A Bare Minimum Tutorial on Hard Disks

1 Hard disk

Hard disk sectors are units for read/write - access on a given device. It is a fixed size subunit of a track. Typically is 0.5 to 4KB. Sectors are stored with self-correcting error codes so the controller is able to detect if error has occurred.

There are two important things: 1) Detecting error and 2) Data recovery. We are detecting the error by writing the block and then verifying it by reading it. When error occurs, hard disk controller can flag the failing block and can remap it to different physical position. This is useful feature of the hard disk but it might affect the performance if there are too many of these blocks. This remapping is done transparently without the software layers on top having to do anything about it as controller handles it transparently.

Hard disks often have cache. This cache has it's own caching policy of caching access requests. Upon read request, controller usually tries read entire track into the cache assuming we are going to read local physical data in a near future. On writing request, cache can be used to store written data and writes it later. Hard disk controller can reorder the requests based on some optimization algorithm, often it's elevator optimization.

The disk cache is usually quite small, ranging between 8 and 256 MiB. The data in the disk buffer is rarely reused. In this sense, the terms disk cache and cache buffer are misnomers; the embedded controller's memory is more appropriately called disk buffer.

Figure 1: Hard disk sectors

2 Block addressing: CHS & LBA

2.1 Cylinder-head-sector (CHS)

CHS is an early method of giving addresses to each physical block of data on a hard drive. It is a 3D-coordinate system made out of a vertical coordinate head, a horizontal (or radial) coordinate cylinder, and an angular coordinate sector. CHS address were "exposed" because early hard drives didn't come with an embedded disk controller. Later, CHS was replaced by LBA.

Translation from CHS to LBA address formula is:

$$A = (c \times Nheads + h) \times Nsectors + (s - 1)$$

2.2 Logical block addressing (LBA)

Logical block addressing (LBA) is a common scheme used for specifying the location of blocks of data stored on computer storage devices. LBA is a particularly simple linear addressing scheme; blocks are located by an integer index, with the first block being LBA 0, the second LBA 1, and so on. Hard disk controllers maps LBA to its physical CHS. We do not have the control over the mapping but we can make an assumption that if two LBAs are close to each other we can tell that two physical sectors are also close.

3 S.M.A.R.T.

SMART stands for Self-Monitoring, Analysis and Reporting Technology. SMART is a system monitoring and early detection of errors of hard disks. The data provided by SMART attributes are different from manufacturer to manufacturer and only few manufacturers document the importance of individual SMART attributes in detail. Because of that, it is very difficult for the user to interpret the SMART attributes correctly.

SMART attribute contains:

- Attribute name.
- Raw value, often physical quantities.
- Normalized value, value between 1 and 253 (best condition), most manufactures use 100 or 200 as a best value.
- The worst normalized value.
- Limit for the normalized value, when it falls below the normalized value of this limit, "Dev fail" is set as SMART status.
- Status flags.

SMART has a few problems:

- Incorrect thresholds. Devices often fail before reaching this point.
- Weight of attributes. Different attributes affect disk health differently, some are very critical (10 - spin retry count for an example) and a small change may indicate a serious problem.

4 The File Allocation Table

FAT is one of the most basic file system architectures. The disk is divided into clusters. The number of sectors per cluster is given in the boot sector byte 14. The File Allocation Table has one entry per cluster, so this entry can use 12, 16 or 28 bits for FAT12, FAT16 and FAT32.

Region	Size in sectors	Contents
Reserved sectors	(number of reserved sectors)	Boot Sector
		FS Information Sector (FAT32 only)
		More reserved sectors (optional)
FAT Region	(number of FATs) * (sectors per FAT)	File Allocation Table #1
		File Allocation Table #2 ... (optional)
Root Directory Region	(number of root entries * 32) / (bytes per sector)	Root Directory (FAT12 and FAT16 only)
Data Region	(number of clusters) * (sectors per cluster)	Data Region (for files and directories) ... (to end of partition or disk)

Figure 2: FAT filesystem layout

4.1 The first two FAT entries

In the first byte of the first entry a copy of the media descriptor is stored. The remaining bits of this entry are 1. In the second entry the end-of-file marker is stored. The high order two bits of the second entry are sometimes, in the case of FAT16 and FAT32, used for dirty volume management: high order bit 1: last shutdown was clean; next highest bit 1: during the previous mount no disk I/O errors were detected.

