High Throughput NoSQL Databases for BigData

Janardhana Reddy Naredula[#1]

Walmart Labs, Bangalore, India

[1]naredula.jana@gmail.com

Abstract **:** This paper describes various problems and solutions in improving throughput of NoSQL Databases like Redis, Kafka, Cassandra, ElasticSearch,..etc.

Keywords: Transaction Per Second(TPC), AppendOnly Log, Mutation operations, Non-mutation operations, Instructions Per Cycle(IPC) [1], Inter Component Communication(ICC), Data Plane Development Kit(DPDK)

1. Introduction

NoSQL Databases like Redis, Kafka, Cassandra, ElasticSearch, RocksDb .. etc. are mainly used for storing and retrieving Key-value store. It contains two type of operations namely Non-mutation and Mutation operations. Non-mutation operations includes retrieving, searching data. Mutation operations includes writing, updating data to mostly immutable files. The file operation on immutable files can be synchronous or asynchronous. Non-mutation operation does not involve disk io if the data is cached, so the TPS of non-mutation operations is relatively higher when compared to mutation operation. All operations involve the following four computations:

1. **Network IO path**: Receiving/sending of message from/to network interface, and then decoding/encoding of network message.
2. **Inter Component Communication(ICC)**: Inter process or Inter thread communication between various components inside the server, or across the server and within the machine.
3. **Disk Write Path**: Appending the record to append log or commit log located on the disk, if it is a mutation operation.
4. **Disk Read Path**: Retrieving the data from append log or some other file if it is non-mutation operations.

To improve throughput of NoSQL database all the above four computations should be improved. The problems and solutions in the above computations on the linux OS are discussed in the next few sections.

Design Patterns used

In the NoSQL Databases like Cassandra, ElasticSearch, Kafka, ..etc part of BigData infrastructure, the following are common design patterns used, based on these patterns or assumptions rest of paper was constructed.

1) Most of the input/output was done using Immutable files only.
2) Four different type of file operations involved:
   a) Writing in to translog file synchronously.
   b) Reading the immutable synchronously
   c) Flushing data to files periodically.
   d) Reading from one of immutable file and creating new immutable periodically as part of housekeeping task.

   Here type-a, type-b are high priority when compared to rest of the two types since it impact the latency of user requests, type-b and

type-c can be done as part of housekeeping with less priority.

3) linux system call part of posix api's are used to communicate with OS for networking and file computations.
4) The machines used are large machines with multiple core like 72, and large amount of RAM like 256G, and good network and disk speeds.

2. Problem Statement

Design of NoSQL Databases on the linux system as limitations due to posix API for getting high throughput. The following are the bottlenecks from the point of linux system:

1. **Network IO path:** Syscall Interface with OS to receive/send network messages need high cpu cycles. In the current linux system NoSQL databases uses epoll, read, write system calls to receive/send the packets. Lot of cpu cycles are consumed in executing these system calls for each message, due to this TPS will be lowered.
2. **Disk Write Path**: Syscall Interface with OS to sync or flush the append log changes during mutation operation is a synchronous operation, and application need syscall to communicate with OS. The syscall can be write, mmap, fsync, msync,..etc.
3. **Disk Read Path:** This is a synchronous operation, and disk reads are lot slower when compare to memory read, due to this user level threads will get blocked. user level threads will be blocked if there is not sufficient memory, due to this application calls fadvise calls to evict the memory.
4. **Multi-core machine inefficiency:** Multi-threaded code does not effectively make use of multi core machine because of various reasons like synchronization between threads, syscalls, .. etc. Lack of efficient Inter Component or Inter Thread communication construct from OS makes multi threaded on multi-core machine not efficient.

Root Causes for the Problems:

Linux being a generic purpose OS, not designed for a specific purpose workloads like NoSQL,..etc. Due to this there is a room for improving efficiency.

1. Most of the operations needed system calls like fsync, fadvise, read, write, futex..etc. These calls slows down the speed of cpu, because it involve switch of cpu privilege from user space to kernel space and vice versa, due to this reason system call have less IPC[1] when compared to other computations. some syscall also need additional memory copy and other validations.
2. Some system calls like fsync are synchronous in nature, means user level thread need to wait for the kernel to complete. This as implications on latency. This is partially mitigated by extra user thread in the userspace without blocking other computations.
3. Communication between components or threads in a multi core machine slows down the cpu. Redis mitigates this to some extent by running all the four computations needed for mutation and non-mutation operations in a single thread, due to this TPS is high but this limits vertical scalability of server.

