Account on performance of CompScan.

The purpose is to evaluate the processing speed and capacity, give recommendation on optimal parameters and suggest further enhancements. I focused here only on the mapping algorithm, because it's the most limiting part. The speed of compression was researched earlier (in short, the speed ranges from 10 to 500 MB/s/core depending on the method, the fastest one is LZ4/level0, the SHA1 speed is about 300 MB/s/core).

The experiments were of 2 kinds. In the first set I scanned real files in the T:\RANDOM directory (about 900 MB total size). In the second set I generated the hashes on the fly and inserted them into the map. Why I did so is explained below. As a final touch I ran the full scan of T:\RANDOM.

This document includes screenshots of the VisualVM profiling tool and the charts built in Matlab from parsed output of CompScan.

Short overview of the results.
1. max processing speed is about 350 MB/s
2. Usual speed profile: processing starts at max speed and keeps it for few seconds or minutes (further denoted as phase 1), then starts to gradually slow down to 50 MB/s (phase 2), then sharply drops to below 1 MB/s (phase 3), eventually approaching zero.
3. Speed limiting factors ("processing" here means digesting+mapping):
 - phase 1: disk IO
 - phase 2: both disk IO and processing, alternately
 - phase 3: processing
4. The onset of the phase 3 (i.e. the sharp speed drop) is strongly connected to the moment when the map size reaches the available system RAM and is caused by the page swapping activity.
5. The amount of data processed before phase 3 (denoted further as processing capacity) is determined by system RAM, average map entry size and block size:

$$capacity = block\ size * \frac{RAM}{entry\ size}$$

6. We can control, to some extent, the average entry size and thus the processing capacity using the parameter –mapListSize, at the expense of decreased peak and average speed.
7. Full scan of T:/RANDOM with block size 4k with appropriate value of mapListSize took 7200 seconds.
8. There is a possibility of further significant improvements in speed and capacity.

Scanning T:/RANDOM

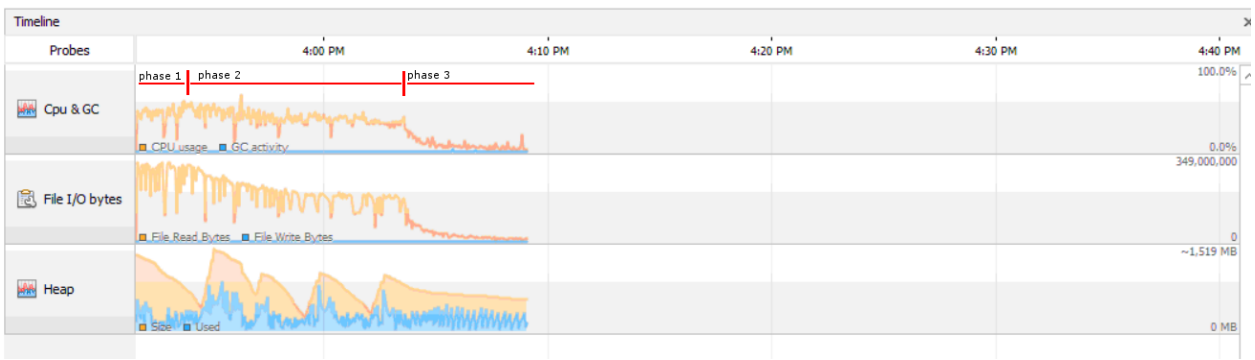
All experiments use relatively small block size 1000 bytes to emphasize the problems.

1. Simplest case with default parameters, block size=1000 bytes:

java –jar CompScan.jar t:/random out/stream 1000 10000 None

Below is the screenshot of VisualVM's Tracer tool.

The 1st graph show CPU util vs time, scale 0-100%
The 2nd graph shows data throughput, or processing speed, B/sec
The 3d one shows currently allocated heap



In the 1st phase we see slight growth of CPU utilization at constant overall speed. This happens because the cost of insertions to the map is initially low but growing, and the overall speed is bounded by disk speed.

In the 2nd phase we see overall processing slowdown. This is due to both the map slowdown with size and the disk slowdown (as demonstrated below).

In the 3-d phase the program stalls after processing 180 MB data in 13 minutes. The interesting thing is how the java profiler fails in explaining the cause of the stall. There is neither visible IO, nor processor activity, nothing. It took quite a time for me and a lot of profiling experiments to realize what happens. The clue was the size of allocated memory-mapped files, which was equal to system RAM size (24 GB).

Another thing to note is the almost-zero GC activity and the non-growing heap. There are lots of temporary arrays but they are easily collected.