The first cluster of data area is actually cluster #2.

4.2 FAT12

Two FAT12 entries are stored into three bytes, let them be *ab, cd, ef* then the entries are *dab* and *cef*. Possible values for an entry are: 000: free, 002-fef: cluster in use; the value given is the number of the next cluster in the file, ff0-ff6: reserved, ff7: bad cluster, ff8-fff: cluster in use, the last one in this file. Since the first cluster in the data area is numbered 2, the value 001 does not occur.

Offset	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+A	+B	+C	+D	+E	+F
+0000	F0	FF	FF	03	40	00	05	60	00	07	80	00	FF	AF	00	14
+0010	C0	00	0D	E0	00	0F	00	01	11	F0	FF	00	F0	FF	15	60
+0020	01	19	70	FF	F7	AF	01	FF	0F	00	00	70	FF	00	00	00

Figure 3: FAT12 layout

4.3 FAT16

Everything is the same but the entries are now 16bit. Possible values for FAT16 are: 0000: free, 0002-ffef: cluster in use; the value given is the number of the next cluster in the file, fff0-fff6: reserved, fff7: bad cluster, fff8-ffff: cluster in use, the last one in this file.

Offset	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+A	+B	+C	+D	+E	+F
+0000	F0	FF	FF	FF	03	00	04	00	05	00	06	00	07	00	08	00
+0010	FF	FF	0A	00	14	00	0C	00	0D	00	0E	00	0F	00	10	00
+0020	11	00	FF	FF	00	00	FF	FF	15	00	16	00	19	00	F7	FF
+0030	F7	FF	1A	00	FF	FF	00	00	00	00	F7	FF	00	00	00	00

Figure 4: FAT16 layout

4.4 FAT32

Again the entries are 32bit of which the top 4 bits are reserved. The bottom 28 bits have meaning similar to older versions. Microsoft operating systems use the following rule to distinguish between FAT12, FAT16 and FAT32. First, compute the number of clusters in the data area (by taking the total number of sectors, subtracting the space for reserved sectors, FATs and root directory, and dividing, rounding down, by the number of sectors in a cluster). If the result is less that 4085 we have FAT12. Otherwise, if it is less than 65525 we have FAT16. Otherwise FAT32.

Offset	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+A	+B	+C	+D	+E	+F
+0000	F0	FF	FF	0F	FF	FF	FF	0F	FF	FF	FF	0F	04	00	00	00
+0010	05	00	00	00	06	00	00	00	07	00	00	00	08	00	00	00
+0020	FF	FF	FF	0F	0A	00	00	00	14	00	00	00	0C	00	00	00
+0030	0D	00	00	00	0E	00	00	00	0F	00	00	00	10	00	00	00
+0040	11	00	00	00	FF	FF	FF	0F	00	00	00	00	FF	FF	FF	0F
+0050	15	00	00	00	16	00	00	00	19	00	00	00	F7	FF	FF	0F
+0060	F7	FF	FF	0F	1A	00	00	00	FF	FF	FF	0F	00	00	00	00
+0070	00	00	00	00	F7	FF	FF	0F	00	00	00	00	00	00	00	00

Figure 5: FAT32 layout

4.5 Directory Table

A Directory Table is a special type of file that represents a directory. Each file r subdirectory stored within it is represented by 32-byte entry in the table. Root directory table in FAT12 and FAT16 file systems have fixed location before the data area, FAT32 have it stored in data area as it has ability to expand. In case of FAT32, #2 entry is used as pointer to the root directory.