All the above issues slows down the cpu speed, or bring down the Instruction Per Cycle(IPC[1]). Below sections describes the solutions for the above problems by addressing the root causes. Partial Prototype was done using Jiny kernel[2]. Jiny kernel provides and implements same posix api as that of linux, due to this linux application can run as it is on Jiny kernel. Apart from Posix api, Non-posix api are added to Jiny kernel to address the root cause of problems.

3. Non-Posix File API

File Write Path: Fsync and write system calls combination is first method to implement Durability in the Database. Second Implementation method is

using mmap and msync syscalls. Here we cover only the immutable writes. Second method is more efficient then first.

File Read Path: File Read using read system call is first method. mmap system call and reading from memory directly is the second method. second method is efficient since it decreases number of system calls if the file fits in memory. In both the methods fadvise system call can be used to evict the un-required pages from memory to create sufficient free memory or to make the page cache of OS efficient.

A. Problems in Posix File API:

1. Calling synchronous Fsync system call for every mutation operation or at end of few batched operations. The system call is an expensive because of switch from userspace to kernel and it is a synchronous call.
2. Absence of low cost notification from OS to the userspace and vicerversa related to the amount of data flushed to disk.
3. Do not have priority among the writers. Most of the database like cassandra, Elasticsearch, ..etc has two types of writes, one is writing to translog that need at high priority and other is low priority file merging as part of housekeeping.
4. Explicit Calling of fadvise syscalls to evacuate pages of files from memory to create free memory for file read.
5. Lazy write wastes the disk bandwidth. It is not suitable for immutable file writes.

B. Algorithm used in Non-Posix File API:

Shared memory region is created using ioctl call for the file by OS to communicate between user level thread and OS, this can be optional feature. Shared memory for each open file is divided into two sections. “User to OS” section updated by user space thread and read by the OS. *OS to user* section updated by OS and read by user level threads. Below are the contents of the shared memory for each opened file:

- User to OS section:
  - Last-byte-written: This gives offset of last byte written to the file by the user. Data can be written using write call or using mmap method. This is an indication to OS during flushing of file content to disk. Since it is an immutable file, so this value will be continuously increasing.
  - Last-byte-not-needed: This gives indication to OS to free the pages from the pagecache till “Last-byte-not-needed”. This avoid calling fadvise syscall by the user. OS can scan this offset whenever there is a memory pressure or during housekeeping.
  - Last-byte-needed: This is an indication to OS for the readonly file. The read-ahead code will pick this and get the pages from the disk accordingly. If page is not present in memory then there will be page faults or blocking read calls that will slow down the application.
  - file Priority: priority of the data to flush. If there are multiple immutable files across the system, then priority helps to flush the data in the priority order. Application set priority as high if it wants to flush the data immediately as soon as possible even if the data size is small. This property is very useful, with the posix api application usually calls fsync when the data size is sizable, but in the Non-posix changing the priority will make the OS picks the data as soon as possible without any system call at the highest granularity, this as huge impact on efficiency and latency.
- OS to user Section:
  - Last-byte-flushed : This is last byte or highest offset in the file flushed to disk by the OS. Here file is assumed to be immutable file.

C. Techniques used in the above Algorithm:

The non-posix file api address the above problems with the following techniques:

1. Almost Zero system calls to increase IPC: After initialization, Non-posix API tries to use almost zero system calls. This is acheived by using the shared memory communication between user level threads and kernel. Along with this polling method is used on both the sides to avoid wait or sleep.
2. Early write to optimize disk io: Early Write means writing to disk immediately after complete writing block size data into memory. linux being generic system, it does not do early write because pages of the file can be overwritten in the near time and too frequent communication with OS need system calls. But in this design most of the files are immutable. Waiting does not help, flushing the data immediately after crossing blocksize will help in improving the disk io. OS will uses polling and shared memory to detect and write to disk as soon as possible.
3. Prioritised IO to decrease latency: Not all writes or reads need be done immediately, but need to be done eventually. There are two type of tasks, first type is io waiting by the client request, second is regular housekeeping tasks like merging immutable files,..etc. so first task need to completed as soon as possible to reduce the latency, and at the same time second tasks need to processed when ever the disk is idle without waiting from the user app or from the OS. This design pattern suits for immutable file and reading sequentially.
4. Evacuate unneeded pages Efficiently: Linux evict the pages based on LRU algorithm, but LRU does not suitable for all cases. Suppose if a file is read, and no longer needed in the near future, in such cases application can signal the OS using the fadvise calls. But fadvise is expensive since it need to be called regularly. This is addressed using shared memory communication and polling.
5. Disk IO bandwidth bottleneck*:* In the write path, the API will be more efficient if disk io bandwidth is not the bottleneck, this can happen in the following cases:
    a. If the speed gap between volatile memory and disk/Non-volatile memory decreases.
    b. If there are large number of disks attached to machine, so that overall disk bandwidth increases.
    c. If the user space thread as less writing when compare to other computations like read,..etc.
    d. asynchronous disk io requirement is less when compared to housekeeping disk io.