2. Second run, with different mapListSize

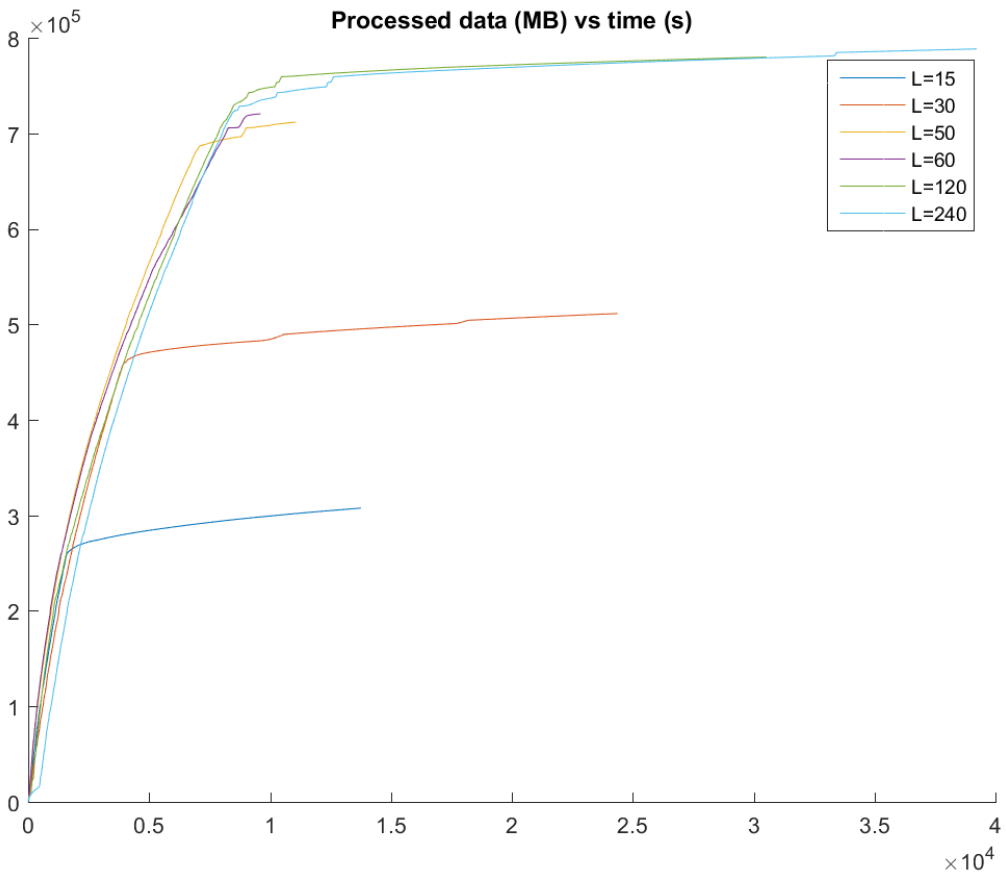
The mapListSize parameters controls the tradeoff between speed and average entry size. I checked if it would have any effect on processing capacity. Its value in the previous experiment was 9 (the default), and this time I tried 25

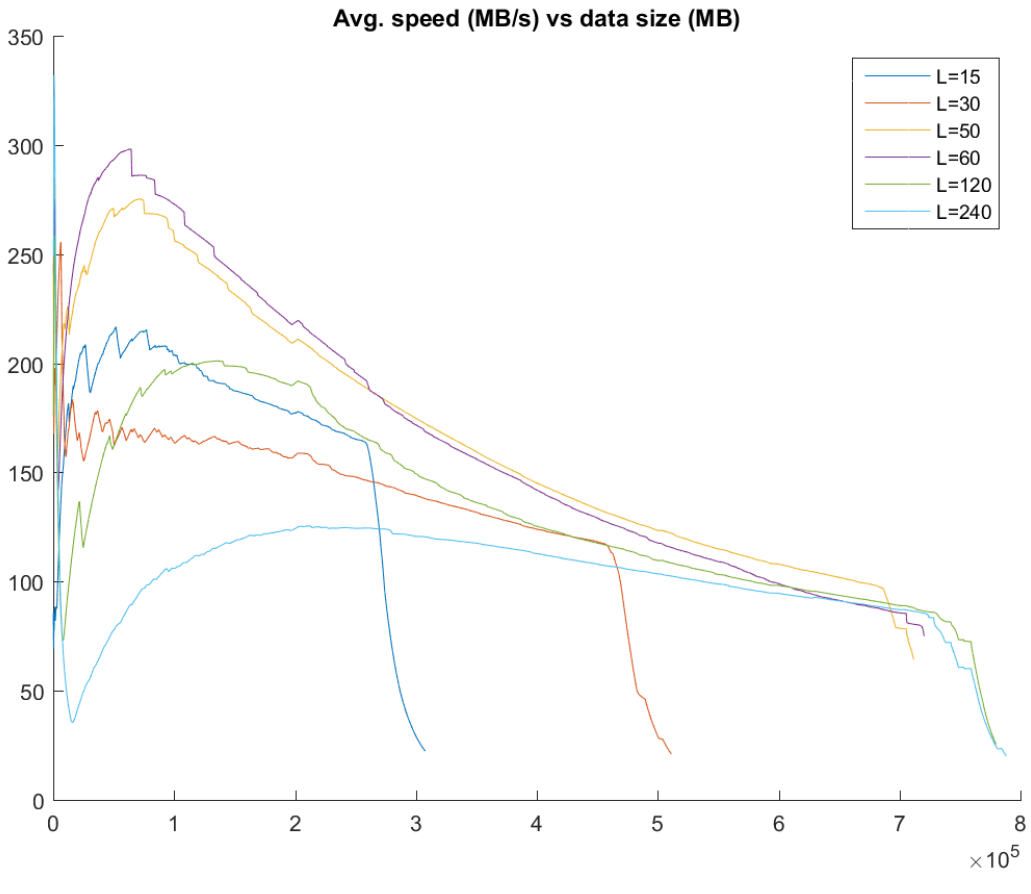
java –jar CompScan.jar –mapListSize 25 t:/random out/stream 1000 10000 None

This time the phase 2 lasted much longer, and the scan stalled after processing 380 MB in 1 hour (the phase 3 didn't fit into the picture). So the mapListSize definitely influences the capacity.