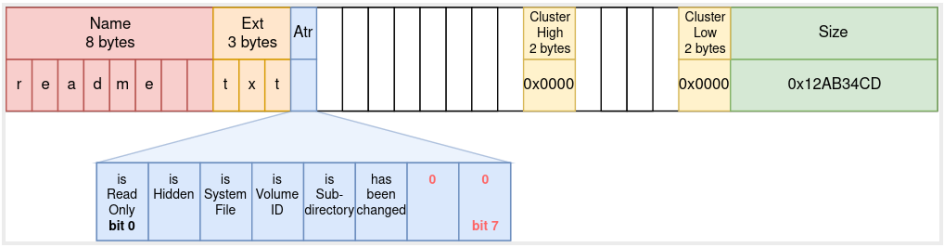

Figure 6: Directory Table entry

A cluster (ID) can be transformed into an LBA value by applying the following formula:

$$lba = cluster\_start\_lba + (cluster\_id - 2) * sectors\_per\_cluster$$

The - 2 is applied because there are no cluster 0 or 1 and cluster's IDs begin from 2. The cluster_start_lba will be obtained from the Directory Table, starting from the root directory table.

5 VFAT long file names

VFAT Long File Names (LFNs) are stored on a FAT file system using a trick: adding additional entries into the directory before the normal file entry. The additional entries are marked with the Volume Label, System, Hidden, and Read Only attributes (yielding 0x0F), which is a combination that is not expected in the MS-DOS environment, and therefore ignored by MS-DOS programs and third-party utilities.

LFN entries use the following format:

Byte offset	Length (bytes)	Description
0x00	1	Sequence Number (bit 6: last logical, first physical LFN entry, bit 5: 0; bits 4-0: number 0x01..0x14 (0x1F), deleted entry: 0xE5)
0x01	10	Name characters (five UCS-2 characters)
0x0B	1	Attributes (always 0x0F)
0x0C	1	Type (always 0x00 for VFAT LFN, other values reserved for future use; for special usage of bits 4 and 3 in SFNs see further up)
0x0D	1	Checksum of DOS file name
0x0E	12	Name characters (six UCS-2 characters)
0x1A	2	First cluster (always 0x0000)
0x1C	4	Name characters (two UCS-2 characters)

Figure 7: Long File Names format

So for an example LARGEFILENAME.SOMETHING becomes LARGEF~1SOM, where ~1 is the shortened entry number in order to support multiple copies with the same short name and SOM is simply the short version of SOMETHING.

A bit detailed example for a filename "File with very long filename.ext" would be formatted like this:

Sequence number	Entry data
0x43	"me.ext"
0x02	"y long filena"
0x01	"File with ver"
???	Normal 8.3 entry

Figure 8: Long File Names format example

6 Volume Boot Record

A volume boot record (VBR) is a type of boot sector that can be found on partitioned data storage device in first sector of an individual partition of the device or the first sector of non-partitioned device. The code is invoked either directly by the machine's firmaware or by code in MBR or a boot manager. In some file systems such as FAT12/16/32, HPFS and NTFS, the VBR also containes BIOS Parameter Block that specifies the location and layout of the principal on-disk data structure.

VBR is loaded at memory location 0000h:7C00h. Before execution, VBR has set up registers:

- CS:IP 0000h:7C00h (fixed) by jumping.
- DL = boot drive unit.

When the VBR is launched by the MBR (or different boot loader):

- DS:SI = points to 16-byte MBR partition table entry.
- AX = magic signature indicating the presence of this extension.
- ES:BX = start of boot sector.

In conjuction with GUID partition table:

- EAX = 54504721h ("!GPT").
- DS:SI = points to a Hybrid MBR handover. structure, consisting of a 16-byte dummy MBR partition.

7 Master Boot Record

A master boot record (MBR) is a special type of boot sector at the very beginning of partitioned computer mass storage device. MBR partition entries and the MBR boot code are limited to 32 bits. Therefore, the maximum disk size supoorted on disks using 512-byte sectors is limited to 2 TiB. The MBR consists of 512 or more bytes located in the first secord of the drive.

It may contain one or more of:

- A partition table describing the partitions of a torage device.
- Bootstrap code: Identify the partition and then load and execute VBR as chain loader.
- Optional: 32-bit disk timestamp, 32-bit disk signature.