    In all the above cases, bottleneck will be shifted from io speed to the communication between user and kernel threads.
6. *Hugepages advantages:* In write path, mmap can be used to avoid write system call. but when the application touches a single byte in file buffer, OS need to flush entire 2M page this will be huge penalty. With new API only the smallest block like 512(size of block on physical device) can be flushed, this is possible with the *LastByteWritten* from shared memory as mentioned in the previous section, this is not possible in existing posix api due to absence of file offset in mmap.
7. ReadAhead : This software construct is already present in the OS to speed up user read request. Here the some differences:
    a. OS detects automatically based on the read pattern, but here the app knows it is going to read all the content of file once.
    b. readahead need lot of free pages and disk bandwidth, so read ahead need some coordination with evacuation-technique for free pages

and prioritised the read request along with other io requests.

D. Summary of changes needed in Application

The following are the changes needed in the application to implement non-posix API:

1. Setting shared memory section for file: Ioctl syscall is used to set up the shared memory section. This is done during file opening.
2. ioctl is used to start and stop the polling inside the kernel. This can be done during initialization or periodically.
3. Fdatasync, mysnc, fadvise syscalls are completely avoided. Both user level threads and Kernel will be polling and updating the shared memory to communicate to other side.

With Non-posix API, user level threads will be populating in to volatile memory using mmap without any slowdown from system calls or blocking from OS. On other hand kernel threads will be flushing dirty pages in a priority manner into non-volatile disk without any interruption. Here both kernel and user level threads will be running in parallel with shared memory as the communication between them, both the threads will be polling, due to this IPC will be high.

E. All put together:

Early disk writes, read ahead advance, memory evacuation, priority io, regular polling, almost zero system calls, Hugepages usage will make entire io faster and efficient.

The above api will make application threads and OS threads to execute in parallel with almost zero system calls.

G. Prototype Test Results: Yet to publish.

4. Shared memory interface

The Ring buffer inside the shared memory is used to exchange messages between two entities. The entity can be process, thread from within same process or threads from two different process or between OS and user space threads.

Advantages of shared memory compare to sockets are:

1. System calls are not needed to exchange the messages.
2. Memory copy is not needed when compare to sockets.
3. Bulk copy of messages can be possible in the shared memory.
4. Polling can be used instead of blocking. This decreases latency and increases TPS at the cost of cpu cycles.

A. Test Results

"Redis with Socket" versus "Redis with shared memory" is compared with various degree of batching of requests[3]. In the below table-1, Shared memory as out performed the socket version by a 37X times with one request is send per network packet(i.e batch=1). Shared memory as variation of with and without copy, without copy means buffers from shared memory is reused to avoid the extra memory copy. Zero copy as shown improvement on top of the shared memory improvements. Zero copy is not possible with the socket. Other implementations of shared memory are LMAX[6].

TABLE I
SHARED MEMORY VERSUS SOCKET COMPARISON

SNO	Type	TPS	Description
1	Redis with Socket (batch=1)	54K	Batch=1 : The ratio of difference is High (1:37), because of lot of round trips or less batching in socket.
	Redis with Shared memory (batch=1)	2000k	
2	Redis with Socket (batch=10)	576k	Batch=10 :The ratio of difference is Medium (1:5) ,because round
	Redis with Shared memory (batch=10)	3000k	

			trips is reduced by 10.
3	Redis with Socket (batch=50)	1800k	Batch=50 :The ratio of difference is relatively less(1: 1.8), because round trips is reduced by 50.
	Redis with Shared memory (batch=50)	3300k	
4	Redis With Shared memory + zero copy	TODO	
5	Server-client With shared memory (batch=10)	3.5M	Better with Zero copy
	Server-client With shared memory + zero copy(batch=10)	4.5M	
6	Server-client With shared memory (batch=50)	8.3M	Better with Zero copy
	Server-client With shared memory + zero copy(batch=50)	12.5M	
batch = number of requests per network packet.			