3. Finding optimal mapListSize

Several scans of T:/RANDOM were carried with mapListSize (denoted in the pictures as L) ranging from 15 to 240.



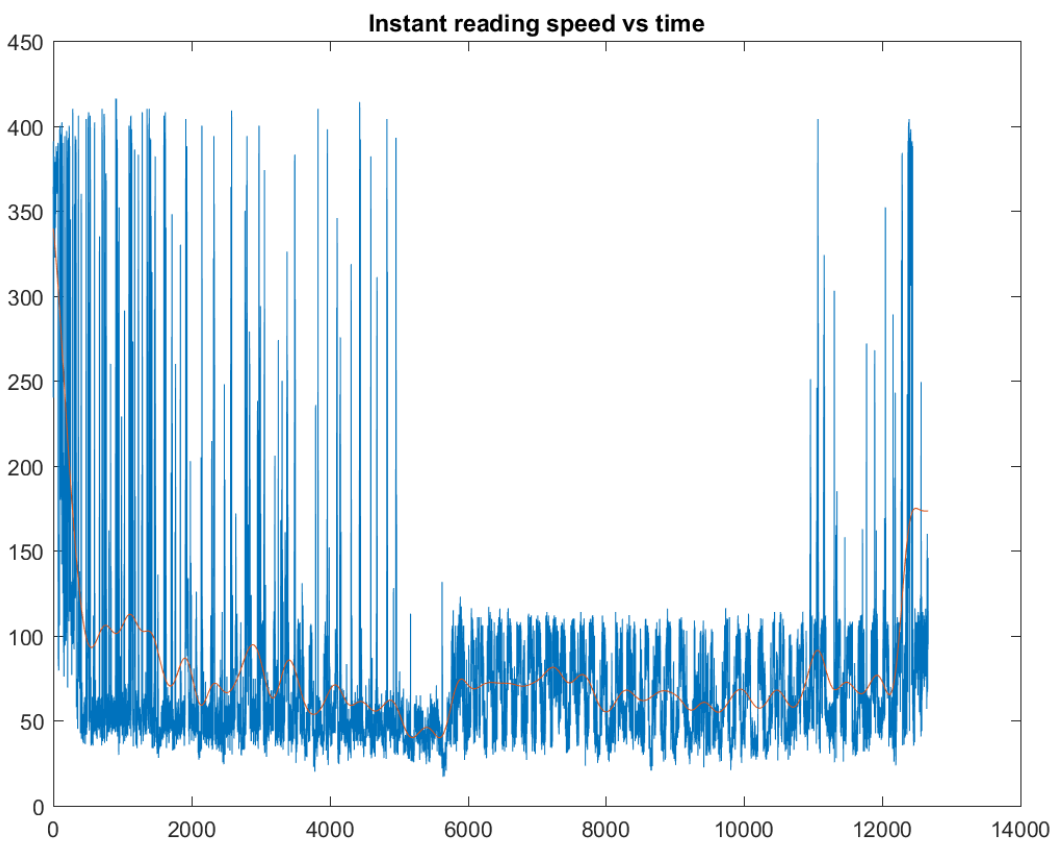


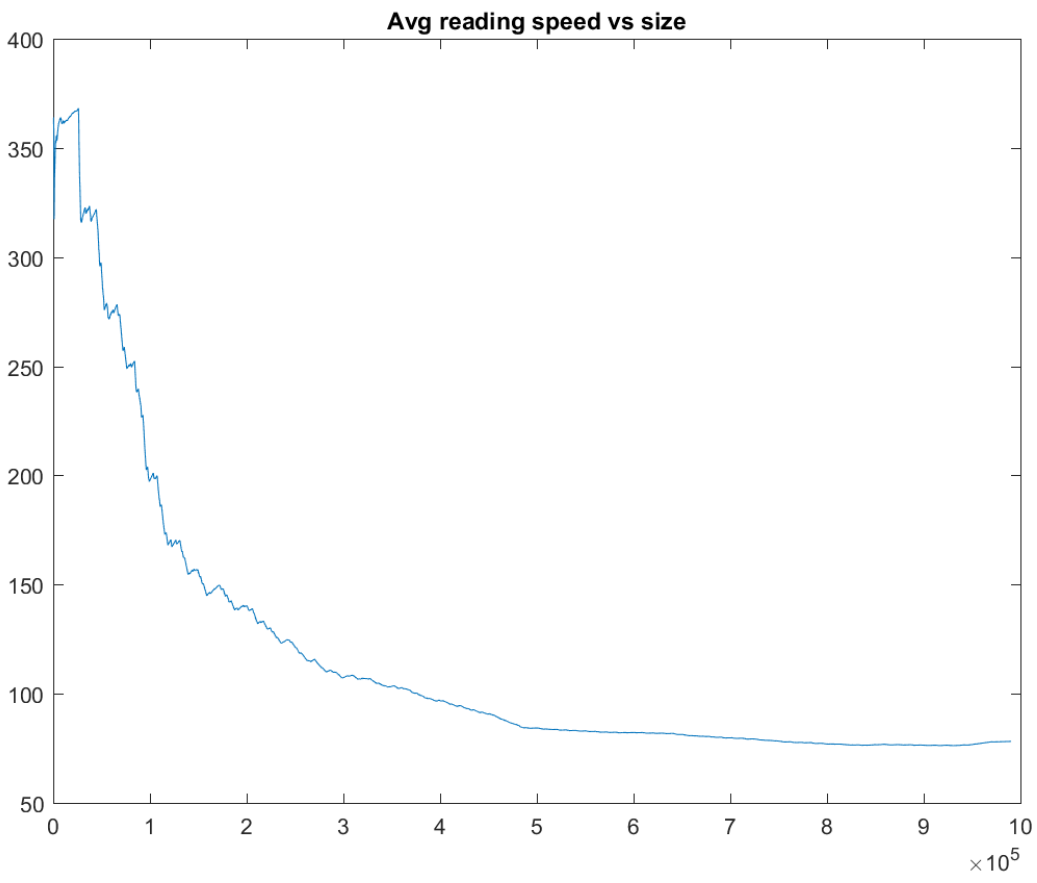
The second picture shows average (not instant!) processing speed vs processed data size. Processing capacity monotonically grows with L, as expected.

