Short Paper

UniCache: Efficient Log Replication through Learning Workload Patterns

Harald Ng
KTH Royal Institute of Technology
hng@kth.se

Kun Wu
KTH Royal Institute of Technology
kunwu@kth.se

Paris Carbone
KTH Royal Institute of Technology
RISE Research Institutes of Sweden
parisc@kth.se

ABSTRACT

Most of the world's cloud data service workloads are currently being backed by replicated state machines. Production-grade log replication protocols used for the job impose heavy data transfer duties on the primary server which need to disseminate the log commands to all the replica servers. UniCache proposes a principal solution to this problem using a learned replicated cache which enables commands to be sent over the network as compressed encodings. UniCache takes advantage of that each replica has access to a consistent prefix of the replicated log which allows them to build a uniform lookup cache used for compressing and decompressing commands consistently. UniCache achieves effective speedups, lowering the primary load in application workloads with a skewed data distribution. Our experimental studies showcase a low pre-processing overhead and the highest performance gains in cross-data center deployments over wide area networks.

1 INTRODUCTION

State machine replication (SMR) has become the de facto approach for building distributed databases that provide strong consistency with fault tolerance. This form of replication is typically based on having a primary server that orders all changes to the database into a log that is replicated on each replica server. The log defines a single execution order for all replicas such that the state remains consistent across them. Apart from featuring in the replication layer of both relational [14, 19, 34] and non-relational [8, 38] databases, state machine replication is also used for coordination services [17, 24] and high-availability in data management systems [1–3].

Using log replication with a designated primary is the most adopted approach to implement SMR due to its simplicity, omitting the need for conflict resolution mechanisms. However, primary-based protocols have the problem that the primary can become a bottleneck. With the additional responsibility of ordering changes in the log and disseminating them to all backups, the network I/O at the primary is significantly higher than other replicas. Recent work has attempted to reduce the primary's network I/O with erasure coding [36] and custom transmission schemes [38]. However, such approaches come with inherent trade-offs; erasure coding lowers the network and storage costs, but backup replicas have incomplete state which affects practicality. Custom transmission schemes use other servers to help the primary to forward the data. This lowers the primary's network I/O, but reduces fault-tolerance as any failure in the transmission path will cause down-time. Furthermore, the additional network hops result in higher commit latency.

© 2023 Copyright held by the owner/author(s). Published in Proceedings of the 26th International Conference on Extending Database Technology (EDBT), 28th March-31st March, 2023, ISBN 978-3-89318-092-9 on OpenProceedings.org.
Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

We propose *UniCache* to address the primary bottleneck problem by minimizing the amount of transmitted data. UniCache expands on primary-based protocols with the ability to optimize network I/O performance by learning and adapting to the application workload. It exploits workload skews such as hot spots and time-varying skews to efficiently transmit data that is frequently replicated. This approach enables UniCache to transparently optimize replication without the drawbacks of current approaches that affect practicality and fault-tolerance.

The design of UniCache is based on two key observations that are distinctive for log replication protocols. 1) Every replica has local access to a committed prefix of the log that corresponds to a consistent view of the application workload history. 2) The replicated log entries represent commands for an application with some recurring workload pattern. From these observations, we introduce the following novelty with UniCache: Each replica maintains a local cache containing popular application commands which is deterministically derived from the committed prefix of the replicated log. It is used as a lookup table that maps commands to smaller encoded formats. If a command to be replicated exists in the cache, the primary sends the encoded version of it. The backups will then decode it using their local cache to get the command in its full form. The encoding and decoding are safe as each replica builds its cache from the committed prefix of the replicated log in a deterministic manner. Thus, each replica is essentially using a cache that is consistently replicated but without any explicit coordination. Furthermore, we use an ML-based eviction policy to be able to adapt to changing workload patterns.

We claim the following contributions with UniCache:

- We describe the primary bottleneck problem in log replication protocols and provide an overview of the trade-offs with existing approaches that target the problem.
- We present UniCache, an expansion to primary-based log replication protocols that exploits the application data and its workload to reduce network transmissions.
- We present an evaluation with synthetic benchmarks that showcases when UniCache can provide the most performance gains, and benchmarks with real-world data to demonstrate the practicality of UniCache. Our results show that UniCache can improve performance significantly, achieving up to 4.5x better performance in high latency settings where data transmission is a bottleneck.