Structure of a classical generic MBR:

- 0x0000 = 446 Bytes = Bootstrap code.
- 0x01BE = 16 Bytes = Partition entry.
- 0x01CE = 16 Bytes = Partition entry.
- 0x01DE = 16 Bytes = Partition entry.
- 0x01EE = 16 Bytes = Partition entry.
- 0x01FE = 2 Bytes = Boot signature.

Partition table entries are 16 byte.

- 0x00 = 1 Byte = Status or physical drive.
- 0x01 = 3 Bytes = CHS address of first absolute sector in partition.
- 0x04 = 1 Byte = Partition type.
- 0x05 = 3 Bytes = CHS address of last absolute sector in partition.
- 0x08 = 4 Bytes = LBA of first absolute sector in the partition.
- 0x0C = 4 Bytes = Number of sectors in partition.

8 GUID Partition Table

As the MBR partition scheme imposed limitations for use of modern hardware, a new partition table format was developed. GPT uses 64 bits for logical block addresses, allowing a maximum disk size of 2^{64}. For disk with 512-byte sectors that is 8 ZiB or 9.44 ZB. The protective MBR is stored at LBA 0, and the GPT header is in LBA 1. The GPT header has a pointer to the partition table, which is tipically at LBA 2. Each entry on the partition table has a size of 128 bytes.

8.1 MBR (LBA 0)

For limited backward compatibility, the space of the legacy MBR is still reserved in the GPT specification. A single partition of MBR type, encompassing the entire GPT drive. Legacy MBR contains max. 4 (primary/extended) partitions, if one of them is marked as an EFI System Partition (partition type 0xEF), it is loaded by the UEFI firmware. Legacy MBR may also not contain any EFI System Partition and can be a currently obsolete pure MBR system. In that case there is a boot code in the beginning of MBR, that is loaded and executed. Exceeding the 2 TiB limit could cause compatibility problems.

There is also a Hybrid MBR (LBA 0 + GPT) which is used for operating systems that support GPT-based boot through BIOS services rather than EFI.

Protective MBR contains only 1 partition, and it spans all the storage media (up to the maximum addressable size in MBR, because it is an 32-bit value) and it's type is GPT Protective type 0xEE, and it is now used in a way that prevents MBR-based disk utilities from misrecognizing and possibly overwriting GPT disks.

8.2 Partition table header (LBA 1)

The partition table header defines te usable blocks on the disk and the number and size of the partition entries that make up the partition table.

8.3 Partition entries (LBA 2-33)

The Partition Entry Array describes partition, using a minimum size of 128 bytes for each entry block.

The format of GUID partition entry is:

Offset	Length	Contents
0 (0x00)	16 bytes	Partition type GUID (mixed endian[7])
16 (0x10)	16 bytes	Unique partition GUID (mixed endian)
32 (0x20)	8 bytes	First LBA (little endian)
40 (0x28)	8 bytes	Last LBA (inclusive, usually odd)
48 (0x30)	8 bytes	Attribute flags (e.g. bit 60 denotes read-only)
56 (0x38)	72 bytes	Partition name (36 UTF-16LE code units)

Figure 9: GUID partition entry format

9 ZONE BIT RECORDING

It is a method used by disk drives to optimise the tracks for increased data capacity. The idea is placing more sectors per zone on outer tracks than on inner ones. Older disk drives places the same number of sectors per track with empty seeks between sectors on outer tracks. By using ZBR, we achieve higher throughput on outer tracks. We can utilise this by placing more frequent files on outer zones. This will be further disscussed in implementation details.

10 Implementation details

- Even though /dev/sdxn is part of /dev/sdx, kernel has two different caches and by writing to one doesn't invalidate cache of the other.

  Example: I did open /dev/sdb1/file123.txt and wrote some data.

  Starting address of /dev/sdb1 is 0x11a0000, which means that this particular root entry starting address is $0x1248080 + 0x11a0000 = 0x23e8080$.