Processing speed, both peak and average, on the other hand, is not monotonic with L, with avg speed topping at L=50, and peak speed topping at L=60, which is in contradiction with the expectations (I expected the speed to drop monotonically with L). This may be caused by a more linear layout of the map entries at larger L. Anyway, I used this result in selection of the default value of mapListSize, changing the latter from 9 to 50.

Disk speed profile.

Let's look at how disk speed depends on the size of read data. I made a program (net.deepstorage.compscan.util.StreamTest) that reads data from T: using the same code as in main CompScan procedure and prints each second the size and the rate.





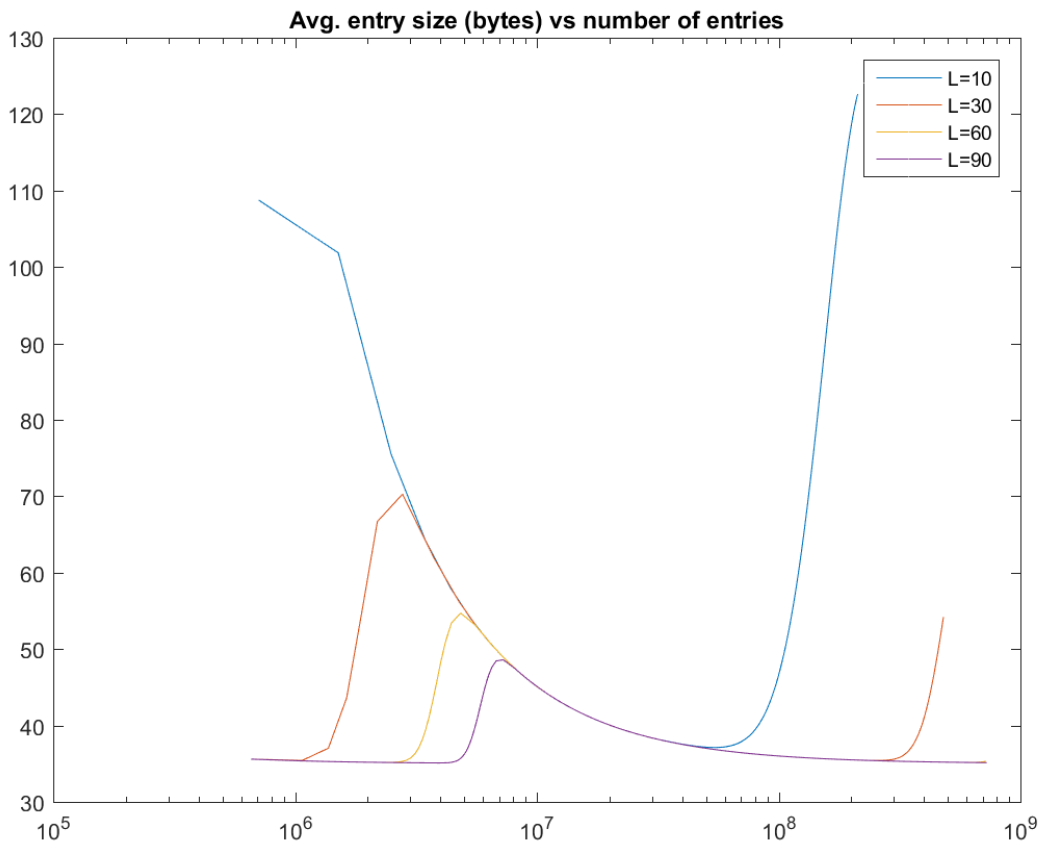
The interesting thing is that the reading speed is lower than the speed of the real scan. That is, the reading+processing is faster than reading alone. I have no explanation for that. I repeated this experiment twice with the similar result.

Experiments with hashes generated on the fly.

Here I tried to factor out the data reading speed from the picture. The program (net.deepstorage.compscan.util.map.Sha1MapTest) makes sha1 hashes from consecutive integers (more exactly, from 20-byte strings containing such integers) and stores them into the map, with single thread. It records the time, number of entries in the map and the size of the map in bytes. It was run with different mapListSize values.

