): sequential read speed
- \( W_{seq} \): sequential write speed
- \( D \): dataSize
- \( T_{transfer} \): transfer time = \( \frac{D}{R_{seq}} \) or \( \frac{D}{W_{seq}} \)
- \( numTrees \): the number of trees in the CSS-tree hierarchy.

There is the following model for the costs of the different events:
- **READ**: \( (T_{seek} + \frac{D}{R_{seq}}) \times (numTrees - 0.5) \)
- **READ_FAIL**: \( (T_{seek} + \frac{D}{R_{seq}}) \times numTrees \)
- **INSERT**: 0 or merge()
- **UPDATE**: 0 or merge()

For the merge() operation there are very different costs depending on the current CSS-tree hierarchy configuration:

- **minimum**: only write the CSS-tree to disk. no merge: \( T_{seek} + \frac{D}{R_{seq}} \)
- **maximum**: merge all trees: read and write all data. Do multiple seeks.

The costs of a merge() is not that easy to calculate. It contains reading all the data from the CSS-trees to merge and writing the data to a new CSS-tree. Depending on the actual configuration this can be only adding the new CSS-tree to the drive. But it could also be merging all the available data into a new CSS-tree. Then the costs can be much higher. Still the costs do not hurt. The writing can be done in larger sequential blocks which is the best one can get for both the drives.

Using this cost model one can now predict the overall throughput of both drives using the performance measurements:
4.3 Index 1: CSS-tree hierarchy

**SSD**

We have the following parameters (from the previous performance measurements):

- Access latency: 0.3ms
- Read speed: 120MB/s

For each access we need to transfer the leaf data (32KB) to the main memory. Calculating this transfer costs with the full read speed is $\frac{32KB}{120MB/s} = 0.26\text{ms}$. So in total: $0.3\text{ms} + 0.26\text{ms} = 0.56\text{ms}$ for each read lookup. When the lookup is 2.5 CSSTrees on average, one calculates $2.5 \times 0.56\text{ms} = 1.4\text{ms}$ for each READ event (with the configuration having 3 CSS-trees in the hierarchy). This means a maximum throughput of 714 READ operations per second can be achieved on the solid state drive.

Note: additionally some time is needed to do binary search over the internal nodes and all the 2.5 leaves from the drive.

**HD**

We have the following parameters (from the previous performance measurements):

- Access latency: 9ms
- Read speed: 115MB/s

For each access we need to transfer the leaf data (32KB) to the main memory. Calculating this transfer with full read speed the costs are $\frac{32KB}{115MB/s} = 0.27\text{ms}$. So in total: $9\text{ms} + 0.27\text{ms} = 9.27\text{ms}$. When the lookup is 2.5 CSSTrees on average, one calculates $2.5 \times 9.27\text{ms} = 23.18\text{ms}$ for each READ request. This means a maximum throughput of 43 READ operations per second can be achieved on the hard drive, when we take the average seek time.

Repeating this calculation for different tree configurations one obtains the maximum throughput values in Table 4.2.

<table>
<thead>
<tr>
<th>numTrees</th>
<th>throughput SSD</th>
<th>throughput HD</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1786</td>
<td>108</td>
</tr>
<tr>
<td>2</td>
<td>1191</td>
<td>72</td>
</tr>
<tr>
<td>3</td>
<td>714</td>
<td>43</td>
</tr>
<tr>
<td>4</td>
<td>510</td>
<td>31</td>
</tr>
<tr>
<td>5</td>
<td>397</td>
<td>24</td>
</tr>
<tr>
<td>6</td>
<td>325</td>
<td>20</td>
</tr>
<tr>
<td>7</td>
<td>275</td>
<td>17</td>
</tr>
</tbody>
</table>

Table 4.2: The estimated maximum throughput values for both drives.
Note that the hard drive will benefit from its internal buffer. Having CSS-trees with the size lower than the hard drive’s buffer (16MB) it can do some optimizations and load data of other CSS-trees already in the buffer. Then the next read access will not be a real read, but instead the data is fetched from the buffer. Additionally in the calculations the average full seek time of 9ms was taken. The hard drive will probably not need that much time to move the head. The data lies sequential on a very small region of the whole disk. The hard drive may do mini seeks which do not need that much time. So the whole picture will not look that bad on a hard drive in our experiments. But having a very huge index this would be the estimated throughput values.

**Results**

Figure 4.9 visualizes the throughput of the experiments. At the beginning of the tests 100MB of data is filled processing only INSERT events. This is very fast and has a high throughput on both drives because adding the events to the delta index is considered as 0 costs. Only the merge step needs some time. But since the data can be written sequentially a high overall throughput can be reached. When 100MB of data is filled into the index structure the real workload begins. Depending on the number of CSS-trees that are in the hierarchy another throughput is reached. When more CSS-trees are in the hierarchy for each read request approximately more trees have to be searched. This results in a lower throughput. The difference between the hard drive and the solid state drive is very high. The solid state drive is more than 4 times faster. Table 4.3.4 shows some measured throughput values. Especially the solid state drive follows the predicted pattern. The number of actual CSS-trees in the hierarchy determines the throughput. The throughput is always calculated after the merge of the $B^+$Tree, so this merge is taken into account. As the index structure grows the merge step involves more and more data. The costs of a merge grow and this slightly decreases the overall throughput. The hard drive does not have such a consistent throughput pattern. It can profit from its internal cache (16MB). This cache has too much influence to see a consistent pattern. Assuming the configuration CSS(64) + CSS(4) + CSS(1) is on the drive. When accessing the data of the first CSS-tree the CSS(4)-tree might also be loaded to the cache. Then the cost of accessing this tree are very low. From the throughput table one can see that the configuration CSS(128) + CSS(2) + CSS(1) reaches a higher throughput than CSS(128) + CSS(4) + CSS(1). We do not exactly know how this buffer is filled and which data is stored, but these results show that in the first configuration more data can be accessed through the cache, what results in a higher throughput.
4.3 Index 1: CSS-tree hierarchy

