Creating a virtual cache for LustreFS: a comparison of key-value in-memory object storage systems

Jorge Marcos Chávez - ARCOS UC3M

September 2018

Contents

1 Initial research **2**
1.1 Candidates 2

2 Test program structure **3**

3 Testing **4**
3.1 Test design 4
3.2 Results 4
3.2.1 Stage 1 4
3.2.2 Stage 2 5
3.2.3 Stage 3 7
3.3 Conclusion 7

1 Initial research

To design a cache system for a distributed filesystem, we will need several requirements fulfilled:

- Be an in-memory storage, since the disk is already used by the filesystem and we are looking to improve loading and writing times.
- Be persistent, save a copy of the data to the disk when needed and load it again on startup.
- Be able to manage big files, since we are going to work with several Terabytes of data at the same time.
- Be written in C++ or C and have a C++/C API.

1.1 Candidates

- Redis: A client-server, in-memory data structure storage system. The C++ library used, *cpp_redis*, has multithreaded support so no *mutex* is implemented in any of the tests. The server has to be executed separately (with *redis-server*), so we can run *redis-cli* (the Redis command line tool) in monitor mode to see how the data is stored and read. It has the ability to save the DB content to disk. By default it does it automatically each 120 seconds if the data has been changed, but we can force the saving or change the times. It is automatically executed after a shutdown call, too, and it creates a local file with the dumped content that is loaded when you start again the server.
- Memcached: A memory object caching system. Given that it does not have the ability to save to the disk, persistence would have to be implemented by other means and it has been discarded.
- LevelDB: A *mutex* has to be used to access one instance of the C++ library concurrently, so we have to take into account its overhead. Although the data itself is stored to disk, the memory is used to have a cache of accessed blocks and a write buffer.
- Couchbase Community edition: A NoSQL distributed database. We only have the synchronous mode since the official library does not support asynchronous IO. In this case the Couchbase server is running with only one bucket of 6.5GB in our machine, and the test is executed in it too, accessing localhost. The Couchbase server is configured to accept a "document" (block) size of 20MB maximum.

- Elasticsearch: A distributed storage system for search engines. Although it allows the execution of several nodes, and so distributing the storage and access load, it only allows inputs and outputs via a REST API (http petitions), so it has been discarded.

2 Test program structure

Testing is done using a program in C++ which creates several threads that insert and get data from the DB:

1. The insertion and read functions are declared. Both are almost the same, with a *while* loop that executes a block of code as many times as specified in the arguments. Depending on the DB, there will be a *mutex* in action (to avoid the access to the DB from two different threads at the same time) and a line to commit the changes (in an asynchronous or synchronous mode).

2. In *main*, the program arguments are used to set the number of threads for each function, insertions per insertion thread and reads per read thread. So if the execution is `./main 4 1000 2000`, the test will be executed with 4 insertion threads, 4 read threads, 1000 insertions per thread and 2000 reads per thread.

3. The chosen file that will be inserted into the DB is opened.

4. The DB is opened or created, and the thread arrays and atomic operation counter are defined.

5. The timer starts running. From this line of code we measure the execution time.

6. First the insertion threads, then the read threads are created. When they are all started, they are joined started from threads number 0 for insertion and read.

7. The timer is stopped, so only the data operations are measured. Then the DB is closed and the statistics are printed.

The data inserted and requested for reading is not unique and can be requested, stored and overwritten several times in a row, and also requested even if the entry still does not exist. In order to maximise the performance of the test program to evaluate as objectively as possible the speed of the DB and the used libraries, the control variable of the *while* in the insertion and read functions is used as the key. The value that will be stored can be chosen from an assortment of files between 1KB and 4GB.

3 Testing

3.1 Test design

All the test programs take the same parameters: `./program [NUMBER OF THREADS] [TOTAL INSERTIONS] [TOTAL READS] [INSERTION FILE SIZE] [INSERTION FILE UNIT IN K OR M]`. An example execution could be `./sync 4 512 512 16 M`, which creates 4 insertion threads, each one with 128 pending insertions of 16MB files, and 4 read threads, each one with 128 pending reads.

