PolarFS: An Ultra-low Latency and Failure Resilient Distributed File System for Shared Storage Cloud Database

Seminar: Techniques for implementing main memory database systems (Winter Term 2020/2021)

LOHMANN, NIKLAS

In publication [4], the protocol ParallelRaft and the file system PolarFS for distributed cloud database systems are introduced. ParallelRaft promises great improvements to the Paxos protocol Raft when facing I/O intensive workloads. By exploiting concurrency, ParallelRaft achieves higher throughput and allows for multi-threaded request processing, however it does not seem to improve performance or lower latency for smaller workloads as implied in the paper [4].

1 INTRODUCTION

With the progress of digitalization and the trend towards big data, the demands for database systems are growing in terms of availability, flexibility, latency and throughput. Tackling different bottlenecks is an important step towards all these goals. The architecture using PolarFS and the protocol ParallelRaft taken from an industry paper [4] will subsequently be explained in detail, the protocol ParallelRaft will also be tested with a sample implementation. It promises higher performance and lower latency for I/O intensive scenarios in comparison to Raft that it is based on.

2 MOTIVATION FOR POLARFS AND PARALLELRAFT

To fulfil the needs of customers, large database service providers are using the concept of distributed database systems that are spread among multiple storage and compute instances in their data centers. The data is not held in memory at all times, yet the database service is constantly available on request. Since the workload is balanced among many different clients' requests, the resources are used more efficient so the client company gets a much cheaper and easier way to deal with their databases than building and administrating their own database server.
Multiple problems with modern distributed database solutions are identified in the paper [4]: First, they typically don't take advantage of recent techniques like RDMA and NVMe SSDs, which would accelerate the performance drastically. Due to the versatility of these service providers they often decide against highly performant clusters designed specifically for high computation or latency demands and provide them as instances to clients instead. These instances are using dedicated resources, have their own SSD and while offering good performance they do not offer much flexibility in regard to shared architecture demands or larger datasets exceeding the capacity of the local storage.
The second weakness are the file systems used by database servers. They can become bottlenecks in scenarios of high I/O frequency and volume due to context switches and interrupts for read and write operations using up large parts of computation time.
Another possible bottleneck is the sequential writing of changed or new data to disk and the following replication to protect from data loss. Paxos consensus protocols typically used to guarantee the correct and immediate replication over network connections do not offer the resilience required for low latency demands. When one connection becomes slow, the whole writing process gets slowed down as well.

3 ARCHITECTURE

The suggested cluster is strictly divided into compute and storage nodes. This makes it easier to customize and optimize each section individually, like choosing SSDs for the storage of the data. It also allows the compute nodes to work almost fully independently of each other, they only synchronize file system metadata, but the actual data is only synchronized in the storage pool, not between the nodes. All this is done while also keeping

the latency low by exploiting Remote Direct Memory Access (RDMA). This technology allows direct transmission of main memory data to other machines' memory without involving the CPU like a typical TCP connection would, thus not interrupting the workload and minimizing protocol expenses. The memory communication then is treated like a queue and polled regularly instead of using interrupts, to avoid context switches and the resulting overhead.
The compute nodes host database instances and use the library of PolarFS to work with the distributed storage system. PolarFS is implemented fully in user space to offer a file system avoiding context switches and interrupts. This saves relevant computation time in I/O intensive scenarios. Every request to the database is analyzed according to the metadata PolarFS collected about the data hierarchy. More simply put, on start up the library loads the directory tree and mapping tables of the stored data into main memory and then knows how to map a request to a certain data chunk. This information is then passed with the request to the PolarSwitch, which is a single background process on each host. The PolarSwitch has the needed metadata to map the chunk of the request to the actual location in the storage pool and sends the request to that instance via RDMA.
The storage pool consists of multiple nodes called chunk servers, each responsible for many data chunks of 10GB each. Among the chunk servers, a shared-nothing architecture is used, so every chunk has a dedicated SSD disk and CPU core. Each chunk also has a separate log like main memory database systems, it uses Write-Ahead Logging to ensure data durability. To prevent data loss even in the case of disk crashes or physical damage, every chunk is replicated to three other chunk servers. These replicas serve as backups and are not used for any request processing. However, they have to be constantly synchronized with the original chunk in order to guarantee that even the requests right before the server crash are either saved or reported as unsuccessful to the client. These replicas will be called "followers", in contrast to the "leader" chunk being regarded as the original data that is used to process requests.