<table>
<thead>
<tr>
<th>bTree#</th>
<th>SSD</th>
<th>HD</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>99</td>
<td>24175</td>
<td>23919</td>
</tr>
<tr>
<td>100</td>
<td>20554</td>
<td>20470</td>
</tr>
<tr>
<td>101</td>
<td>714</td>
<td>107</td>
</tr>
<tr>
<td>102</td>
<td>521</td>
<td>98</td>
</tr>
<tr>
<td>103</td>
<td>523</td>
<td>92</td>
</tr>
<tr>
<td>104</td>
<td>411</td>
<td>84</td>
</tr>
<tr>
<td>105</td>
<td>727</td>
<td>98</td>
</tr>
<tr>
<td>106</td>
<td>528</td>
<td>91</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>127</td>
<td>360</td>
<td>50</td>
</tr>
<tr>
<td>128</td>
<td>302</td>
<td>46</td>
</tr>
<tr>
<td>129</td>
<td>1726</td>
<td>234</td>
</tr>
<tr>
<td>130</td>
<td>902</td>
<td>211</td>
</tr>
<tr>
<td>131</td>
<td>910</td>
<td>201</td>
</tr>
<tr>
<td>132</td>
<td>618</td>
<td>185</td>
</tr>
<tr>
<td>133</td>
<td>918</td>
<td>175</td>
</tr>
<tr>
<td>134</td>
<td>619</td>
<td>158</td>
</tr>
<tr>
<td>135</td>
<td>624</td>
<td>142</td>
</tr>
<tr>
<td>137</td>
<td>930</td>
<td>150</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.3: The throughput table of the experiments.
4.4 Index 2: B$^+$Tree

The second index structure we try is a simple B$^+$Tree that stores the data in the leaves on a drive. Figure 4.10 shows the sketch of this index structure. The B$^+$Tree implementation we use is from Stefan Hildebrand’s master thesis [22]. Only small changes had to be made. The leaf nodes now store their data on a drive. The data has to be loaded when the node asks for it. All the node objects are stored in-memory. Only the data of the leaves is stored on the drive. A leaf cache was added to the index structure. This cache can store a specified amount of leaves. It is between the leaves and the physical storage of their data. Whenever data is needed from a leaf, then this cache is considered first. If the leaf is already in-memory in the cache, there is a cache hit and the data must not be loaded from the drive. Updates will be done on the cache data directly. If the leaf is not cached, then it has to be loaded from the drive. To do this first another old leaf has to be deleted from the cache. It this old leaf is dirty it is written back to the drive. Actually the least recently used (LRU) leaf is chosen to make space in the cache. This leaf cache was added to make the two index structures more comparable. It is possible to calculate the memory usage of both index structures. The leaf cache uses the rest of the available memory that is not used for internal nodes.

4.4.1 Implementation

The index structure manager extends the Java class Thread and is linked to the generator of the events. As events are processed the corresponding methods of the DiskBTree are called to maintain the index structure.

The changes made in the DiskBTree classes are in the leaf level. The initial implementation contained a class LeafArrayMap that stored the keys and the data as an int[]. Now each
time the leaf data is accessed this data has to be loaded from the drive. To adapt the whole
B+ Tree implementation in the most easy way just the methods load(), unload() and flush()
amare added. Every time the data is needed the load() method has to be called that makes the
data available. The load() method then loads the data of the leaf into the cache (if necessary)
and links the int[] of the actual leaf. Then the data can be processed and updated. When the
data processing is finished the method unload() is called which sets the data pointers to null
and updates some parameters.

A LeafArrayMap uses a static class LeafCache to handle the I/O requests and update the
data. The LeafCache contains two int[][] to store the leaves and their values. Additionally an
array is used to hold the pointer to the leaf and to mark the leaf status dirty if the data was
updated for a leaf. Having the appropriate index one can access all the necessary data of a
leaf. A HashMap stores the mapping between a leaf and the used index. The diskPosition of
a leaf is mapped to an index (Integer object). Internally a free list contains all the indexes
of the free storage for new leaves. Another list stores the LRU order of the recently used
leaves. To read and write data from a drive the LeafCache uses the static I/O service called
StaticBlockIOService.java. This class handles all the bock-wise communication over the used
Linux raw interface. It contains a read and a write method that can be statically called.
Arrays can be passed down to this class to execute the I/O requests.

For a short class description see the Appendix Source Code Overview C.3.3.
4.4.2 Characteristics

This second index structure has other I/O characteristics. The write operations are always executed directly and in case of an UPDATE event an update-in-place is done. The leaf data is loaded, the value that corresponds to the key is searched and updated, and then the new data is written back to the same position on the drive. In case of an INSERT event the leaf node where the data fits is searched, then the key is inserted at the correct position and the updated data of the leaf is written back. If the leaf is then full it additionally has to be split. This means copying half of the data and write the new leaf to the drive, update some parameters to maintain the leaf nodes. So in every write operation at least 1 seek has to be done. Adding the LeafCache to the index structure changes the behavior. First always the cache is considered before doing an I/O operation on the drive. Using a cache one can insert sorted data very fast because the leaves are continuously filled and written back when they are full. A cache speeds up a workload where the same or near keys are used very frequently. Then this leaf is still in the cache. But having a total random workload a cache does not have much influence and cache hits do not occur often.

The read operations are also different. For each READ the corresponding leaf node is searched and the data has to be loaded and traversed. In the best case this needs only one seek. If the leaf is in the cache then no seek has to be done and the search can be considered having no costs.

The update-in-place data structure is generally bad for the solid state drive. It suffers under small random writes (see performance experiments 2.4). The hard drive does not have to face this problem. Once the head is moved to a certain position it only has the cost of the rotational delay and not the full access latency when writing back the data. Read lookup in this data structure is very cheap. Not many seeks have to be done to find the data. The internal nodes are kept in memory. Only the leaf nodes contain data that is persisted on the drive. The cache can even speed up the reading. Everything that is in the cache must not be loaded from a drive and has therefore nearly no costs.