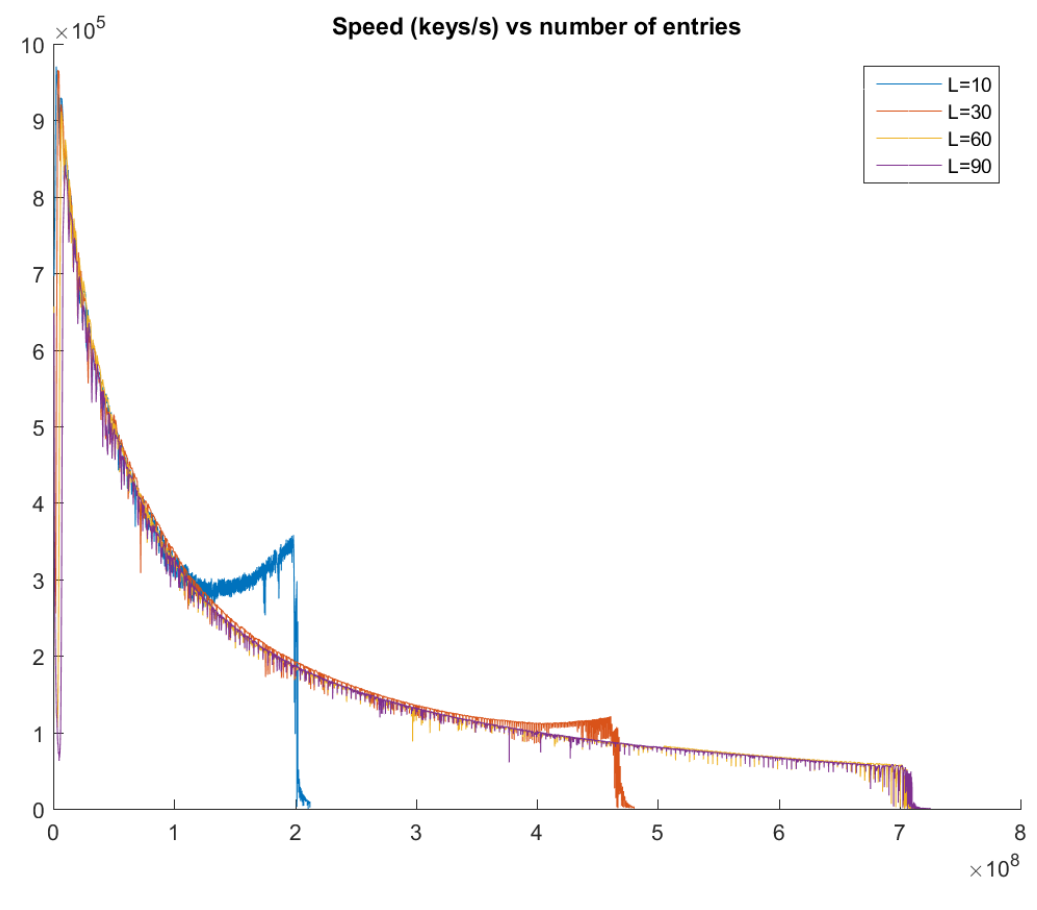
To compare these experiments with the previous ones just note that 1 entry corresponds to 1 block or 1k of processed data. The same applies to the speed: 100k entries/sec correspond to 100 MB/s.

Avg. entry size vs number of entries (note logarithmic X scale):



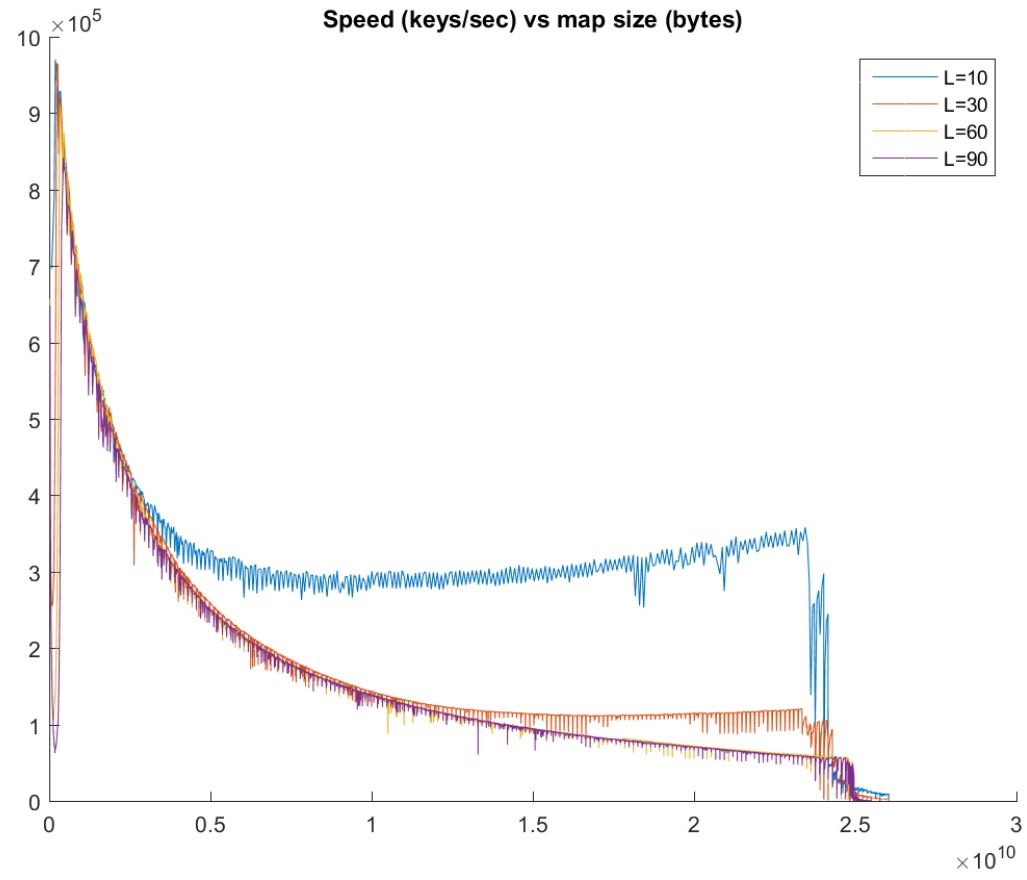
What is missing in this picture is the periodic nature of average entry size. If there were much more RAM, we would see it.