4 PARALLELRAFT

ParallelRaft was developed to make the described replication process more performant and to reduce latency. The protocol is responsible for the way the replication of requests is handled and exploits concurrency techniques, while still remaining as resilient as the original protocol Raft.

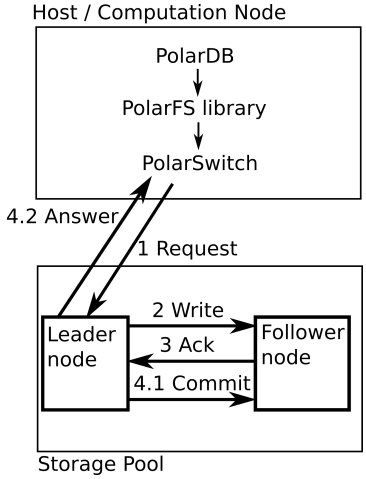


Fig. 1. The communication in the cluster for a write request

Raft uses a sequential approach to guarantee easy and safe data replication. Requests are first sent to the leader, from there to the followers and written to the log on all corresponding chunks. After saving the request to log, the followers send an acknowledgement to the leader to signal that the request is now synchronized. After receiving these acknowledgements from all followers, the leader commits the actual changes resulting from the request and edits the data in the storage according to it, a commit message is also sent to the followers for them to do the same. After sending out the commit message, the leader can process the next request in the queue. This makes sure that requests are processed strictly in order, but Raft can not handle large amounts of I/O requests very well due to the two-way communication taking place before another request can be processed.
At the core of ParallelRaft is the possibility of out-of-order processing. When two requests are not writing to the same block of the data chunk, there is no risk of overwriting each other, they could be writing their changes to disk in any order. This is made possible by the precise mapping to data blocks and allows for multiple requests to be processed in parallel before any of them is committed, multiple threads can also be used to improve concurrency. Even with a single thread, the leader can use the time spent waiting for acknowledgements by processing and forwarding more write requests. More generally, any nodes can be kept busy while waiting for answers from their counterpart, which results in higher throughput. There is a limit N on how many requests can be processed ahead of the most recent commit. The leader can commit requests that are acknowledged only by the majority of nodes, since from that point, any request can be reconstructed by trusting a majority of nodes. With out-of-order acknowledgements also come out-of-order commits and therefore, short lived holes in the log can occur.
This comes with a challenge to the correctness of ParallelRaft: How can the protocol guarantee that no conflicting requests are committed in the wrong order?
This is solved by a look-behind buffer that every request contains. It consists of the Logical Block Addresses (LBA) of the previous N-1 entries. This logical block address identifies the block that is edited, if two requests access the same LBA, they are conflicting and have to be applied in order. If the later request reaches any node before its conflicting predecessor, it will be put into a pending queue and only processed once the conflicting previous request has been committed. Since there is the possibility of multiple holes in the log, we need to track the LBAs of all missing requests and no new request may write to these pages before the earlier requests are finished. Any new request has to compare the LBA it demands to edit to this list of blocked LBAs and the LBAs of its look behind buffer - if the corresponding requests haven't been committed yet - and may only get processed if it doesn't conflict, otherwise it gets put into the pending queue.
Whenever a request gets committed, it checks the pending queue for requests that write to the same LBA and moves them back into the input queue as if they just reached the node. This way they don't block the input queue while waiting for the conflicting request and only the requests that might now be unblocked get checked again.