4.4.3 Experiment

The index structure is tested on both drives with the same parameters and its throughput is measured (how many events can be processed each second).
Setup

The following parameters are used in the experiment:

- INSERT 100 MB data
- LeafCache with actual constant size of 2MB.
- LRU cache replacement policy.
- Scenario workload: 70% READ, 10% READ_FAIL, 10% UPDATE, 10% INSERT

Model

The different events that reach the index structure have always the same actions:

READ, READ_FAIL: Check the LeafCache. Check the BDiskTree leaf data.

UPDATE, INSERT: Check the LeafCache. Check the BDiskTree leaf data, if necessary. Update. Split if necessary.

parameters:

- $T_{\text{seek}}$: access latency
- $R_{\text{seq}}$: sequential read speed
- $W_{\text{seq}}$: sequential write speed
- $T_{\text{erase}}$: block erase time (1.5ms [2])
- $D$: dataSize
- $D_e$: size of an erase unit (512KB)
- $T_{\text{transfer}}$: transfer time $= \frac{D}{R_{\text{seq}}}$ or $\frac{D}{W_{\text{seq}}}$

For the hard drive the model for in-place-update is very simple. Access the drive, read the data, modify the data, write back the data. On the solid state drive this is much more complicated. The data is accessed, read and modified, but when writing the data back internally the solid state drive has another behavior. The data is not written back to the same physical location. The full erase block is updated in the buffer and written to another free erase block. Then the old block is erased. So the simplified steps are: access the data, read the data to main memory, (write the updated data back to the disk), read the full erase block (maybe still in the buffer), update the full erase block in the buffer, write the full erase block to another free erase block (including another access), erase the old erase block and finally update the internal data structures.
SSD
- READ: $T_{seek} + \frac{D}{R_{seq}}$
- INSERT: $T_{seek} + \frac{D}{R_{seq}} + T_{seek}$
- UPDATE: $T_{seek} + \frac{D}{R_{seq}} + T_{seek} + \frac{D}{W_{seq}} + T_{erase}$
- split: $T_{seek} + \frac{D}{W_{seq}} + T_{erase}$

HD
- READ: $T_{seek} + \frac{D}{R_{seq}}$
- INSERT: $T_{seek} + \frac{D}{R_{seq}} + \frac{D}{W_{seq}}$
- UPDATE: $T_{seek} + \frac{D}{R_{seq}} + \frac{D}{W_{seq}}$
- split: $T_{seek} + \frac{D}{W_{seq}}$

Cache Influence
The cache stores some leaves. If the leaf is already in the cache then the read is considered as no costs. If the leaf is not in the cache then additionally to the read costs are the write costs of a leaf to empty some space (write back of an old leaf). Using no cache one immediately has the cost of writing back the data to the drive. But using a cache this costs occur later. Depending on the workload a cache might speed up the throughput. If the distribution of the keys is random then the cache hat not much influence on the throughput. But if there is another workload (for example accessing the same key frequently), then the cache can speed up the performance. Using an LRU strategy the frequently accessed data stays in the cache and must not be fetched from the drive again. Then these updates are done in main memory. The more event that can be done in main memory, the less are the differences between the two drives in the experiments. With a cache the insertion of sorted key-value pairs into an empty index is much faster. Then the leaves are filled in the cache and written when they are full.

4.4.4 Solid State Drive
Sequential read speed: 120MB/s, access latency 0.3 ms, sequential write speed 45MB/s.
- READ: 0.3ms + 0.26ms = 0.56ms.
- INSERT: 0.3ms + 0.26ms + 0.3ms + 11.36ms + 1.5ms = 13.72ms
- UPDATE: 0.3ms + 0.26ms + 0.3ms + 11.36ms + 1.5ms = 13.72ms

Calculating the average (with the read workload):
$0.56*0.8+13.72*0.1+13.72*0.1 = 3.19ms \rightarrow \text{throughput} = 313$
4.5 Memory Usage

4.4.5 Hard drive

Read speed: 115MB/s, Access Latency 9ms, Write speed 120MB/s.

- READ: $9\text{ms} + 0.27\text{ms} = 9.27\text{ms}$.
- INSERT: $9\text{ms} + 0.27\text{ms} + 0.26\text{ms} = 9.53\text{ms}$
- UPDATE: $9\text{ms} + 0.27\text{ms} + 0.26\text{ms} = 9.53\text{ms}$

Calculating the average (with the read workload):

$0.8 \times 9.27\text{ms} + 0.1 \times 9.53\text{ms} + 0.1 \times 0.53\text{ms} = 9.32\text{ms} \rightarrow \text{throughput} = 107$

These values are simplified estimates of the maximum possible throughput performance for each drive. They were calculated based on the model and the drive specifications. The split operation cannot be modeled easily and is thus left away which does not influence the calculations a lot since a split operation does not occur frequently.

Results

Figure 4.12 shows the experiment results. When the index structure starts 100MB of data is inserted sorted. Since there is a LeafCache this is very fast on both drives. The leaves are filled in the cache and persisted when a split occurs. So the data can be written sequentially. Without a cache the leaf must be persisted every time it is updated and inserting 100MB would need a lot of time. When the index is filled the defined workload starts. On both drives there is a constant throughput. The solid state drive reaches a throughput of 286 while the hard drive can only process 75 events per second. This is a huge difference. Whenever a leaf has to be fetched from the drive, an old leaf has to be persisted to the drive and the new leaf is read to the cache, which causes the costs. If the leaf is already in the cache no real costs occur for updating or reading.

Another interesting point to see is that the hard drive buffer does not help that much in this index structure. The more data in an index, the less this hard drive buffer can help in a totally random workload.