B. Analysis of Tests

1. Shared memory performs better in batch=1 when compare to batch=10: The more the message exchanges the better the improvement, this is due to savings from system calls. In batch=1, there will be 10 times more system calls when compared to batch=10.
2. Reusing the buffers without freeing the buffer causes less memory copies. Memory copy is a expensive operation, due to this TPS is higher.

5. DPDK for NoSQL

DPDK is proven and as delivered better network throughput in Networking workload like Network Function Virtualization(NFV). In the NoSQL Database, Network path is one of the major bottleneck especially for non-mutation operations. To transmit the packets at high rate, the application need to bypass the system calls and kernel. DPDK is implemented as a user space library. DPDK and shared memory can be used to route the network packets efficiently from the NIC to database and vice versa. With DPDK the networking stack of OS is completely bypassed.

A. Advantages of DPDK

The DPDK library in user space can receive/send the packet at high rate using poll mode driver inside the kernel, the following are the advantages:

1. Zero system calls: The interface between proxy and OS, and between proxy and server does not involve any system call.
2. Zero copy: The buffer from NIC-proxy-server and vice versa are done through shared memory inside the huge pages.
3. Uses of HugeTLB pages and bulk copy: Buffers are pre allocated inside the Huge page memory, and transmitted in bulk instead of one packet each.

B. Disadvantages and Mitigation

At low load, poll-mode driver inside the kernel and userspace will consume cpu. This can be mitigated by converting polling mode into blocking mode when the load is less dynamically.

6. Other Optimizations

A. NUMA Awareness

NUMA awareness can make cpu cache friendly. There will be lot of churn of data inside the cache. With proper pinning of threads to the appropriate core will avoid the pollution of L2, L3 caches.

B. Java Off-Heap Memory

Since most of the NoSQL are implemented in Java, Performance of JVM will be critical. One of the problem in Java is Memory management during the processing of request and response, there will lot of memory buffers are created and destroyed during the life cycle of request response cycle, this creates extra pressure on the JVM garbage collector, and causes periodic cpu and memory spikes. This can be minimised by using the reusable off-heap memory. Off-heap memory can be used for the following two purposes:

1. The off-memory also well connects with the shared memory described in the earlier section. with this the network requests coming from outside the jvm can receive/transmit

without memory copy by the off-heap memory module.

2. off-heap memory can used instead of jvm heap, so that the to java objects in the request-response can be reused.

C. Non Posix API for Network IO

If DPDK was not used then below can be used as alternatives. Extending epoll to receive as well as sending the packets. Sending/receiving packets to multiple sockets with a single system call. The new posix api will merge the functionality of send, recv and epoll in to single new syscall. The advantages of new epoll call are as below:

1. Avoid calling multiple receive calls for each socket, suppose epoll is called, and after that individual socket is used to read syscall repeatedly. Using the new syscall, read calls are not needed, this saves large number of system calls.
2. Avoid calling multiple send calls. Along with the poll, multiple send buffers can be feed , so that os internally calls send on multiple sockets.

With this, system calls will be reduced by a large number.

7. Putting All together

Putting all the above optimizations together and applying on NoSQL databases like Redis, Kafka, Cassandra, ElasticSearch,..etc. This as shown in. Figure-1. optimizations are represented in oval shape. The following are advantages when compared to the non-optimized version:

1. **Network Path:** DPK and network proxy are used for network path as shown in the figure. NoSQL databases uses network proxy instead of socket layer.