5 IMPLEMENTATION

To check the claims made in the paper [4] regarding the throughput and latency improvements gained by the usage of ParallelRaft, a sample implementation is made as a single-machine process with multiple threads. For each leader and follower node, there is an I/O thread and a writer thread that writes the protocol to disk. The production of requests is done by a separate thread, simulating the workload of random write requests. This thread sends packages of requests in fixed intervals and will pause sending when the input queue of the leader reaches a certain size to prevent overflow of the queue.
The implementation of ParallelRaft may use multiple I/O threads per node in order to increase throughput even further, however this isn't used in the experiments due to the few cores of the testing machine limiting the achieved effect.
Since the communication via RDMA is not used for the experiment, requests are instead getting passed through thread-safe queues directly. The data structure is therefore the same as for RDMA and with RDMA reaching up

to a million transactions per second [3] and latencies of few microseconds [5] the saved latency and performance will be negligible.
Writing the data to log is implemented with a separate writer thread, but disabled for performance tests to avoid disk-writing as the bottleneck.

6 EXPERIMENTS

To compare the performance and latency of ParallelRaft and Raft in different scenarios, the package size S is varied as well as the interval t in milliseconds between packages and the probability P that two packages write to the same page. The maximum hole size is set to two, as recommended in the paper [4].
The latency of a package is measured from the time of its production to when the leader commits, which would result in a positive answer to the client.

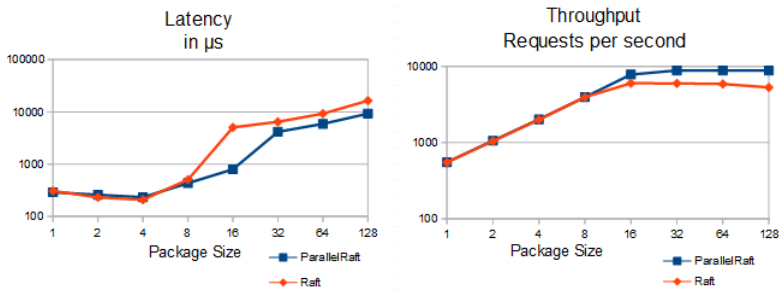
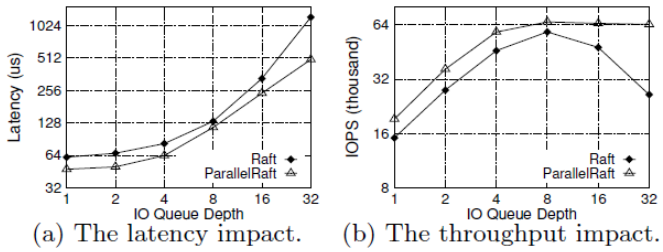


Fig. 2. Throughput and latency results for varying input package sizes



(a) The latency impact. (b) The throughput impact.

Fig. 3. Results from the paper [4]

While the throughput stays almost identical with growing package sizes up to 8, the performance of Raft is significantly lower for larger input packages as seen in [Fig. 2], which is due to its sequential algorithm not handling large input queues very well. When it cycles through the request queue, it is only looking for exactly one request, while ParallelRaft is looking for 3 different requests because it allows for holes of size 2. The latency of ParallelRaft is similar to Raft for small input packages and then beats Raft's latency drastically for large package sizes, answering requests in about half the time at highest package size. Again, this is a result of the long input queues slowing down Raft. Since the package intervals are fixed to 1ms for all package sizes, the growth of the throughput up to package size 8 is observed proportional to the package size because the amount of input requests grows as well. At the point when the performance is at its maximum, the threads can just process all the requests before the next ones come in, therefore not losing any performance due to growing input queues. The

low throughput at lower package sizes is not due to the protocol not dealing well with them but simply the low input rate.
Compared to the findings of the original paper [Fig. 3], these results seem to contradict in some points: The latency starts at a much higher value, which might be due to the limited CPU used for testing and time measurement inaccuracies. The performance of Raft also doesn't drop too far with higher input depth.
The experiments above were conducted with 255 different pages and the requests evenly distributed over those, so the probability of two randomly chosen requests to conflict is exactly 1/255. Although a chunk contains a lot more blocks that can be accessed, it often contains a few blocks of special interest [2]. This is of course not a problem with Raft, but ParallelRaft could have problems with requests conflicting more often. When the amount of pages to choose from gets smaller, requests block each other more often, the performance is however not much impacted, as [Fig. 4] shows. The throughput and latency are almost constant regardless of conflict probability, the small performance drop at package size 128 might be due to worse caching behavior. This experiment was conducted with a package size of 32 at the throughput peak of ParallelRaft.