4.5 Memory Usage

The application can compute the overall memory usage. It could be written in C much more efficiently leaving away the object orientation. The memory that is consumed in the index structures is calculated in the following way:

4.5.1 CSS-tree hierarchy

The total used memory in the CSS-tree hierarchy is the memory used for the internal nodes and the BTree delta:

$\text{CSSInternalMemory} + \text{BTreeLeavesMemory} + \text{BTreeInternalNodesMemory}$
**Index Structures**

BTree: a BTree consists of LeafNodes and InternalNodes.

**LeafNode**
- data storage: LeafSize * tupleSize [int]
- int currentSize: 1 [int]
- nextPointer: 1 [int]

A LeafNode uses a variable to store its size and the next LeafNode. The memory used for this variable is one int each. Then a LeafNode stores the data and uses the space LeafSize * tupleSize for it. LeafSize is the maximum leaf size given as a parameter in the initialization of the BTree. With these parameters the application can calculate the total used memory for a LeafNode in Bytes (shift by 2: ‘<< 2’):

→ Total size [Bytes/Leaf]: ((LeafSize * tupleSize) + 2) << 2

**InternalNode**
- key[]: size of the node [int]
- Pointer[]: size of the node [int]
- int currentSize: 1 [int]
- nextPointer: 1 [int]

→ Total size [Bytes/InternalNode]: ((InternalNodeSize * 2) + 2) << 2
4.6 Analysis

4.5.2 BDiskTree

The total memory used in the BDiskTree index structure is: LeafCacheMemory + BDiskTreeLeavesMemory + BDiskTreeInternalNodesMemory

**LeafNode**
- long diskPosition: 2 [\#int]
- int currentSize: 1 [\#int]
- nextPointer 1 [\#int]

→ Total size [Bytes/InternalNode]: \(4 \ll 2\)

**InternalNode**

→ The same as in BTree.

**LeafCache**

→ Total size in Bytes: \((\text{currentCachedLeaves} \times (\text{LeafSize} \times \text{tupleSize})) \ll 2\)

The application can then calculate the overall memory usage that is used to maintain both indexes. This allows us to give the same amount of memory to both indexes for the comparison. The memory size of the B\(^+\)Tree delta and internal nodes of the CSS-trees is never more than 2MB in the experiments. So we can give the same 2MB to use for the internal nodes and the LeafCache in the other index structure.

4.6 Analysis

Comparing the two index structures we can say that the B\(^+\)Tree index will have a constant throughput. The operations mostly have constant costs depending on the drive performance specifications. In the CSS-tree hierarchy structure instead the throughput varies a lot. It depends on the number of CSS-trees that are actually in the list which changes over time. The more CSS-trees the more one single lookup costs and the less throughput can be measured. The write operations are collected in a B\(^+\)Tree delta index. Merging them with the given structure does not cost very much. From time to time this update has to be executed. During this merge no event can be processed (in the current implementation). The throughput also depends on the merge strategy. In our experiments we had a 2-way merge strategy. In our read-mostly workload a merge strategy with a larger \(k\) parameter would be much worse. Then more trees have to be traversed in a read request. With much data, there might be a configuration when more than 8 trees are in the list. Then this index performs worse than the B\(^+\)Tree index until the next merge occurs. One could think of a new merge strategy that does not consider the levels of the trees but rather use the number of trees in the list. For example whenever there are 3 trees in the list they must be merged. Then a minimum
throughput could be guaranteed no matter how big the data is. One could argue doing an immediate merge strategy. Then the cost of traversing the CSS-tree hierarchy is always the lookup of 1 tree, which is the best we can get on the tree hierarchy. It would no longer be a hierarchy, but instead only 1 large CSS-tree is used. After each merge one new CSS-tree would be generated, which might be a bit overhead.

The B⁺Tree index on the contrary uses an update-in-place strategy for write operations. Each time a write request arrives the index or the cache is updated. If no cache is used the full index lies on the drive. To construct the full index in the CSS-tree hierarchy the in-memory B⁺Tree and all the available CSS-trees have to be merged.

4.7 Conclusions

In this chapter we built two different index structures and tested them on a solid state drive and a conventional hard drive. We measured the throughput (the events that can be processed each second) to compare the indexes. From the experiments we have seen that this throughput is limited by the random accesses a drive has to do. Especially on the hard drive this is the huge limitation. On the solid state drive it is much cheaper to do a random access.

In the experiments we have seen the differences of a solid state drive compared to a conventional hard drive. A conventional hard drive does not suffer much from in-place updates. The solid state drive instead has problems with in-place updates. When writing back small data to the drive it internally suffers under the properties of flash memory. The erase problem drastically reduces the overall write performance, especially for small random write requests. In our experiments we could reach a much higher throughput on our solid state drive using a delta index approach. In this approach new data is gathered in the delta. Then it can be written in a large sequential request. This is the best write pattern for a solid state drive.

When doing read requests some random accesses have to be made, but on the solid state drive this costs are much lower than on a conventional hard drive. The more trees in our index structure, the more of these lookups have to be made and the throughput gets lower. We might think of other merge strategies to adapt to this problem.

In both experiments the solid state drive could achieve a much higher throughput (more than factor 3) than the conventional hard drive. This is because its access latency is very small. In a typical index structure some random accesses have to be made to read or update data. On a solid state drive this can be made faster than on a hard drive. These results clearly state that solid state drives have the potential to be used in high performance server architectures.

In the mean time there are high performance solid state drives which have specifications high above ours. In our experiments we tested the basic throughput performance of typical index structures. From the results we deviate that a solid state drive can compete with a traditional hard drive for many index structures used in running server systems, or even introduce a better performance depending on the workload. Adapting index structures to the new characteristics of the solid state drive could even improve their performance.
5 Conclusions

5.1 Summary

In this thesis we presented a lot of information about solid state drives. In chapter 2 the properties of flash memory have been discussed in details. By introducing wear leveling techniques the lifetime of flash memory cells can be enhanced and flash memory can thus be used in solid state drives. Internally a solid state drive contains a controller that efficiently tries to manage the I/O requests. The read performance of solid state drives is high, but the write performance is much lower.