  As we can see that is the same physical address but the dump is different as we are doing hexdump of /dev/sdb and /dev/sdb1 cache. We can see that last 4 bytes (size) is totally different as /dev/sdb is outdated.

```
syrmia@SYRN16-0307-L:~$ sudo hexdump -C /dev/sdb -s 0x23e8080 -n 32
023e8080  54 45 53 54 31 32 33 20  54 58 54 20 00 0f fb 6b  |TEST123 TXT ...k|
023e8090  04 55 04 55 02 00 fb 6b  04 55 39 4e 04 00 02 00  |.U.U...k.U9N....|
023e80a0
syrmia@SYRN16-0307-L:~$ sudo hexdump -C /dev/sdb1 -s 0x1248080 -n 32
01248080  54 45 53 54 31 32 33 20  54 58 54 20 00 84 4b 78  |TEST123 TXT ..Kx|
01248090  04 55 04 55 02 00 4b 78  04 55 39 4e 04 00 12 88  |.U.U..Kx.U9N....|
```

Figure 10: /dev/sdb and /dev/sdb1 example.

- We can bypass any kernel caching by using O_DIRECT flag and open system call. O_SYNC is blocking flag used to be sure we wrote both data and metadata to device.

  The O_DIRECT flag may impose alignment restrictions on the length and address of user-space buffers and the file offset of I/Os. In Linux alignment restrictions vary by filesystem and kernel version and might be absent entirely. However there is currently no filesystem-independent interface for an application to discover these restrictions for a given file or filesystem. We can use aligned_alloc with alignment of sysconf(_SC_PAGESIZE) to allocate buffer for read/write purpose with O_DIRECT flag. It is recommended that we use O_DIRECT if our application implements any sort of caching as O_DIRECT gives the best performance.

  fsync and fdatasync are useful syscalls used for synchronizing data + metadata/data in user and disk space.

- In order to control write cache, ATA specification included FLUSH CACHE (E7h) and FLUSH CACHE EXT (EAh) commands. These commands cause the disk to complete writing data from its cache, and disk will return good status after data in the write cache is written to disk media. In addition, flushing the cache can be initiated at least to some disks by issuing Soft reset or Standby (Immediate) command.

- hdparm is powerful tool for verifying disk information. hdparm -W /dev/sdx should give us write caching information. We can turn it on/off. hdparm -f should flush buffer cache to the disk.

- Writing files with different flags:

  - O_DIRECT: If you want to use only O_DIRECT, you should make sure all the storage write caches are crash safe. It is blocking the program on write, but it does not guarantee that the data is written. That's why it is recommended to add a fsync/fdatasync call after a write to a file opened with O_DIRECT.
  - O_SYNC: Synchronizing Linux writing request data and metadata to actual device. Blocking the program on write.
  - O_DSYNC: Synchronizing only data. Blocking the program on write.
  - O_ASYNC: Asynchronous flag doesn't block the program on write system call. We should call fsync/fdatasync as verification of our write.

- Depending on file size, fsync and fdatasync perform differently. The smaller the file the difference between fsync and fdatasync is bigger, at most double. As fdatasync only persist the data to the disk it performs one write operation instead of two for fsync. However that may cause some trouble as explained as the first point of this Notes.

  Below is the example time in seconds for writing 409.5kB file thousand times to the USB device. As we can see, when the file is opened with O_ASYNC, program is blocked on fsync/fdatasync insdead of the write, as opposed to O_SYNC/O_DSYNC, which is blocked on write.

```
running: DIRECT...
        seek time:  0.002541
        write time: 6.351881
        fsync time: 3.443845
time taken: 9.925621

running: FSYNC...
        seek time:  0.002824
        write time: 0.272913
        sync time:  10.164986
time taken: 10.547785

running: FDATASYNC...
        seek time:  0.002680
        write time: 0.259963
        dsync time: 5.725631
time taken: 6.096127
```

Figure 11: File open with O_ASYNC

```
running: DIRECT...
        seek time:  0.002335
        write time: 6.412692
        fsync time: 3.403160
time taken: 9.948404

running: FSYNC...
        seek time:  0.000981
        write time: 10.019415
        sync time:  0.002917
time taken: 10.132172

running: FDATASYNC...
        seek time:  0.001136
        write time: 5.970172
        dsync time: 0.002854
time taken: 6.083866
```

Figure 12: File open with O_SYNC

- Finding track boundaries is used to make zones and find manufacture defects. We can calculate how big is each track by reading consecutive sectors that spans around one revolution of the disk, separated by track skew. Track skew is an unusually large change in angular position between two adjacent sectors. On the graph we have a lot of noise as tested HDD has a lot of bad sectors.