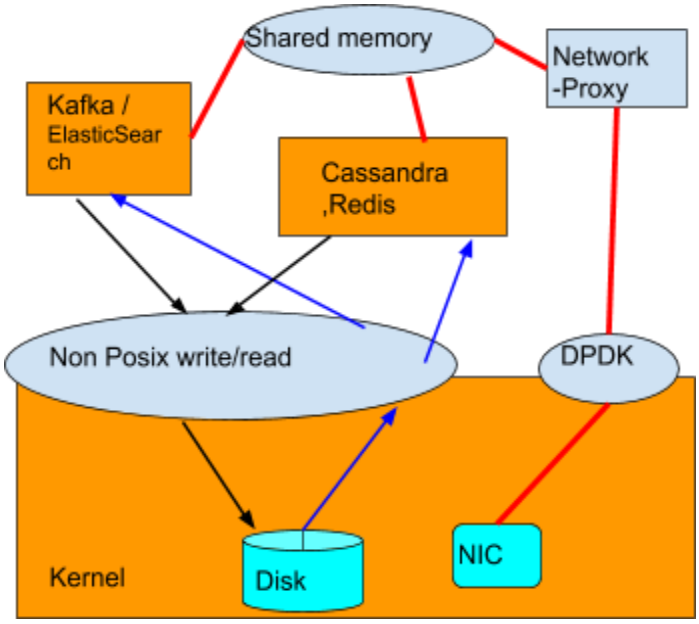

FIGURE-1

2. **Disk Write Path:** Cassandra, Kafka can flush all files like append log, commit log, sstables using non-posix api. Both applications do not need any system calls to flush the data. Huge TLB pages can be efficiently do mmap the commit logs. OS can flush portion of large page instead of complete page with the help of offset inside the fsync shared memory. Priority of flushing helps Cassandra to flush commit log before sstable files.
3. **Inter Component Communication(ICC):** Shared memory can be used to communicate between proxy, Kafka, Cassandra and other app without any system calls, this can be done with zero copy.
4. **Disk Read Path:**Read path can be improved by caching the hot pages in the memory without depending on linux pagecache, To do this unwanted pages need to be removed from the memory using fadvise.
5. **cpu core and cpu cache**: The application need to be NUMA aware by pinning the appropriate threads of the applications to the corresponding cores.

Summary of changes in NoSQL databases:

1. Network Path: NoSQL databases listen's on shared memory for network messages. NoSQL database receives/transmit network messages through network proxy instead of socket layer. Tcp/IP stack is implemented inside the network proxy as part of DPDK library. Entire Network path does not involve any system call and it does with zero copy with huge pages. One core each in Kafka, Cassandra, proxy and kernel will be in poll mode to listen the packets from shared memory. Encoding and decoding of the message can be done as part of the polling threads. Example: Redis as shown 32X improvement with the shared memory when compared to sockets. Network Proxy need to be started before starting the Kafka/Cassandra/ES.
2. Non-Posix api and priority use case:
    a. Kafka: Using Non-Posix api for commit logs. At any point of time there will be large number commit log files if there are active topic and active consumers. Currently Uses write+fsync model, so there will large number of write and fsync calls. This can be changed to mmap+msync model, by doing this write system calls and fsync syscalls are totally avoided.
    b. ElasticSearch: Using Non-Posix api for Translog and Lucene files. During Lucene merge, priority can be set low, rest of the the time can be set high to reduce the latency.
    c. Cassandra: Using Non-Posix api for commit log and sstables can. commit log can use high priority and rest can use low priority.
    d. Redis: Using Non-posix api for Commit log.
3. Multi Core Friendly changes:
    a. Attaching the threads to the specific cores.
    b. Using Huge TLB pages for commit log, sstables, .. etc.

8. Summary

Improvements in Disk path, network path and communication between various components of NoSQL Databases are described in this paper.

Inefficiency in write disk path, read disk path, network path and communication between components are described in this paper. The Root cause and corresponding solutions for the above three are summarised below:

1. **Almost Zero system calls:** Non-posix Fsync for write path, Shared memory for ICC and DPDK for network path avoids system calls completely. Due to zero system calls IPC and TPS of the components will improve drastically. There will be small percentage of system calls during file or socket setup.
2. **Efficient ICC:** Shared memory and DPDK have Zero buffer copy. Polling mode in DPDK, shared memory and Non-posix fsync makes communications faster and efficient.
3. **Parallelism between user level and kernel level threads:** user level and kernel threads will be executed in parallel without blocking, this is due to polling and shared memory communication.
4. **Multi Core Friendly:**
    a. Polling mode without system calls is well suited for multi core when compare to single core.
    b. NUMA friendly: polling mode and less system calls makes more NUMA friendly and cpu cache friendly.
    c. HugeTLB can be efficiently used with Non-posix fsync, shared memory and DPDK.

In Summary Zero System calls, Batching of Requests, zero memory copy, priority, Numa friendly, shared memory communication and Huge Pages improves performance of NoSQL Databases on multi core machine by many folds.

Improvements further: Disk write path , Network path and Inter Component communication are addressed to larger extent. But disk read path need some more refinement.

References

[1] Instruction Per Cycles paper , https://github.com/naredula-jana/Jiny-Kernel/blob/master/doc/Perf_IPC.pdf.

[2] JinyKernel: https://github.com/naredula-jana/Jiny-Kernel/.

[3] Shared Memory Implementation: https://github.com/naredula-jana/PerfTools/tree/master/SharedMem

[4] DPDK : https://software.intel.com/en-us/articles/dpdk-performance-optimization-guidelines-white-paper

[5] DPDK : http://www.dpdk.org

[6] LMAX disruptor: “https://lmax-exchange.github.io/disruptor/”