In chapter 3 we spent some time analyzing the external sorting algorithm. We implemented a test suite that executes the basic external sorting algorithm. The data is stored on the given drive. The algorithm contains the 3 phases: run generation, multilevel merge, final merge.

In chapter 4 we constructed two different index structures. The first structure is a CSS-tree hierarchy which stores the data in the leaves on the drive. An additional delta index is kept in-memory and merged with the CSS-tree hierarchy when it is full. The second index structure is a conventional B+Tree. The data of each leaf is persisted on the physical drive. This structure contains an additional cache to buffer some leaves. An LRU strategy is used to determine which leaves to keep.

5.2 Concluding Remarks

Solid state drives are a relatively new technology. We have seen that the performance measurements of flash memory devices are not straightforward. Solid state drives are in a way like a black box. The vendors keep the details of their internal life secret. Nevertheless in our performance measurements we discovered that a solid state drive is very good in small random reads. The access latency is much much faster compared to a traditional hard drive. This is the big improvement. Our model had an access latency of 0.3ms, the hard drive had 9ms. On the contrary the write speed is much lower. In small random writes a solid state drive reaches a very bad performance due to the erase problem.

We implemented the external sorting algorithm and experimented on both drives. For traditional hard drives a multilevel merge step avoids too much randomness in the I/O patterns. It pays off to do some merge steps in between. For the solid state drive instead we found a new strategy. The cost of an access is low and therefore a solid state drive can sort faster by merging all the runs in one step. No multilevel merge has to be done. We expected to see a difference. But from this discovery we derive that other algorithms might also be changed to achieve a better performance on a solid state drive.
The focus of this thesis lies on indexing on flash-based solid state drives. With the knowledge from the previous chapters we implemented two different index structures and tested them on both drives. In the B+Tree index structure a constant throughput was reached. The update-in-place structure is generally bad for the solid state drive. Internally it cannot manage this requests efficiently. Still the performance of the solid state drive was higher due to its much lower access latency. The introduced leaf cache does not have much influence on totally random workloads. But accessing the same keys frequently will improve the throughput of the tests of both drives. The more that can be done in the cache (in memory), the more alike is the behavior of both tests.

The CSS-tree hierarchy index structure is different. A delta index is kept in-memory and merged when it has the specified maximum size. In this index structure the throughput depends on the number of trees the list contains. Data is collected and written in larger sequential chunks. This is the best write pattern we can do on the solid state drive, therefore a solid state drive favors a delta indexing approach.

Comparing both indexes the test results showed that a higher throughput can be reached in a delta indexing approach. This approach avoids small random writes (in-place-update) which are very bad on a solid state drive.

On all these experiments the solid state drive reached a better throughput performance (by more than the factor 3). Thus using a solid state drive on a database server might enhance the throughput. Many algorithms and data structure have been explicitly tuned for hard drives for years to avoid disk seeks. But a solid state drive has other performance characteristics. A drive access is cheap. We have seen that solid state drives are fast or even faster than hard drives. But adapting data structures and algorithms to the new characteristics of solid state drives may even improve their performance.

During the 6 months of this thesis the market of solid state drives advanced a lot. The density nearly doubled. Now there are drives with 250GB on the market. The performance also made huge progress. In the meantime there are high performance solid state drives solutions which achieve the following performance (according to the specifications): 250MB/s read, 160MB/s write, 0.1ms access latency. Having such performance requirements they can compete with the best available performance tuned hard drives.

Today solid state drives are not widely used in server architectures. Flash memory is used in many mobile devices and also in notebooks. But we think that this is just the beginning of this technology. In some years solid state drives (or other flash-based devices) will be used in most computer systems. Flash memory offers great improvement in terms of performance, power consumption and usability. The technology is new and will improve very fast. Data structures and algorithms (maybe even file systems) might be adapted to support the full characteristics of flash memory. Tape is dead, disk is old. The next storage device in the line is flash memory.
5.3 Future Work

Solid state drives do not exist for a long time. There is a lot of research one can investigate in this area:

**Benchmark for solid state drives:** There are many tools for testing the performance of conventional hard drives. But applying these to solid state drives does not always show the whole truth. New testing tools, which understand the functionality of solid state drives, might be better able to compare two different solid state drives but also a solid state drive with a conventional drive. A very new paper which tries to understand I/O patterns of solid state drives and build a benchmark is uFlip [5].

**Test other drives:** In the meantime there are a lot of different drives on the market. Many vendors produce solid state drives. Their performance specifications vary a lot. An obvious step would be to execute our experiments on other solid state drives and hard drives.

**SLC / MLC:** In our experiments we only had a mainstream solid state drive containing MLC memory flash cells. On the market there are new high performance solid state drive which use SLC flash memory cells. It would be interesting to investigate some time to find the performance differences between such drives. Some workloads might perform better on one type of drives.

**Test other algorithms and data structures:** We specifically looked at the external sorting algorithm and two different index structures. There are other algorithms and data structures of interest that could be optimized for the use on a solid state drive. When solid state drives replace hard drives then one definitely has to consider this at the latest.

**Maximum performance test:** Solid state drive technology advances very fast. We did not have much hardware available. The newest high performance solid state drives are still expensive. One could buy the very best solid state drive and make a competition against the best hard drive on the market. This would be an interesting setup. what is the maximum performance that can be reached? Can the solid state drive compete? Which workloads are better? Where do the drives suffer? What are the results on a real-life workload?

**Server architecture testing:** Very interesting would be the use of solid state drives in common server architectures. Usually such architectures contain high performance hardware. Solid state drives are not yet used often in such architectures. But recently there emerged some high performance solid state drives on the market which could fulfill the needs of such server structures. Using solid state drives one may improve the performance and throughput of such architectures. Further investigation is needed to really compare solid state drives with conventional hard drives in this area.