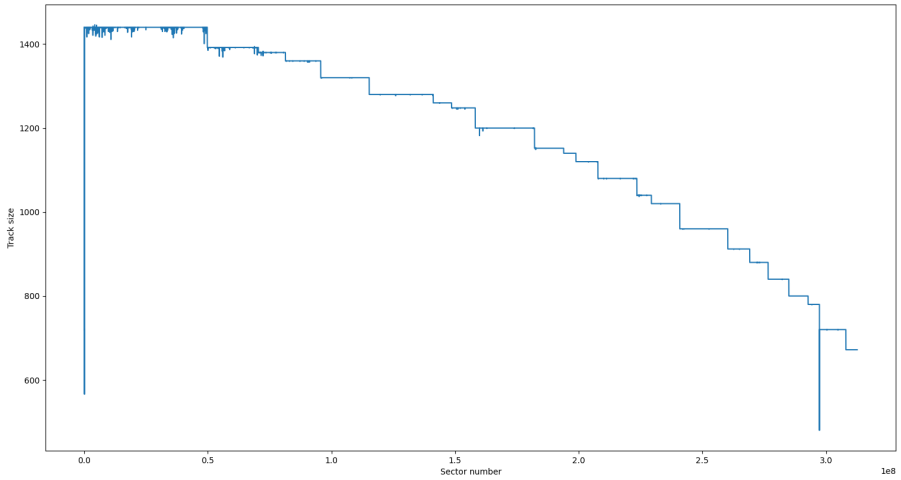

Figure 13: Tracks size

- Sector angular position is an angle between any two sectors and can be calculated by accessing the two sectors and measuring time between the responses. We can calculate the best, the worst and the average access time between sectors but it takes tens to hundreds of measurements per

sector which makes it really expensive. It is advised to be used as examining small parts of disk and plotting the results we can get information about other features, such as defects. It is useful for calculating track skew and track offset as first sectors of tracks are not on the same angular position.

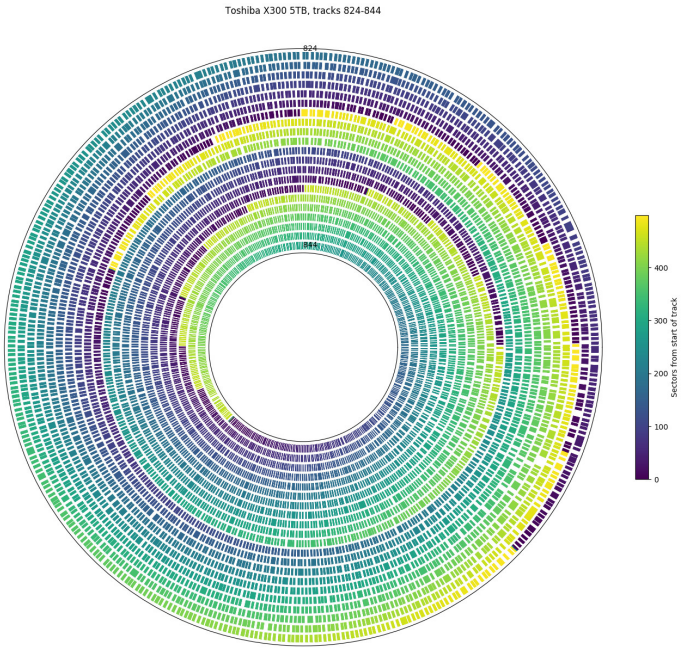

Figure 14: Angular position

- We can use access time from angular position to calculate the nearest free sector, measured in time. Access time is sum of rotational latency and seek time of hard drive so closest sector doesn't have to be physically closest. This is very expensive operation, for an example on Hitachi Deskstar 7K160 it takes on average 0.04s to calculate access time from a reference sector.