Instant insertion speed vs number of entries:



Smaller values of L perform faster but provide less processing capacity, as expected. Values of L over 60 don’t add capacity.

Instant insertion speed vs map data size (bytes):

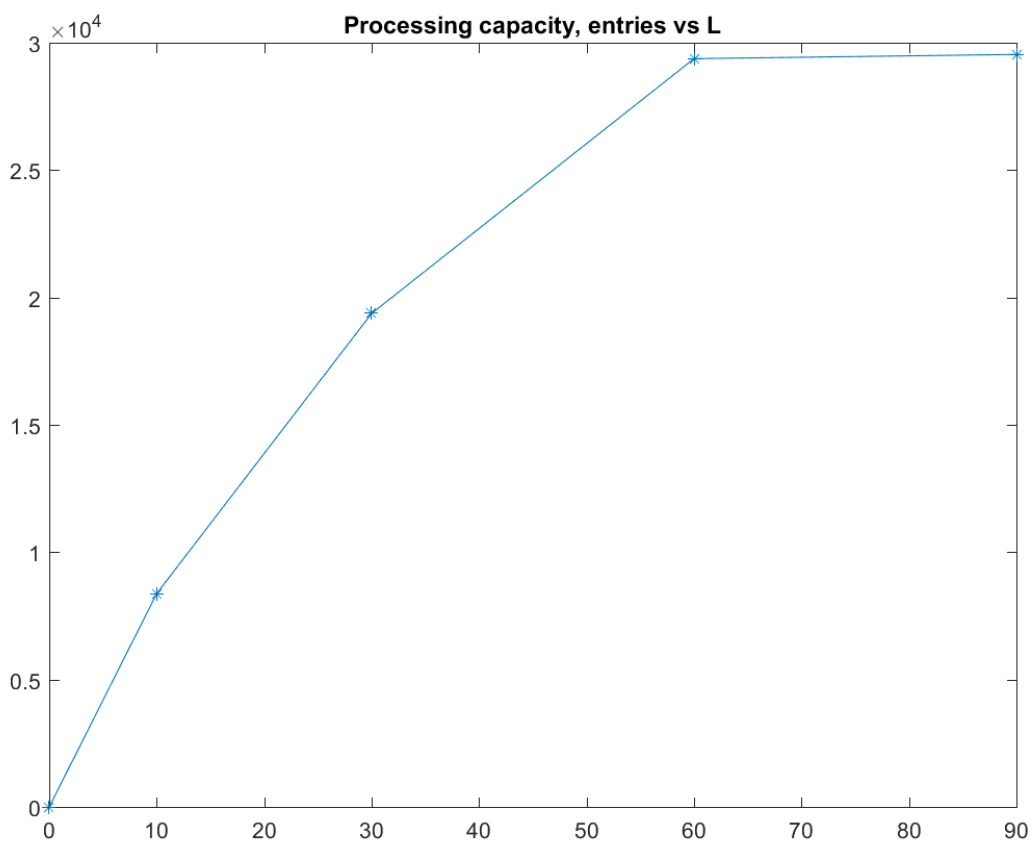


Here we clearly see that processing capacity is strictly bounded by system RAM (which is 24 GB).

This set of experiments shows monotonically decreasing dependence of top and average speed on L, exactly as expected. But this is in contradiction with real file scanning experiments, where best speeds were achieved for L=50 or 60. I have no explanation for this.

