Index

Intro

The two main problems we are to solve are the speed and the limited map size. As for the speed, we just need to push it as far as possible. And as for the map size, we need to make it grow beyond the available RAM.

Here I will focus mainly on the speed. Yet while discussing the garbage collection I will also touch the memory problem, because GC interferes with both memory and speed, and the solution will work for both.

The main contributions to the processing time are (in random order):

- data IO
- taking SHA1 hash
- compression
- map operations (check & put)
- garbage collection (GC)

The first 4 are deterministic operations, meaning they are the same for each data block. The GC activity is non-deterministic, its profile significantly changes over the application lifecycle. So I'll consider deterministic operations and the GC separately.

The deterministic operations: compression, digesting, mapping and IO.

Purpose: determine the throughput of each operation alone. Compare the predicted combined throughput given by formula

$$T = 1 / (1/t1 + 1/t2 + 1/t3 + \dots)$$

to observed one. Determine the most promising targets for improvement. Assess the expected improvement.

I did the following:

1. using specially written tests directly measured the performance (in terms of throughput in MB/s) of few compressors ; the speeds were in the range 15 to 600 MB/s.
2. in a similar way directly measured the throughput of existing SHA-1 implementation, which was approx. 105 MB/s
3. same for IO throughput (in particular the FileWalkerStream.getBytes() method), which was 182 MB/s
4. prepared the test data (1298 MB worth of video files) and ran the scanner against this data, using the compressors mentioned above. The speed for 'None' compressor was approx 60 MB/s, for LZ4:0 - 56.4 MB/s
5. Ran a CPU profiler against None compressor, with CPU sampling enabled. This gave me the following figures
   - Digest computation – 65% CPU
   - IO – 22 %CPU
   - map – 7 %CPU
   - service (JVM startup,logging,etc) – 6%

   For unknown reason this result doesn't agree well with the above results. E.g. profiler data suggests that SHA1 has 60/0.65 = 92 MB/s, while it's actually 105, discrepancy 15%. I decided to rely on direct measurements.
6. from p.5 I estimated the mapping throughput as 60/0.07 = 857 MB/s, and therefore the IO+map throughput is 1/(1/182+1/857) = 150 MB/s

All figures are approximate and valid for my box (Wndows 8.1, Java 1.8, Celeron 2Gz, 4Gb).

Compressor	Compressor throughput, measured		SHA-1 throughput, measured		IO + mapping throughput, guessed		Predicted total throughput	Observed total throughput	Expected SHA-1 throughput		Expected total throughput		Expected total throughput, 2 threads	
	MB/s	%CPU	MB/s	%CPU	MB/s	%CPU	MB/s	MB/s	MB/s	+ %	MB/s	+ %	MB/s	+ %
None	+inf	0%	105	59%	150	41%	61.7	59.0	250	138%	93.8	59%	150	154%
LZ4:0	599.0	9%	105	54%	150 500*	37%	56.0	56.4	250	138%	81.0 130.0*	44% 130%	150 176	166% 212%
LZ4:1	31.6	66%	105	20%	150	14%	20.9	20.9	250	138%	23.6	13%	28.1	34%
LZ4:6	31.4	66%	105	20%	150	14%	20.8	20.0	250	138%	23.5	18%	28.0	40%
LZ4:12	30.6	67%	105	16%	150	11%	20.4	19.7	250	138%	23.0	17%	27.2	38%
GZIP:1	22.7	73%	105	15%	150	11%	16.6	15.6	250	138%	18.3	17%	20.8	33%
GZIP:6	21.9	74%	105	15%	150	11%	16.2	15.4	250	138%	17.8	15%	20.1	31%
GZIP:9	21.3	74%	105	15%	150	11%	15.8	15.3	250	138%	17.4	13%	19.6	28%
(*) – For a fast 500 MB/s SSD														

The deterministic operations: discussion.

The most CPU load is produced by compression and digesting, the less important is the IO.

The compressors cannot be improved any further, because they are already implemented in optimized native libraries. The only improvement I can see is to add an extra ZSTD compressor, which has today's best compression-speed profile.

The SHA1 digesting can be improved using faster library, e.g. OpenSSL. On my system it gives 170 MB/s in 32bit version and 260 MB/s in 64bit version. The expected throughput with this improvement is shown in yellow columns.

The IO speed can only be improved only in hardware part, by using the fast SSDs. The fastest ones perform at 500+ MB/s, while the HDD speeds are limited to 150-200 MB/s. On the other hand, during the IO delays the CPU isn't loaded, so if we move the computations to different thread, the throughput will be the

T_total=min(T_io, T_copmression_digesting_mapping),

even on a single-core processor. The corresponding predictions are shown in the last column. The best expected throughput with non-zero compression is limited by 176 MB/s.