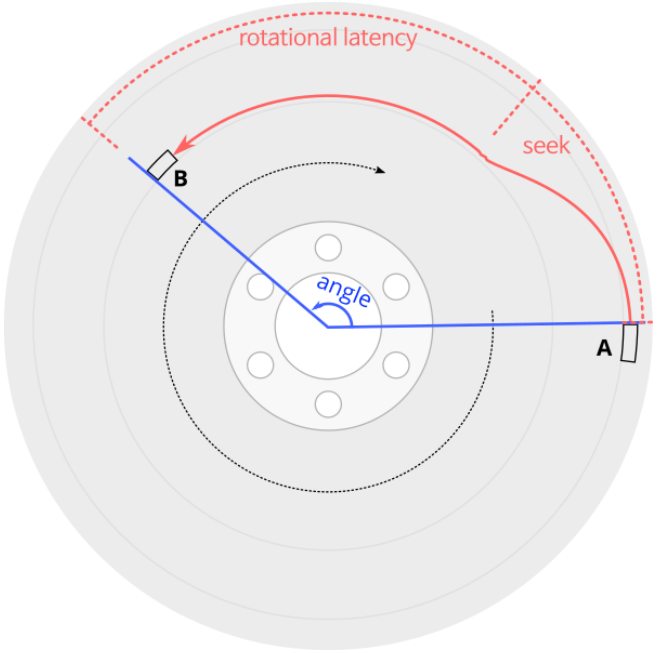

Figure 15: Access time

- As calculating access time is expensive operation, and in each track access time is linear function of sector address, we can calculate the seek profile. Seek profile is the smallest access time to sector for each track. We can calculate seek profile and then using that information we can calculate sector access time by using the linear function. The points outside the line are hardware bad sectors, they are in range as they are reallocated to the other part of HDD. The offset we see in our graph is track skew.

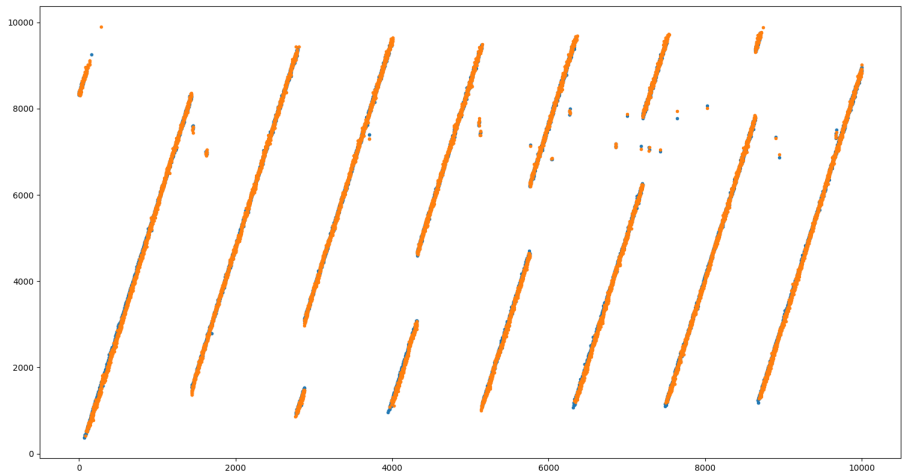

Figure 16: Access time by sector number

11 Useful Links

- https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system
- https://en.wikipedia.org/wiki/Volume_boot_record
- https://en.wikipedia.org/wiki/Master_boot_record
- https://en.wikipedia.org/wiki/GUID_Partition_Table
- https://www.win.tue.nl/~aeb/linux/fs/fat/fat-1.html
- https://man7.org/linux/man-pages/man2/open.2.html#NOTES
- https://superuser.com/questions/215341/dev-sda1-not-a-subset-of-dev-sda
- https://www.percona.com/blog/2018/02/08/fsync-performance-storage-devices/
- https://blog.stuffedcow.net/2019/09/hard-disk-geometry-microbenchmarking/