The test is divided in three stages, and each one is designed to test certain characteristics of each storage system:

- Stage 1 Using many blocks with a small block size. A total of 4096 insertions and 4096 reads, with block size from 1KB to 1MB (in steps of powers of two). This stage is executed four times, each one with 1, 2, 3 or 4 threads. Tests the small block management speed and responsiveness.
- Stage 2 Using several blocks with a bigger block size. A total of 2GB will be inserted and read, using a block size from 1MB to 512MB. The number of insertions and reads are calculated to reach the the total size. This stage is executed four times, each one with 1, 2, 3 or 4 threads. Tests overall performance with a wide range of block sizes.
- Stage 3 A few blocks with sizes of 1GB, 2GB and 4GB. A total of 16GB will be inserted and read, and the number of insertions and reads is calculated on the fly too. This stage is executed with 2 threads. Tests for stability and reliability with large blocks.

3.2 Results

3.2.1 Stage 1

In this stage, block sizes are represented in kilobytes, from 1 to 1024. As we can see in the figure 1, Redis is faster with blocks smaller than 128KB by almost 1.5 times, and gradually slowing down until Couchbase gets the head position starting at 128KB. LevelDB gets a ridiculously slow speed compared to both.

The multithreaded version of the Redis test gets a significantly higher speed than the single thread one, and in contrast LevelDB gets a reduced speed when using a multithreaded library access. That may be influenced by the need of a *mutex* in the LevelDB library, and the thread-safe capability of Redis.

In the figure 2 we can confirm the previous results, since Couchbase spends more time executing the test with blocks smaller than 128KB than Redis. After that, they switch positions (Redis gets slower). LevelDB is one and a half orders of magnitude slower than Couchbase, and 10 times slower than Redis.

