# Assumptions, trade-offs and decisions made

1. We are implementing Time Series Data Storage, log is divided into blocks, each block contains some amount of milliseconds (buckets)

2. Log is append only

3. Individual records can be deleted immediately OR just marked and later collected. I CAN NOT make informed decision, ‘cause mark is faster for DELETE, but will slow down READ. If you have intensive READ - then delete immediately is better. If READ ops are relatively rare, then mark is better.

4. TTL is considered as a way to mark records as deleted.

5. Deleted records should be removed by the background daemon process (sweeper) which can be triggered manually. I’ll write compaction logic, but not daemon. Tests will run vacuum() from time to time.

6. Sweeper will also remove dead blocks

7. Optimization to mark a block with max TTL value to speed up VACUUM will not be implemented. We need a separate “archive” process to calculate stats of the block which is no longer current. I’m not going to implement this, but this is a useful optimization. If somebody will implement it as a straight field update at the time of WRITEs to the current block - this will slow down WRITE and this is not the best way.

8. Same for any block-level counters on how many records added/deleted and is block empty. No enough info to make an informed decision. But this is a trade-off between slower WRITE to current block and slow VACUUM on non-current blocks. If you have a lot of data, long sessions and few WRITE ops, this can be useful.

9. No events ordering in the same millisecond to achieve a bit of performance. If events came in some order in the same millisec, this order will not be preserved on read. Reason: functionality doesn’t need it and WRITE will be faster just a bit. Nevertheless, if you want to preserve and order, well, use 3rd party concurrent linked hashmap.

10. FLUSH is not an atomic operation. If READ is reading data, there is a chance that some flushed records from active uncomplete FLUSH command will be excluded and some included. I can make it atomic by implementing a lock which will stop READ operations (hell, NO!!!) or multiversioning (but this is not a 4 hours even for design)

11. Ordering of DELETE commands is not preserved for active READ commands especially if they touch items from the same millisecond bucket. If you have an active long READ command and DELETE A then DELETE B in parallel, you may have a situation when B is present in results and A is not or vice versa. I don't see any reason to implement a fix for this, but if needed - multiversioning is the best way to achieve SERIALIZABLE transaction isolation level

12. READ will be slower on sparse data layout, so VACUUM is important.

13. Don't forget that we have compactification at the block level only. Block itself can be almost empty and this will slow down READ. The nightmare scenario is a bunch of blocks with a single long-living record in each of them. So, choose block size wise as well as TTL

14. Blocks are chained to speed-up search of the next non-empty block. Also we have a block index (block id : block reference) to find a first block of a READ range. Block Index is a simple ConcurrentHashMap, so the initial search is a bunch of map.get() to find the first existing block.

    I have a little optimization with minKnownBlock to avoid searching for blocks which are gone, so if READ asks for blocks 88-999999 most of which no longer exist, search will start looking from Math.max(minKnownBlock, requestStartBlock).

    I don't like this approach and I believe we should think about some more advanced tree-like index, which will be able to support floor() and ceil() to find the first available block after a given number.

15. VACUUM will not impact any READ operations even when it removes the blocks from index or chain. If long READ got the reference to the block which was deleted from index and chain, reference to the next block will be active and available till READ will be completed and only then GC will remove the block. So, "God bless GC architecture".

    Nevertheless, any attempts to clean-up or render invalid or corrupt in any way the block at any time will ruin everything. Even vacuumed block MUST be valid. Don't forget that the block which was vacuumed can be still referenced by long READ operations.

16. We have an index (and this is a potential weak spot) of items for DELETE and GET command (item : millisecond registered). It is a ConcurrentHashMap. This index imposes some size limitations (2B records and serious demand to RAM) which can be partially solved by offheap storage AND proper partitioning at the cluster level.

17. TTL filter out is guaranteed to be consistent. If READ took 10ms from 100 to 110ms of absolute time, records which had TTL 101, 102, 103, 110 will be ALL excluded or ALL included based on timings.

18. There is a potential caveat which may render the system inoperable - significant process scheduler fault. Let's imagine that we have a WRITE operation which is delayed in the middle of operation for some reason. Now we have a GHOST record, which was half-registered but never fully written to an index/block.

    If DELETE will come - we are doomed. The only way to fix it is to use a queue for WRITE/DELETE ops + dead-letter-queue for unsuccessful DELETE ops.

    But anyway, we have a situation when a block with a GHOST record is VACUUMed.

    Potentially, we may try to register a record in a block first, but this will NOT resolve the situation completely, the weak point is the place when we chose a block to write and then hanged.

    The best way to avoid such a situation is to implement a locking mechanism which will not impact WRITE/DELETE but will stop VACUUM until all the operations complete. Read/Write lock is good. WRITE operation MUST poll a target block number until lock acquired. This is 100% reliable solution.