Can something else be done? Yes, it's the parallel processing on multicore CPU. If we manage to evenly distribute the work between the cores the processing speed will increase times the number of cores. 4 cores would provide 176*4 = 704 MB/s, quite over 500 MB/s, which currently is the absolute disk speed limit.

The garbage collection

First, few facts about garbage collection in Java (the GC). The allocated objects in Java cannot be explicitly deleted as in C, they are removed automatically by the GC. The GC activity is controlled by a complex set of triggers. The main trigger is the lack of free memory in the heap space. The activity may be of varying strength, from shallow to full scan.

The easiest memory usage scenario, from the GC viewpoint, is when the program constantly creates a lot of big short-lived objects, which are abandoned just after creation. In this case the GC takes only the shallow scans. The worst profile for GC is when a program keeps a big graph of cached objects, which cause the full scans.

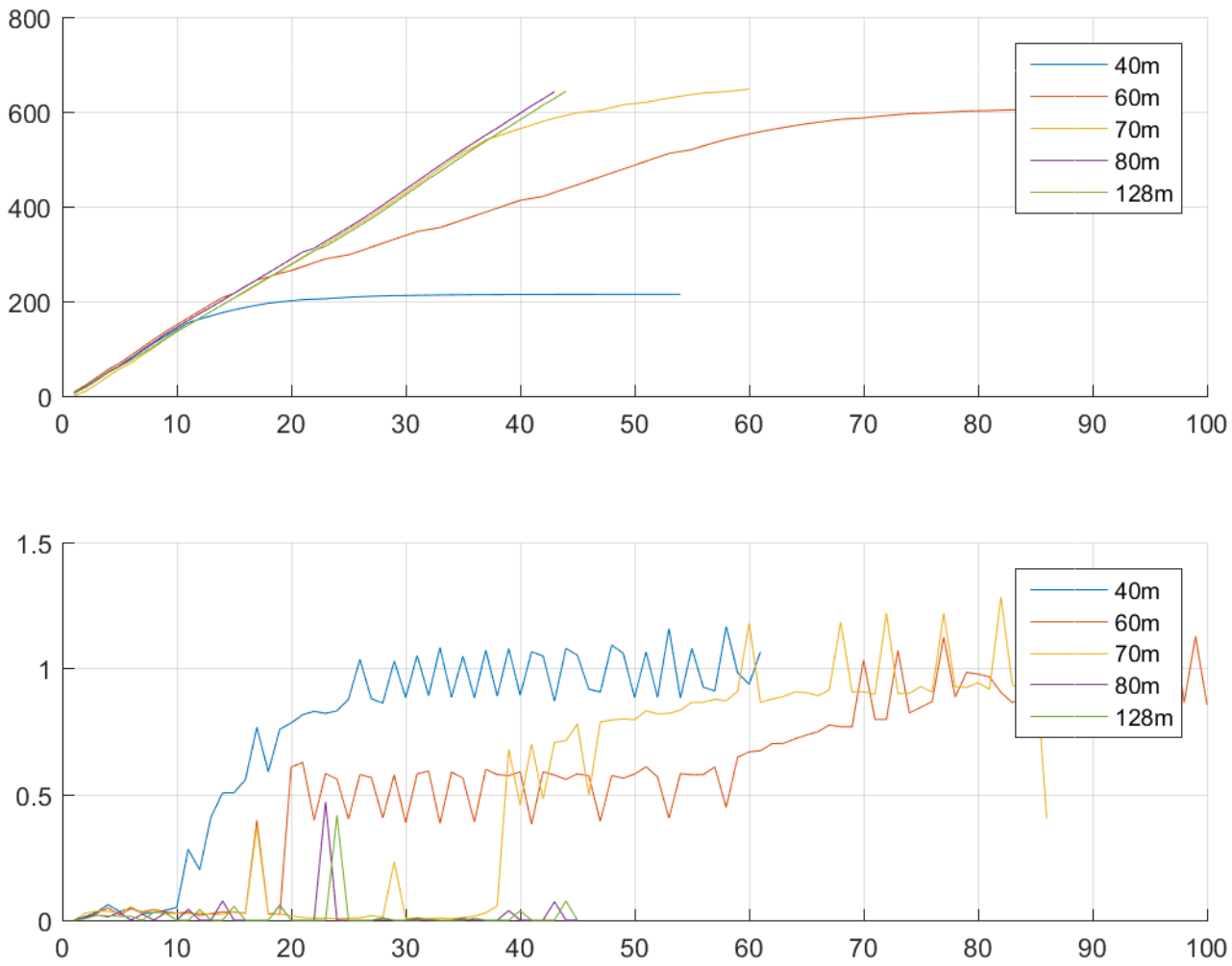
In the CompScan we have exactly the 2 types of objects described above:

- the short-lived objects are the IO buffers. New buffer is created for each block read and instantly discarded.
- the long-lived objects are the block hashes. They are stored till the program end, and their number constantly grows.