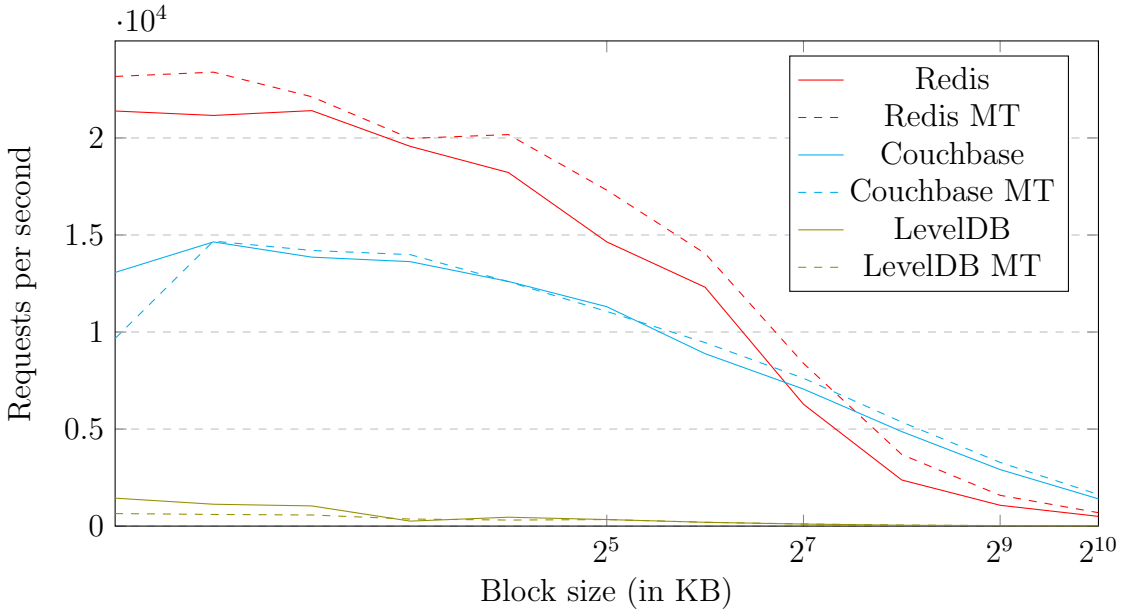

Figure 1: Requests per second against the block size

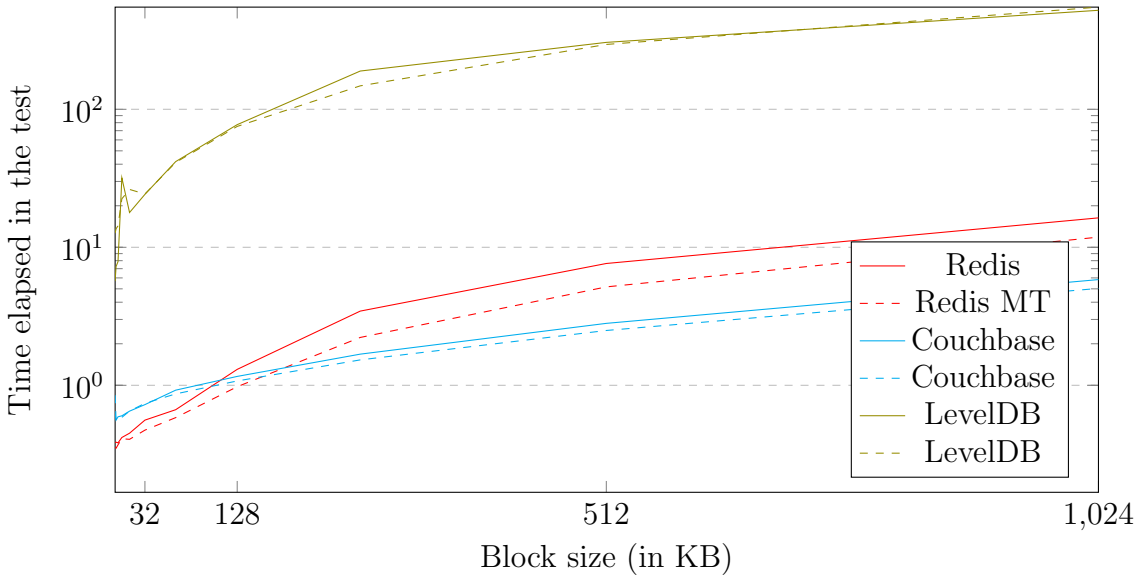

Figure 2: Time spent with the first stage test

3.2.2 Stage 2

Now we have a total of 2048MB inserted and read, and we vary the number of blocks and, inversely, the size, starting with 2048 blocks of 1MB and ending in 4 blocks of 512MB.

In this case the biggest dealbreaker is Couchbase not being able to use blocks bigger than 20MB. In our tests, then, the last successful result comes from the 16MB block size test. This is a server side limitation and is not possible to change without modifying the code.