**File systems:** On many solid state drives a file system is used. This is not optimal. The file system is not aware of the underlying flash memory and is tuned for the use on top of hard drives. Using a flash file system on a solid state drive is also not optimal.
The controller already implements the functions to maintain the flash memory. Here we might be interested to find a good solution. There is a huge variety of file systems. One could try to find the file system that mostly supports the characteristics of flash memory. Is there a solution using a flash file system and Memory Technology Devices? Is there a file system that does not have much overhead when using it on top of a solid state drive? Or must this be newly implemented?

**Energy Consumption:** Solid state drives are said to consume less energy. We could not investigate a lot in this direction. Having the necessary tools one could build two identical systems except for the drives and measure their power consumption in typical but also real workloads. Nowadays power consumption is a constraint to be respected. Solid state drives might save a lot of energy compared to conventional hard drives and still reach a good performance.
A Test Environment

A.1 Software

The testing software is all written in Java. We use the Sun Java version 6:

SSDT:/home/fkeusch/codesvn# java -version
java version "1.6.0_07"
Java(TM) SE Runtime Environment (build 1.6.0_07-b06)
Java HotSpot(TM) Server VM (build 10.0-b23, mixed mode)

Time is measured as the duration of nano seconds a test run needs to complete the write/read requests. The measurement is done with the System.nanoTime() method. In the tests there is no file system on the disk. Any layer in between could distort the measurements. The data is directly written to the raw device at the specified position. For this the class RandomAccessFile.java and the Java NIO classes are used.

The test environment runs a Debian Linux operating system. The running version is the current Debian etch release with the kernel 2.6.18-6-686:

SSDT:~$ uname -a
Linux SSDT 2.6.18-6-686 #1 SMP Mon Oct 13 16:13:09 UTC 2008
i686 GNU/Linux

A.2 Hardware

A.2.1 Test Computer

The computer where the tests are running has the following specifications:

- 2GB Ram
- 2 Intel cores 6400 @ 2.13GHz
- 2MB CPU Cache

The operating system is installed on an additional hard drive.

All the devices are connected with the SATA computer bus interface.
A.2.2 Hard Drive

The hard drive is from the vendor Western Digital with the following specifications:

<table>
<thead>
<tr>
<th>Specification</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rotational Speed</td>
<td>7200 RPM</td>
</tr>
<tr>
<td>Capacity</td>
<td>320 GB</td>
</tr>
<tr>
<td>Cache</td>
<td>16MB</td>
</tr>
<tr>
<td>Format</td>
<td>3.5&quot;</td>
</tr>
<tr>
<td>Average seek time</td>
<td>8.9 ms</td>
</tr>
<tr>
<td>Model Number</td>
<td>WD5000AAKS</td>
</tr>
<tr>
<td></td>
<td>SATA</td>
</tr>
<tr>
<td></td>
<td>NCQ</td>
</tr>
</tbody>
</table>

In the report the abbreviation 'HD' is used to address this drive.

A.2.3 Solid State Drive

The solid state drive is from the vendor OCZ technology (see Figure A.1) with the following specifications:

<table>
<thead>
<tr>
<th>Specification</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capacity</td>
<td>32GB</td>
</tr>
<tr>
<td>Format</td>
<td>2.5&quot;</td>
</tr>
<tr>
<td>Average seek time</td>
<td>0.3 ms</td>
</tr>
<tr>
<td>Read speed</td>
<td>120-140MB/s</td>
</tr>
<tr>
<td>Write speed</td>
<td>80-93MB/s</td>
</tr>
<tr>
<td>Model Number</td>
<td>OCSSD2-1C32G</td>
</tr>
<tr>
<td>Series</td>
<td>core series v1</td>
</tr>
<tr>
<td></td>
<td>SATA</td>
</tr>
</tbody>
</table>

In the report the abbreviation 'SSD' is used to address this drive.
Figure A.1: The OCZ solid state drive core series v1 [3].
B  Documentation

B.1  NIO - New I/O Java Classes

We tried to overcome the kernel buffering using the NIO Java classes. The NIO interface is very complicated and unintuitive to use. With a ByteBuffer some space can be allocated: ByteBuffer.allocateDirect(byteBufferSize). This space is then allocated directly in the memory and outside the JVM heap (it should even not go through the system I/O buffers). Using this direct ByteBuffer we hoped to bypass the Linux kernel buffer, but it did not work. The official Java documentation says [45]:

"Given a direct byte buffer, the Java virtual machine will make a best effort to perform native I/O operations directly upon it. That is, it will attempt to avoid copying the buffer’s content to (or from) an intermediate buffer before (or after) each invocation of one of the underlying operating system’s native I/O operations”

We could not find a way to bypass the kernel I/O buffering using only the NIO Java classes. There was still buffering whatever we tried. We are not sure how this should work, but somehow there must be the possibility to write directly to a drive without going to the I/O buffers. Of course then one must be aware of the special properties of a block device. In the internet some bugs concerning this allocateDirect() method were reported, but we are not sure how the buffering is affected. We could not find useful information of how to do direct I/O using Java as the programming language.

In the internet there was also a kernel patch for very old kernel versions to disable some caching (not tried). Finally we found a way using the deprecated Linux raw interface.

B.2  Raw Interface

In our tests we used the Linux raw interface to overcome the kernel buffering. This raw interface is used to bind a device as a raw block device. Accessing the raw device bypasses the kernel’s block buffer cache entirely. All the I/O is done directly to and from the address space. But there are a few restrictions when accessing a raw binded device:

- The I/O must be correctly aligned to 512 Bytes.
- An I/O request size must be a multiple of 512 Bytes.