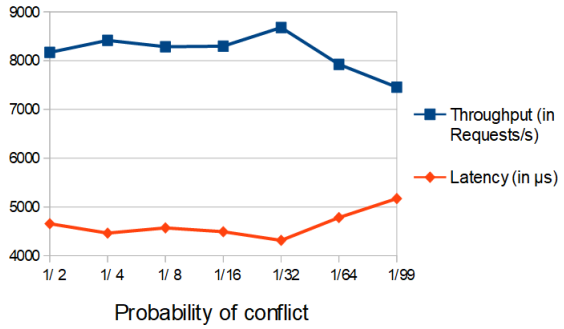


Fig. 4. Throughput and latency results for varying conflict probability

7 CONCLUSION

With the experiment supporting the claim of the paper [4] that ParallelRaft reaches higher throughput and lower latency in comparison to Raft, it is an option worth exploring for large cloud database providers. The possibility of multiple I/O threads responsible for the same chunk can also be useful for load balancing with threads getting added for higher workloads.
However, it is still questionable whether ParallelRaft is faster than Raft at low input depth, not only do the experiments not support this, it also can't be explained with the concurrency. The processing of single requests should not result in any out-of-order advantages, but rather a slightly higher computation effort due to the LBA checks of ParallelRaft.
Further experiments could be conducted with multiple I/O threads per node and a real RDMA setup in order to optimize towards non-blocking behavior and the behavior of RDMA queues.
ParallelRaft does come with a few implementation difficulties, like leader election and catch up techniques being a little more complex than with Raft. This is not an obstacle for larger corporations of course: The protocol and the architecture have already been put to use by Alibaba [1], a large Chinese corporation offering (among other products) cloud databases and the source of the original paper [4]. ApsaraDB for PolarDB is presented as a "next-generation relational database" and seems to be the company's flagship for cloud databases. It also seems to have been used for Alibaba's own online shop, similar to Amazon, with the claim it supported 1 billion orders in just one day.

REFERENCES

[1] Alibaba Cloud. 2020. Architecture of ApsaraDB for PolarDB. https://www.alibabacloud.com/help/doc-detail/58766.htm?spm=a2c63.p38356.b99.7.50401625l0SRUM

[2] Claudio Gutiérrez-Soto and Gilles Hubert. 2014. Probabilistic Reuse of Past Search Results. In *Decker H., Lhotská L., Link S., Spies M., Wagner R.R. (eds) Database and Expert Systems Applications. DEXA 2014. Lecture Notes in Computer Science, vol 8644. Springer, Cham.* https://doi.org/10.1007/978-3-319-10073-9_21

[3] Philipp Fent; Alexander van Renen; Andreas Kipf; Viktor Leis; Thomas Neumann and Alfons Kemper. 2020. Low-Latency Communication for Fast DBMS Using RDMA and Shared Memory. In *36th IEEE International Conference on Data Engineering (ICDE 2020).*

[4] Wei Cao; Zhenjun Liu; Peng Wang; Sen Chen; Caifeng Zhu; Song Zheng; Yuhui Wang and Guoqing Ma. 2018. PolarFS: An Ultralow Latency and Failure Resilient Distributed File System for Shared Storage Cloud Database.

[5] Jiuxing Liu; Jiesheng Wu and Dhabaleswar K. Panda. 2003. High Performance RDMA-Based MPI Implementation over InfiniBand. In *International Journal of Parallel Programming 32, 167–198.* https://doi.org/10.1023/B:IJPP.0000029272.69895.c1