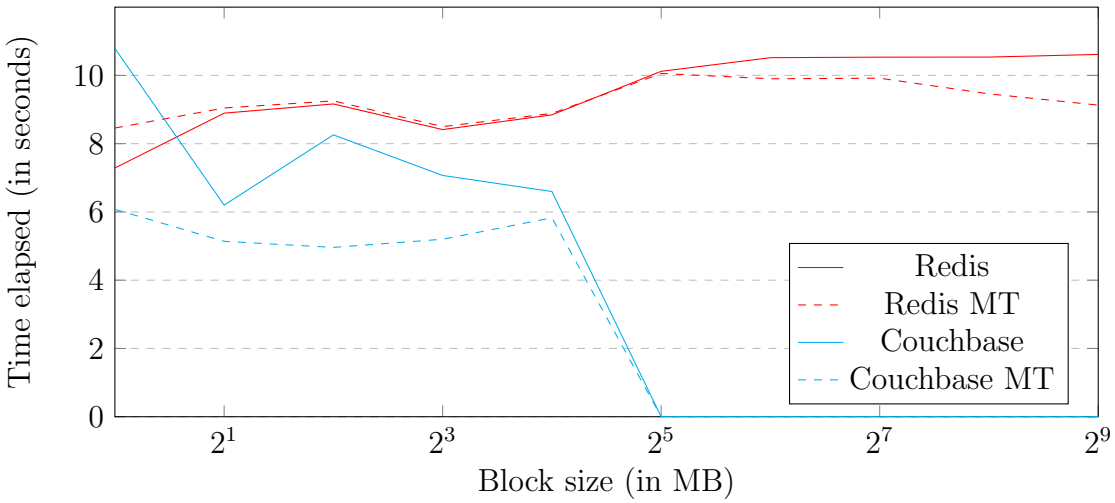

Figure 3: Time elapsed in seconds against block size for Redis and Couchbase

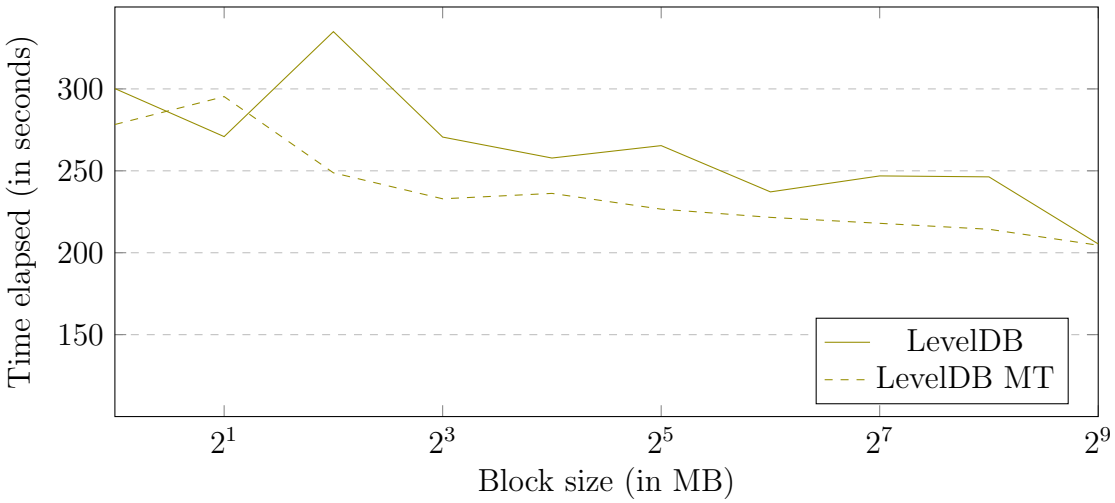

Figure 4: Time elapsed in seconds against block size for LevelDB

In the figure 3 we can see that actually Couchbase single threaded works best with blocks bigger than 1MB, while by using several threads we can get the best results of the test, almost reducing the overall time by 40%. Redis single threaded, in contrast, is slower than Couchbase, and works better with smaller blocks but from 32MB on the results show that the block size does not matter much. In fact,

when using several insertion and read threads, the time when using blocks bigger than 32MB gets reduced by around 15%.

LevelDB has its own figure (figure 4) because the time is several times slower than Redis and Couchbase and it does not fit in the previous graph. The times decrease with bigger blocks a 33% between 1MB and 1024MB sizes, but still the fastest time is almost 20 times slower than the slowest times of Redis and Couchbase.

3.2.3 Stage 3

The only system that survived this stage was LevelDB, with times of 1717.01 seconds (29 minutes) with 1024MB blocks, 1722.37 seconds (28 minutes) with 2048MB and 1007.0 seconds (16 minutes) with 4096MB.

Couchbase was unable to complete the stage from the beginning (with a block size limit of 20MB), and Redis gave disconnection errors when trying to complete the tests although that problem comes from an out-of-memory, which would not have happen if the test had to insert only 6 or 8 GB. (The testing computer has 18GB so taking into account that the file has to be loaded into the test program, and then into the database, the memory usage surpass the theoretic memory usage).

3.3 Conclusion

The only storage system which has been able to successfully handle most of the stages of the testing has been Redis. It is fast for all block sizes, fulfills the persistence requirements (can dump and load from disk when needed) and it stores everything to memory.