The experiment. The idea was to run the CompScan under various levels of GC activity and see its impact on performance. The desired level of GC activity can be forced by imposing a heap size limit on the app (using –Xmx option).

I ran CompScan few times against the same 1298 MB dataset with the block size 1000 bytes, so there were 1 298 000 block hashes. Considering that each hash takes 50-100 B, the required heap size was 65-130 MB, so I set the heap limits in this range, namely 40, 60, 70, 80, 128 MB.

During the each run I recorded the processed bytes and the share of CPU cycles taken by GC. The results are on the figure.



Processed data vs time (top), GC share in CPU cycles vs time (bottom). The 'NNm' are the heap limits.

GC: Discussion

- The avalanche of short-lived buffers quickly fill the heap space and causes the start of shallow scans in the first 1-2 seconds. At the first stage (0-10 sec for 40m, 0-16 sec for 60m, 0-37 sec for 70m, 0-end for 80m and 128m) we see only the shallow scans. The CPU activity owned by the GC is negligible.
- The total size of block hashes grows much slower than the IO buffers size, so at first the GC ignores them. When their total size reaches certain threshold, the GC starts the full scans (at 10s in 40m, 16s in 60m, 37s in 70m). This threshold isn't reached in 80m and 128m.
- The greater is the total size of block hashes, the more often happen the full scans, and the more time they take. At this point the program quickly chokes under the GC activity, the GC share exponentially approaching 1, while the processing speed approaching the zero. The 40m and 60m finally failed with OutOfMemoryError. The 70m slowed down and failed at the very edge of succeeding. The 80m and 128m succeeded.
- From the above observation we conclude that each hash takes approx 70m/1298000 = 54 bytes

How can we prevent the GC taking over the CPU? The only solution is to move the map from the Java-managed heap to the unmanaged native memory. There will remain only small number of long-living objects and therefore no full GC scans.

So we need some off-heap key-value storage. We need the one(s) which can operate both

- over the filesystem
- in the RAM only,

be it the same solution or not. The filesystem-based mode is important because it breaks the RAM-size limit for the number of mapped blocks. The RAM-only mode is important because it's much faster.

The best available free alternatives I could find:

- LMDB (https://en.wikipedia.org/wiki/Lightning_Memory-Mapped_Database) – embedded KV DB, native library
- LevelDB (http://leveldb.org/) – embedded KV DB, native library
- MapDB (https://github.com/jankotek/mapdb) – embedded KV DB, off-heap collections, Java library
- Chronicle Map (https://github.com/OpenHFT/Chronicle-Map) – embedded KV DB, off-heap map, Java library

Library	RAM-only mode	file mode	Speed, op/s	Easy to implement
LMDB	no	yes	600 000	yes
LevelDB	no	yes	300 000	yes
MapDB	yes	yes	?	yes
Chronicle Map	yes	yes	1 000 000+	yes
Custom	yes	yes	15 000 000+	no

First 4 libraries above are a bit over-engineered for our task, they are actually a databases with support for transactions, sorted index and other features, at the price of lower speed, while we need only a simple map, even without the key removal operation. So I added a variant with custom simple off-heap map which would have much better speed, but requires some time to implement (extra 4-5 days). The speed figures were collected from different sources on the internet and aren't reliable, except C.M., where it's official. The Chronicle Map looks the most promising in the speed department, so I plan to start with it. Moreover, even if LMDB or LevelDB shine in the file mode, we'll need the RAM-only solution anyway, so I'll try them in the last turn. The final choice will be based on the following criteria:

- speed, especially at large sizes (over 30 GB)
- support for both RAM and file-based modes

It should be noted that file-based map would be practically usable only if the backing file is on the SSD, because of the huge HDD latency.

The conclusions

- The first target for speedup is the SHA1 implementation. By switching to OpenSSL we get 44% performance gain (56 to 80 MB/s) on the HDD-based filesystem, and 130% (56 to 130 MB/s) on a fast SSD; the disk speed becomes the bottleneck.
- To push the performance any further we should implement parallel processing on multicore CPU. The expected throughput on the 4-core CPU and the fast SSD is over 500 MB/s (the processing alone is 700 MB/s, the disk speed is the bottleneck)
- The severe slowdowns at high memory usage are caused by the garbage collector (GC). This is because the CompScan's algorithm contradicts the very premises of java garbage collection.
- We can completely eliminate the slowdowns caused by the GC. To this end we should use some off-heap map implementation. There are some appropriate libraries, the final choice requires experimenting. There may be some speed impact.
- We can make it possible for the map size grow past the RAM size. This is achieved by the same method as in previous paragraph. Only this time the map implementation should operate over file-mapped memory. Of course, this solution is practical only over SSD disks, because of high HDD latency.