2 THE PRIMARY BOTTLENECK PROBLEM

Log replication protocols such as Raft [32] and Paxos-based [28] derivatives guarantee safety such that the replicated log is consistent at every server and the decision to commit a command is durable. To provide liveness, a designated primary is elected to lead the log replication. A prepare phase is first established when the primary is elected in order to ensure that the server is equipped with the most up-to-date log. From that point, the protocol enters the accept phase which is repeatedly performed

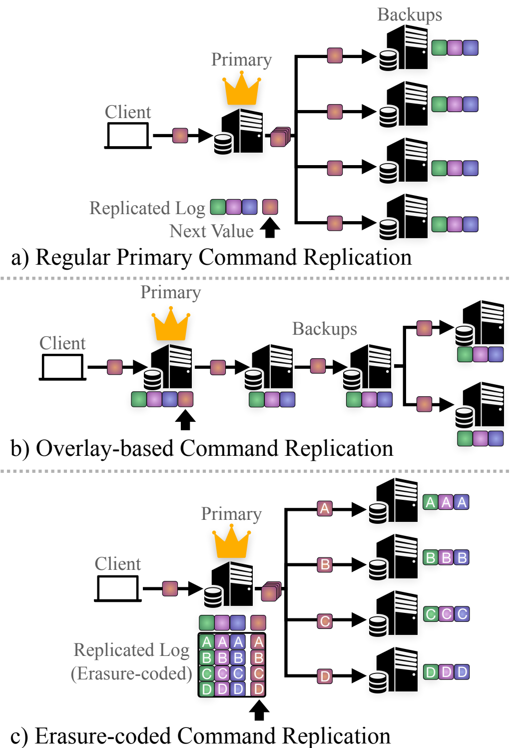

Figure 1: Approaches in I/O-reduced log replication

to populate the replicated log. During this phase, the primary handles all client requests and orders them in the log. It then replicates these commands in log order to the backups. When a majority have replied to the primary that a command is replicated, it is committed and the primary sends this decision to all backups. The accept phase thus involves heavy data transfers when the primary repeatedly transmits commands to the backups. As seen in Figure 1a, the network I/O at the primary grows linearly with the number of backups.

Early adopters of log replication protocols were mainly for metadata services, such as Google's Chubby[17] and ZooKeeper[24]. However, log replication is now used in the critical path of distributed databases such as TiDB [23], MongoDB [38], and CockroachDB [34] that replicate data at the scale of terabytes with hundreds of thousand requests per second [5, 12]. With these increased data volumes, the primary bottleneck in terms of network I/O becomes a serious challenge for achieving the latency and throughput objectives for large-scale systems.

2.1 Trade-offs for Reducing Network Costs

Previous work has achieved network I/O reductions at the primary using different methods. By using custom transmission schemes [38] or adding additional servers that disseminate the data [37] similar to network overlays, the network I/O at the primary is reduced but incurs higher commit latency due to more network hops. Furthermore, the use of overlays reduces the fault tolerance since any server along the dissemination path becomes a single point of failure (see Figure 1b).

CRaft[36] introduced erasure coding that enabled the primary to only replicate fragments of the data at the backups. However, this implies that only the primary has a full copy of the data. This is not practical in applications such as distributed databases

Table 1: Typical command sizes in systems based on log replication protocols.

System	Protocol	Typical command size (bytes) [13]
TiKV [11]	Raft	8 - 4096
dragonboat [4]	Raft	16 - 1024
TigerBeetle [10]	VR	128
etcd [20]	Raft	128 - 4096
MongoDB [38]	Raft	1024
ZooKeeper [24]	Zab	1024

where each replica needs the state to serve client requests. It is worth noticing that in both described approaches, the IO optimizations do not occur transparently. Instead, they substitute or critically modify core components (e.g., the transmission path and the replicated data) of the log replication middleware and therefore alter its existing usage assumptions, properties and guarantees.

2.2 Workload Skews and Data Sizes

Log replication protocols replicate application-provided data in a mechanical fashion, being completely agnostic of data trends and therefore unable to employ sophisticated optimizations. However, if we look more carefully into the typical replicated data sizes and trends, there is clearly untapped potential for optimization. The *data size* has a several-fold effect on the network I/O at the primary as the data is sent to all backups. Table 1 summarizes the typical sizes of replicated system commands. These range between 128 and 4096 bytes.

Real-world applications are also known for exhibiting skew in their workloads. Distributed databases [23, 34, 38] typically have OLTP workloads where different skew types are common [30, 33]. *Hot spots* are a small number of objects that are accessed more frequently than the others, for instance, celebrities in a social network application. *Temporal skew* occurs when certain objects become popular at specific time intervals, e.g., travel destinations that are trending during certain seasons. Having such types of data skew over log replication means that the primary will blindly re-transmit the same data to backups. Given the observed data ranges of 128 – 4096 bytes, this implies a significant amount of data re-transmissions that amplify the primary bottleneck problem, which seeks a principled and generalized solution.