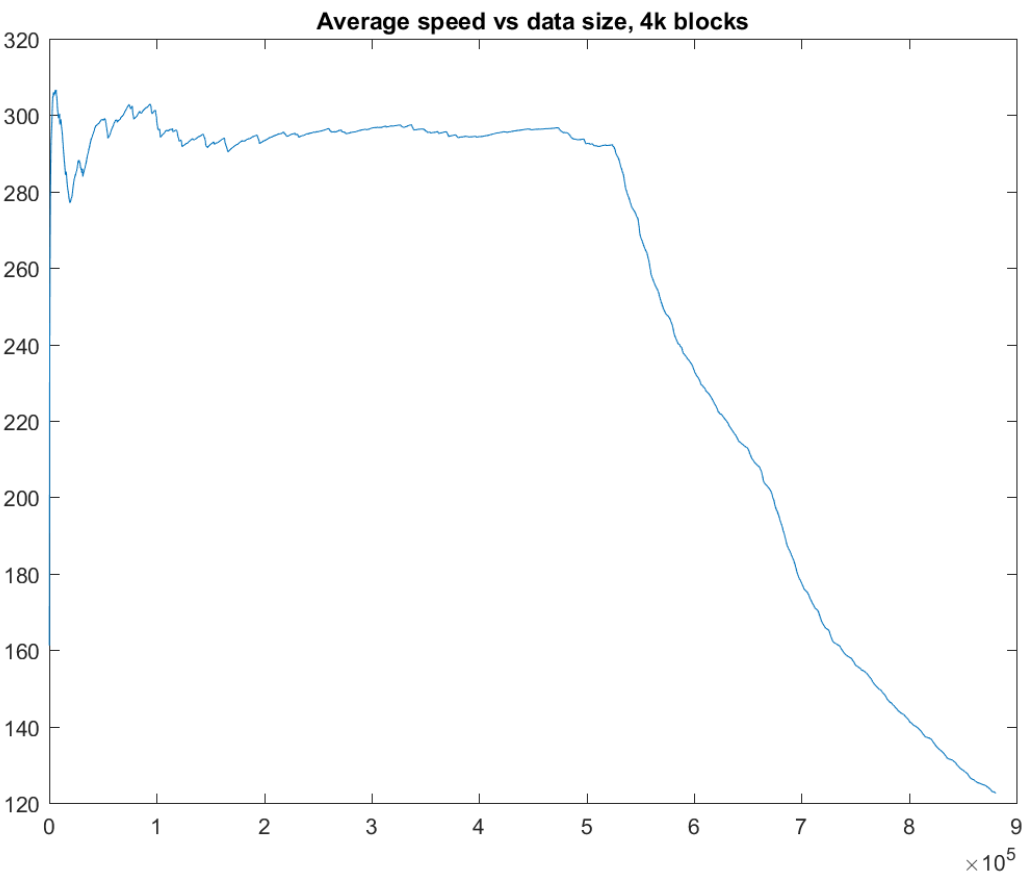
Unfortunately, using sha1 of 20-byte strings was a methodical mistake in this set of experiments. I had to not use the sha1 at all (which would give me pure map performance) or use sha1 with 1000 byte strings, which would make the results directly comparable with previous set of experiments.
I estimated the pure insertion speed afterwards using another PC with less memory. The initial speed is about 3.5M entries/s, which would finally drop to ~400K for L10, and to ~100K for L60.

And finally here is the processing capacity graph, entries per 1 GB RAM as function of L:



Complete scan of T:/RANDOM with 4k blocks

As a final touch here is a full scan of T:/RANDOM with 4k blocks and optimal L:
java –jar CompScan.jar –mapListSize 50 t:/random out 4k 10k None



The scan took 2 hours (7188 sec) with average speed 122 MB/s. First 500 GB were scanned with average speed 290 MB/s.

Discussion

The main factors that determine the map performance are huge memory consumption and the intensive and highly random IO on this memory. Let's consider two cases:

- **Allocated memory fits into system RAM**
  In this case we deal with the speed of random IO over large arrays of DRAM. It turn out that this speed is surprisingly low. To make it clear, I made a small experiment on reding+writing 1 byte over 1 GB array at random places. The result was 10M IOPS, which explains the average speed of the map (< 1M op/s).
  Another factor is relatively large value of mapListSize which we have to use in order to keep better capacity. More on this is discussed in the "Further improvements".
- **Allocated memory exceeds system RAM**
  As the size of map data exceeds free system RAM, some entries get offloaded to disk and the growing numbers of map operations result in disk IO, and the total speed quickly drops by few orders of magnitude.
  The random disk IO speed is somewhere in the range 100-300 IOPS. So finally the speed would approach some 10-30 entries/s (and the disk would be quickly losing its life span). So by any means we cannot rely on HDD as storage for the map data. The possible workarounds are discussed below.

Further improvements

As the compression and the hashing have already been implemented using the fastest available libraries, I will focus only on improving performance of the map. Here is what can be done to improve its speed and capacity.