A sector on a raw device has the length 512 Bytes. With these restrictions it is for example not possible to read 256 Bytes from the position 32. Instead the software has to read from 512 Bytes from position 0 and then cut out the interesting parts. The write is even more complicated. When writing only 4 Bytes then internally the 512 Byte sector has to be loaded, updated and written back. This is normally done behind the scenes by some I/O buffers.
Note: This interface is deprecated since the Linux kernel 2.6.3. It should not be used anymore. They advice to modify applications to open the block device with the O_direct flag (C programming language). Anyway to still bind our two devices with this Linux raw interface we use the following script (makeRaw.sh):

```bash
#!/bin/sh
# script to bind raw devices.
# note: the raw interface is deprecated.

# load raw module
modprobe raw

# make folder
mkdir /dev/raw

# make nodes
mknod /dev/raw/raw0 c 162 1
mknod /dev/raw/raw1 c 162 2

# make symbolic link
ln -s /dev/rawctl /dev/raw/rawctl

# mount devices as a raw device
# mount hard drive
raw /dev/raw/raw0 /dev/sdc1
# mount solid state drive
raw /dev/raw/raw1 /dev/sdb1
```

Using this raw interface to bind our devices we can be ensured that no kernel buffering is done. This helped a lot in our experiments. Otherwise the kernel buffer would cache a lot of data and fill up the whole available RAM. This would disturb the experiment results. Most of the data could be recovered from the kernel I/O buffer and is not read from the real drive.

### B.3 Open Files Limit

When working with many queues you may get an exception ‘Too many open files’ and the application aborts. The standard configuration has a default number of 1024 files that are allowed to be opened simultaneously. For some running applications this is not enough. You can see your actual number with the command:

`'ulimit -n'`

You can change this number by editing the file /etc/security/limits.conf.

* `soft nofile 4096`
* `hard nofile 4096`

Save and reboot to load the new configuration.
C Source Code Overview

The whole software written in Java is stored in the SVN repository in the folder src. In the folder scripts there are small bash scripts that execute the Java code.

C.1 Performance Experiments

The package javaCode.ioTesting holds the classes used for the performance experiments.

**IOTesting_raccess.java**: This class is used to test the access latency of the disk. All the parameters can be set inside this class.

**IOTesting_raw.java**: This class tests the sequential I/O pattern on a drive. All the chunk-sizes to test are stored inside in different arrays.

**IOTesting_raw_random.java**: There are multiple versions of this class. It tests the random I/O of a drive. All versions have a slightly different method to calculate the time or drive access positions.

**IOTesting_raw_random_file.java**: This class does the random tests on a file system. The device path parameter is a file.

**IOTesting_atto.java**: This class simulates the ATTO benchmark.

C.2 External Sorting

The classes that correspond to the external sorting are located in the package flash.externalsort. It contains the following subpackages:

**flash.externalsort.arrayObjects** contains the old version of the external sorting that is all done with objects. This version has a very high object overhead and is therefore just too slow.

**flash.externalsort.arrayBasic** contains the classes that are used for the external sorting algorithm.

**flash.externalsort.arrayBasic.test** contains all the testing classes that can be executed with some parameters.

**flash.externalsort.nio** contains the classes that persists the integers to sort.
The most important classes are:

**NIOIntArrayPersister.java**: This class executes I/O requests on a drive. It reads/writes arrays of the primitive type int from/to the specified position.

**ExternalIntArrayQueue.java**: This class represents one queue, one run. A run can be defined by 3 parameters: the path to the drive, the position where it starts on the drive and the number of elements it contains. Read and write can only be done in int[]. This class contains a persister to do the I/O requests.

**IntArrayWrapperQueue.java**: This class wraps a run. It contains a buffer to read/write a chunk of integers from the underlying run, but then it can communicate with its clients element-wise. This class can be initiated to either read or write data, but it cannot be mixed.

**IntMerger.java**: This class takes several wrappers and prepares them for merging. The client can get the next element until all elements from all the wrappers have been consumed. Then it is empty.

**IntArrayMultiLevelMerger.java**: This class does the multi level merge step. It always merges the smallest runs into new bigger runs until the appropriate fanIn is reached for the final run.

**ExternalIntArrayMemorySorter.java**: Here all the sorting is initialized and started. This class initializes the multi level sorting. At the end it can give back the final run online.

**ExternalIntArrayMemorySortTester.java**: This is the class that does all the testing. Initializes all the parameters and does an online sorting at the end.

**ExternalIntArrayMemorySortTesterWithFinalWrite.java**: Does the same thing as the class above but writes at the end the final run to the drive.

**ExternalIntArrayMemorySortTesterMultiDisk.java**: This class takes two disks to sort. On the first drive (HD) all the data is stored. Then it generates the sorted runs on the second drive (SSD). In the next step it merges all the runs with a big fanIn and writes the data back to the initial disk (HD).

**ExternalIntSortConstants.java**: Contains many constants and parameters which can be set for the external sorting application.

**AbstractTest.java**: This abstract class is used to specify a test using the external sorting implementation.

**TestSorting.java**: This specific class is a test class that uses the given parameters to execute the external sorting algorithm. The parameters are read from the arguments.

**TestSortingMultiDisk.java**: This class starts the sorting process using two different disks.

**TestSortingWithFinalWrite.java**: In this class the final merge is written back to the drive.

**TestElements.java**: This class executes different sorting tests varying the data size.
TestFanIn.java: In this class different sorting tests are executed varying the fanIn.

TestRunSize.java: In these tests the runSize is varied.

General classes that are used in both implementations:

javaCode.utils.TimePrinter.java: This class is used to print the actual time duration. At the initialization the time is stored, then the actual duration can be calculated consecutively.

javaCode.utils.FileLogger.java: This class can be used very simple to log some strings. After the initialization clients can just append strings and they are written to the specified file on the system drive.

javaCode.utils.VMLogging.java: This class is used to start and stop the logging of vmstats. In a separate thread the 'vmstat' logging is started with the given command and written to a file.

C.3 Index Structure
This section contains the class description for the event generation and the two index structures CSS-tree hierarchy and BDiskTree.

C.3.1 Event Generation
For the event generation the following classes located in the package flash.indexing.event are used:

Event.java: This class just holds the different event types.

EventObjectBlock.java: In this class multiple events are stored as a block.

EventBlockGenerator.java: Class that defines a block generator.

EventBlockGeneratorStrategy2.java: This class is a specific block generator that implements a certain strategy. In this strategy it is possible to define a certain workload and to fill the index structure at the beginning.