3 UNICACHE

UniCache is an expansion to log replication protocols that exploits recurring workload patterns to reduce the network I/O. The core idea is to maintain a cache with data that is repeatedly replicated so that it can be encoded and transmitted over the network in a compressed form. As illustrated in Figure 2, the cache resides locally on each server and is used for encoding and decoding to achieve lossless compression of the transmitted data. UniCache therefore only modifies the representation of the data being sent over the network but not the communication path or the data that is replicated. As a result, the network I/O can be reduced while avoiding the trade-offs discussed in §2.1 such as additional network hops and incomplete replicas. There are several key questions that need to be addressed related to the design of UniCache:

- How is UniCache integrated into a log replication protocol? (§3.3)
- How does UniCache guarantee correctness? (§3.4)

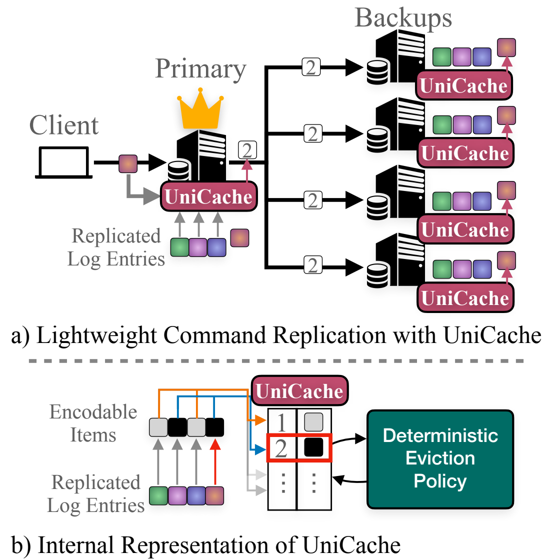

Figure 2: Log Replication with UniCache

- How does UniCache dynamically adapt to changes in the workload? (§3.4)

3.1 Log Replication Overview

UniCache is a technique that can be extended to any leader-based log replication protocol such as Raft [32], VR [29], and Zab [25]. These protocols share a similar design where one of the servers is elected as the primary which coordinates the replication of a strictly growing log. The consistency of the replicated log is defined by the following safety properties:

- *Uniform Agreement*: For any two servers that decided logs L and L' respectively then one is the prefix of the other.
- *Integrity*: If a server decides on a log L and later decides on L' then L is a strict prefix of L'.

These properties imply that the log at all servers are consistent and a backup always has some prefix of the primary's log.

Different protocols use different terminology for similar state and messages. For simplicity, we will describe the general design of log replication protocols based on the Paxos[28] terminology. When a server is elected as the primary, it first performs a prepare phase[1]. The prepare phase is for ensuring that the primary adopts all possibly committed entries before extending the replicated log. The primary initiates the prepare phase by sending a message to all backups to announce itself as the new primary. This message contains metadata that indicate how updated the primary's log is. Each backup replies with its corresponding metadata and any log commands that the primary is missing. When the primary has received a majority of replies, it adopts the commands provided by the most updated backup by appending them to its log. This ensures that any committed entry is not lost. The primary then synchronizes the log with the backups by sending the commands that they are missing. This concludes the prepare phase, which is only performed when a new primary is elected. New commands proposed by the client can now be appended to the log in the accept phase. Each new command is handled by the primary which appends it to its local log and sends it to the backups via an ⟨Accept⟩ message. When a majority have acknowledged that the command is appended to their local log, it is decided (or committed). The primary announces the commit decision by sending ⟨Decide⟩ to the backups.

The primary sends more messages which also contain more data compared to the backups. In the accept phase, the primary sends the command to every backup and handles all the acknowledgements. This implies that the network I/O at the primary grows linearly with the data size and the number of servers. In the prepare phase, both the primary and backup need to send data. Reducing the network I/O in the prepare phase is another interesting topic, however we do not consider it in UniCache as the prepare phase is only performed during primary fail-over and not critical to steady state performance.

[1] Raft does not have a prepare phase but instead continuously performs log synchronization while replicating. The elected primary must therefore have the most updated log among a majority to guarantee safety.