    Nevertheless, one more mechanism MUST be added 'cause it may greatly reduce the problem - delay before block can be VACUUMed and this delta must be at least 500ms + blocksize in ms. My recommendation is to keep up to 1-10 minutes of a blocks. In this case we will let GHOSTs to complete in a reasonable time. This mechanism WILL NOT prevent data loss for GHOSTs, but will reduce the risk significantly.

    For the reliable solution we must use the proper locking mechanism described above I'm not going to implement.

    **WARNING: As I explained, there is a chance to lose the data if WRITE ops will be on hang until proper locking mechanism will be implemented to stop VACUUM of blocks which still have a dead WRITE on them which is NOT properly registered.**

19. We may have a TTL in the item index to clean-up dead records, nevertheless I would prefer to let VACUUM clean-up this index. Anyway, coherence between data and indexes is always somewhat a pain. This is a question to discuss and the choice will rely on the data I need.

20. Variable block size is not supported, but can be considered in the future.

21. Properties must be immutable, especially block size for now. So, I added comment but added some capability to refactor code in a future to dynamic properties and block size

22. The whole implementation is built around two principles

    1) only buckets are mutable and we have ConcurrentHashMap to handle concurrency

    2) Data structures and algos designed in a way that if change happened in another thread, this will not affect threads currently active. This reduced the need for synchronization dramatically.

    For example, if a block was vacuumed, the reader or writer, who already picked up it's reference, is still able to read/write it and jump to the next block. We have a GC, so I'm heavily relying on it, leaving deleted blocks integrated and consistent. GC is extremely important for my logic, 'cause, actually, I have a 2-stage vacuuming process and GC is used as a thread's references tracker and kinda trash bin.

    If this code will be implemented on C++, then GC-like logic should be introduced to free memory on this 2nd stage.

23. Algorithm designed in a way that FLUSH will not erase records which are added after the FLUSH command was issued with 1ms accuracy. Data which arrived the **same** millisecond as FLUSH was issued are not guaranteed to be protected, but code is written in a way that minimizes the chance to erase data added the same ms, but after the FLUSH command.

    Look, this is concurrency and we don't provide strong commands ordering. If we want really strong ordering, we will need to have the command queue.

    So, if FLUSH (1, 2000) was issued at the time 100ms and then record A was added at the time 101ms, FLUSH (1, 2000) will not affect A even if it will work from 100 to 300ms.

    But if FLUSH(1,2000) and then ADD will be issued the same millisecond, algorithm will try to minimize the chance that FLUSH will erase ADD.

    If you don't like it - remove commandTime restrictions.

24. Very similar approach is used for GET. Algorithm will try to return data no later than timm command time with 1ms precision.

    If you don't like it - remove commandTime restrictions.

25. There are no restrictions for GET on records count or execution time. I would propose to limit this or create and API with limitations in rows or milliseconds, something like
    ```
    get(long startTimeMillis, long endTimeMillis, int rowLimit)
    ```
    or
    ```
    get(long startTimeMillis, long endTimeMillis, int querySoftLimitMs)
    ```

26. GET may clean-up buckets from expired elements, this may slow down first GET a bit, but speed up following GETs. VACUUM, actually, do this also.

    If you don't like this approach, see LogBlock.get() and remove bucket update and index update.

    But if you'll ask me, I'd propose to
    1. Let GET trigger VACUUM in separate thread, based on statistics gathered
    2. Let GET do clean-up to some extent, like no more than 25% of rows or no more than 10ms

    Anyway, this is a long discussion. Let's have it!

27. ADD may return an exception if the process hangs under the load, this is intentional. It would be better to return an exception, rather than hang indefinitely.

    So, if ADD can not get a lock for block rotation for more than 100ms or ADD is unable to complete write in time to a newly rotated block (1000ms) then the method throws an exception.

    Such exceptions must be ultimately considered as a request to decrease speed and retry later (exponential delays must be used or DLQ). Also cluster scaler must start immediate dynamic scaling UP.

28. Some code, like Log.add() can look strange because of duplicated code. This is intentional to save some time in a hotplate. The idea is that the happy path is jet fast and if we need to rotate the block, then we will add more checks.

29. I expect that FLUSH is a rare operation, so FLUSH actually scratches the data out from buckets and index. So, FLUSH is slow, but GET is fast. If FLUSH is used frequently, we will need obsoleteBucketsLog which will be honored by GET and will be a task queue for VACUUM.
    ```
    this.obsoleteBucketsLog = ConcurrentHashMap.newKeySet(blockSize);
    ```
30. Static block size

31. Unique itemId. Can be fixed with advanced Items Index or PUT should delete old one

32. Get is not consistent on TTL, VACUUM can run in background and remove some TTL-outdated items

33. Vacuum is not parallel and not time bound