Improving speed

1. **Algorithm: improve the internal structure of the map.**
   The map's data consists of tables and linked lists of entries (those controlled by --mapListSize), making up a trie-like structure. The tables are rather bulky and are the main source of overhead. Their current size (256 entries, 256*7 bytes) is chosen for simplicity of coding. I can try the smaller size (16 or 32 entries), which will certainly result in small average entry size even at small values of –mapListSize. Currently we have to trade between speed >300k entries/sec at entry size 120 bytes or speed <100k entries/sec at entry size 40 bytes (smaller entry size means better capacity). The purpose is to eliminate this tradeoff completely and have max speed and max capacity at the same time. The complexity of this work is low (<1 week)
2. **Algorithm: Less layers in the memory access code.**
   The map code works with façade that represents memory as continuous space. Under this façade there is memory management code that allocates ByteBuffers (either direct or file mapped) of fixed size and keeps them in ArrayList. Each read/write operation involves computing buffer index and offset, retrieving the appropriate buffer and reading/writing it at the offset. This is the most frequent operation, and speeding it up would have a significant impact.
   Ideally we would
   - replace ByteBuffers with natively allocated buffer addresses, accessed through sun.misc.Unsafe
   - replace a pool of multiple buffers by a single continuous buffer, dynamically expanded when necessary. The trick is to do it using some virtual memory magic, without copying and excessive memory allocation.
   I made a quick test on the effect of replacing the pool of ByteBuffers with single fixed ByteBuffer, the effect was 6-7% speedup.
   As for the dynamically expanding continuous buffer, I'm not yet quite sure if it's possible. But quick research into such fuctions as realloc/mremap on Linux and CreateFilemapping(MEM_RESERVE) + VirtualAlloc(MEM_COMMIT) on Windows shows it's at least worth considering.
   Expected speedup (with both parts implemented) is 10-20%. Work complexity small to medium (1-3 weeks), mostly due to research and extensive testing.
3. **Algorithm: make the lists in the map sorted (they aren't now).**
   This will result in list scans being twice shorter. Expected speed gain is 10-50%. Work complexity is low, < 1 week.
4. **Employ multiple memory channels.**
   CompScan app has 1 map instance with all operations on it exclusively locked, so only one thread works with the map at time. Therefore, it uses no more than 1 RAM channel at a time. And, as stated in previous sections, RAM throughput is the main speed limiter. So by allowing more than one thread to work with the map at time we would employ more RAM channels (there are usually 2 or 4 ones) and significantly increase the speed. But the map by its nature is not thread safe. The solution is partitioning the map into independently synchronized shards, just like it's done in java's ConcurrentHashMap. If the number of shards is 2x-3x times the number of channels, all chances are that each thread will work with its own shard.
   I carried out a quick test on a system with 2-channel motherboard. The 2 threads worked with 2 different maps simultaneously, and the combined speed jumped to 170% of normal. The more threads didn't add up anything, so this definitely was the effect of 2 channels. So the idea is proved and should be implemented.
   The expected speed gain is 70% for 2-channel boards and 190% for 4-channel ones. Work complexity is low, < 1 week.
5. Just a reminder, fiddle with JIT settings (-XX:MaxInlineSize, -XX:InlineSmallCode). Makes sense after the above changes.

The above changes combined, supposedly, may push the average insertion speed to the 500K-1M entries/sec range without sacrificing capacity. Currently we have 300K at low capacity or <100K with max capacity.

Improving the capacity

As stated above, the processing capacity of the system is limited primarily by the installed DRAM, which is expensive even in mild quantities (>$12/GB). And if one needs more than mainstream configurations allow it becomes insanely expensive. So what can be done to increase the capacity except throwing in more DRAM?

- **More tight map entry packing**
  This is closely related to p.1 in the previous section, see discussion there. There is very limited room for such improvement.
- **Optimize the map for SSD storage**
  SSD memory is about 10+ times cheaper than DRAM, if we could use it the problem is solved.

Its speed is somewhere between the HDD and DRAM, and it has certain peculiarities (we speak about random access, of course). The speed of SSD greatly depends on the "queue depth", that is, a number of simultaneous independent IO requests. At the lowest QDs (like 1) the typical speed values are in the range 1-10k IOPS. At QD=32 the speeds for mainstream SSDs are in the range 100-200k IOPS. Hi-end SSDs with NVMe interface promise 440k/360k r/w IOPS (http://www.pcgamer.com/best-nvme-ssds/). The 400k IOPS roughly correspond to average speed of 40k entries/s. With 4k blocks this would be 160 MB/s processing speed, which would be quite acceptable.

How can we provide high QD? The answer is the same as to p.4 of the previous section: multiple threads and the independently synchronized shards inside the map. The only difference is that the number of threads and shards should be much higher, at least 32+ threads and 2-3 times more shards.

Another approach, complementary to previous, is to use multiple SSD disks and distribute the map data between them.

The work complexity is low to medium, 1-2 weeks, mostly because of research and testing.

- **Cluster of low-cost nodes**

  With some sort of distributed system we can scale up both speed and capacity. There are 2 approaches:

  - use some existing distributed KV database: Redis, Cassandra, MongoDB, etc. The candidate should be capable of batch inserts/checks to avoid network overhead. The reported speeds for such system lay in the 10k – 500k inserts/sec range, which sounds promising.

    Work complexity is medium, 1-1.5 months, most of it is testing candidates.
  - build own clustered system using the existing map as a core engine. This makes sense, because with suggested above improvements the map will be superior to existing db engines by a huge margin, due to its high specialization. The same will apply to the network level.

    Work complexity is medium to high, 2-4 months, and greatly depends on required features, like fault tolerance.

Improving the code quality

Though this is not about performance, I'd point at a lot of bad-quality code in the project. There are a few decent parts, but the main body of code is bad, including the argument parsing, main loops and statistics. Both dataflow and control flow are dispersed between classes in complicated and confused ways, lot of value duplications (when the same value is represented by repeated field in several classes), wrong use of OOP. This makes the code extremely hard to modify and extend, I'm speaking from my own experience in the first place. If any further development is planned, I'd suggest to rewrite this parts from scratch.