EventBlockGeneratorStrategy3.java: Is nearly the same as the strategy2, but inserts the elements sorted. This means it generates the data, sorts the data and is then ready to fill the values.

EventQueueManagerBlock.java: The main class of the event generation. It extends the class Thread. This class has a queue that stores new event blocks. After the start this class generates new events when the queue is not full. This class also logs the events using the helper class flash.indexing.utils.EventLogger.
C.3.2 CSS-tree hierarchy

The classes that are used for the implementation of this index structure are located in the package `flash.indexing.index`:

**IndexStructureConstants.java**: This class contains many constants that are used in the application. At the beginning these constants are loaded. Using this class it is easy to change parameters for the whole application.

**StartIndexStructureDisk.java**: This class is used to start the application with the CSS-tree hierarchy. It starts the two threads: the event generator and the index structure manager.

**IndexStructureManagerDisk.java**: This is the index structure manager. It extends the class `Thread`. It evaluates new events and executes the correct action according to the event definitions.

**CSSTreesManager.java**: This is the general implementation of the CSS-tree hierarchy. It contains the needed fields, parameters and abstract methods.

**CSSTreesManagerDisk.java**: This is the disk implementation for the CSS-tree hierarchy manager. It manages the CSS-tree hierarchy that is stored on a drive. A key can be queried. Or a full bTree can be merged with the actual CSS-tree hierarchy. Then first the data of the bTree is persisted in a new CSS-tree of the lowest level. In a second step all the CSS-trees are gathered that must be merged according to the defined k-way merge parameter. If there are trees to merge, then they are wrapped and merged. The merged data is written to a new updated CSS-tree. Its level is one higher than the highest CSS-tree in the merge phase.

**CSSDiskTree.java**: This class represents a CSS-tree on a drive. It contains a position and a service to do the I/O request. Every instance of this class contains the same `BlockIOService` and no buffering. Every time this tree is accessed the data is read from the drive. The internal nodes are not stored on the drive. They are inside this class as an array of the primitive type `int`. Whenever a key is searched, then the internal nodes are searched using the binary search algorithm. This gives back an index that indicates where the leaf lies on the devices. With this position the data is fetched from the drive and the leaf can be searched using the binary search algorithm again. An age indicates the logical age of this tree which is used for the merge process.

**CSSDiskTreeMerger.java**: This class can merge multiple `CSSDiskTree` classes. It is used in the merge step of the `CSSTreesManagerDisk`. In the merge one has to consider that in a newer `CSSDiskTree` a key with the newer value could lie. To consider this an age was added to the class `CSSDiskTree`.

**CSSTreeWrapper.java**: This wrapper class wraps a `CSSDiskTree`. It manages the element-wise processing of the merge phase. Internally it contains a buffer to read blocks of elements. It implements the comparable interface to ensure the comparison of trees. Updated elements have to be filtered and the newer value has to be taken in the merged tree.
**BlockIOService.java**: This class manages all the I/O requests that occur in the application. It contains read and write methods. There is only one instance of this class in the CSS-tree hierarchy. No buffering is done. Every request goes to the drive. The devices are accessed via the Linux raw interface and no kernel buffering is done.

For this implementation the classes from the event generation package are used to generate the events and do the workload.

The package `flash.indexing.index.bTree` contains the bTree taken from Stefan Hildebrand’s master thesis [22].

The package `flash.indexing.index.helper` contains some small helper classes to define an index.

The classes of the inMemory implementation lie in the package `flash.indexing.index.inMemory`:

**CSSTree.java**: The inMemory CSSSTree taken from the implementation of Stefan Hildenbrand’s master thesis [22]. Small changes are made to make this tree comparable and to be able to go through the content.

**SortedArray.java**: The CSSTree.java has this class as the underlying implementation.

**CSSTreeMerger.java**: This class can merge CSSTrees. Since the CSSTrees implement a comparable interface it can manage to take only the newest updated value in the merge.

**CSSTreesmanagerInMemory**: This class manages the CSSTrees and does the merges. It can query a key to the in-memory hierarchy.

**IndexStructureManagerInMemory.java**: The application with the in-memory implementation can be started with this class.

### C.3.3 DiskBTree

The classes that are used especially for the implementation of this index structure are located in the package `flash.indexing.index2`:

**DiskBTree.java**: This is the BTree taken from Stefan Hildebrand’s master thesis [22] adapted to write on the disk. The package `flash.indexing.index2.diskBTree` contains the rest of the needed classes. In most of these classes only small changes were made comparing to the existing in-memory implementation. The big change lies in the class listed below, where the data is persisted to a physical drive.

**LeafArrayMap.java**: This class contains the most changes compared to the in-memory implementation. It stores the data of a leaf on the drive. To do this it needs an additional position. To make the smallest changes on the overall BTree implementation the class below is used. The data is then linked from this class into the LeafArrayMap. To do
In this the two methods `load()` and `unload()` were added. Basically they do nothing else than load and link the data when needed and unload the data afterwards. This class accesses the `LeafCache` that manages the data persistence.

**LeafCache.java**: This class caches some leaves. It implements a cache between the index structure and the physical drive. The strategy that is used inside to choose the leaves to keep is LRU. It uses the `StaticBlockIOService` to read and write leaves.

**StaticBlockIOService.java**: This class does the I/O requests on a drive. It will read the keys and values from the drive and store them in local arrays. These can be used to manipulate (e.g read, update and write back).

**IndexStructureConstants.java**: This class contains many constants that are used in the application. At the beginning these constants are loaded. Using this class it is easy to change parameters for the whole application.

### C.3.4 Helper Classes

General helper classes that are used in both implementations:

**flash.indexing.utils.EventLogger.java**: This class enables simple logging. A logging can be started calling the `start()` method and giving a filename as parameter. Then simple strings can be appended to the log file.

**flash.indexing.utils.TimeConverter.java**: This class stores the time at the initialization. Then it can simply give back the duration from the stored time to now. This is very handy when one need to report the time in an application.
Bibliography