3.2 Data Pre-processing

UniCache introduces a local cache at each server. The cache is a bidirectional map where each item is associated with an index (integer). An item in the cache can be a log command or some partial data of a log command. The associated index is the encoded representation of it. Depending on the application, some data in a command might not be possible to cache (e.g., unique ids). Thus, we provide an interface for developers to specify which data in the command should be used by UniCache. The interface consists of the PreProcess and Recreate functions (see line 1-2 in Appendix A). PreProcess takes a command as an input and produces two lists; one with data items that might get encoded by UniCache, and one with data that cannot be cached and should therefore just be sent in its original form. The Recreate is the inverse of PreProcess for recreating a command given these two lists. As an example, consider the following SQL query which creates a new member with some personal information, the timestamp of becoming member and the membership level:

```
# original query
INSERT INTO members
VALUES ('John', 'Doe', 'Dentist', '2022-01-01 10:10:17', 1)
# after pre-processing
encodable = [INSERT INTO members VALUES (?, ?, ?, ?),
            'John', 'Doe', 'Dentist']
not_encodable = ['2022-01-01 10:10:17', 1]
# after encoding
items = [encoded(1), encoded(4), decoded('Doe'), encoded(35)]
not_encodable = ['2022-01-01 10:10:17', 1]
```

As the timestamp will not be repeated and the membership level cannot be further compressed (already an integer), these fields will not be used in UniCache. However, the query might be replicated with other arguments in the future, thus the template of it will be considered by UniCache. The other fields could be repeated (in the same query or other queries) and therefore also used by UniCache. In this example, the query, the first name, and the occupation were found in the cache and encoded as integers by UniCache. The surname "Doe" was a cache miss and will thus be sent in its original form. We should note that the developer only needs to specify which fields should be considered by the pre-processor, but not what values of them might be popular.

3.3 Cache and Encode

When a new command is received by the primary, it first pre-processes the command to extract the encodable items. If an item exists in the cache, the associated index is used to represent the encoded version of it. Instead of sending the usual ⟨Accept⟩ with the command to the backups, the primary sends an ⟨AcceptEncoded⟩ which contains the encoded items and the

data that could not be encoded. A backup decodes the items in the message by looking up the index in its local cache. As we describe in §3.4, the index of an encoded item is guaranteed to exist in the cache of the backups. Once all the items are decoded, the complete command in its original form can be generated using the RECREATE function. The backup then acts as it would with a normal ⟨ACCEPT⟩ message. The command is appended to the local log before replying to the primary with an acknowledgment. When the primary receives a majority of replies for a command, it is decided and ⟨DECIDE⟩ is sent to the backups. In UniCache, the primary and backups additionally update their cache when a new command is decided. The decided command is fed into the cache as input and depending on the eviction policy, might cause items in the cache to be evicted. Updating the cache requires using the item of the decided command for correctness. To avoid reading and processing the decided command only to update the cache, an item can be buffered until it is decided. Another approach is to update the cache eagerly already during pre-processing. This requires the cache to support rollback as it might be updated by a command that was not yet committed and dropped later.

3.4 Eviction Policy and Correctness

For correctness, UniCache must guarantee that each item encoded by the primary gets decoded as the exact same item by the backups. This requires that the cache is consistent on all servers. UniCache uses the consistency properties already provided by the replicated log to guarantee correctness. The general idea of this informal proof is simple: If each server maintains a deterministic object which handles the same sequence of inputs, then the object of each server will be consistent. This is the same idea as state machine replication [26]. In this case, the deterministic object is the cache while the same sequence of inputs is provided by the properties of the replicated log (§3.1). As UniCache uses the decided log to update the cache, it will be consistent as long as the cache eviction policy is deterministic. Furthermore since the log is guaranteed to be consistent across different views, the cache will also remain consistent in the event of a primary fail-over.

UniCache can thus use any deterministic cache eviction policy that fits the application workload. To be able to adapt to workload changes, a dynamic policy might be needed rather than a static policy such as least frequently used (LFU) or least recently used (LRU). Thus, we propose to use an ML-based policy that can learn from the history of the workload. In UniCache, we use LeCar [35], an RL-driven cache eviction policy that optimizes between using LFU and LRU. In short, LeCar maintains a probability distribution between the policies. When a cache miss occurs, LeCar uses the probability distribution to decide whether to evict based on LFU or LRU. A history of evictions labeled with the policy taken for it is kept. For every cache miss, it checks if the item was in the eviction history. If it was, a regret value associated with that policy is increased, and the weights are recalculated. This policy treats eviction as an online learning problem that aims to minimize the regret.

4 IMPLEMENTATION

To showcase UniCache in real applications, we implemented a pre-processor for two common data formats:

Comma-separated values (CSV). The pre-processor takes a set of string values from a CSV record as an input. This set is defined by the user and corresponds to the fields that can have repeated values. For each record, UniCache will check if the value of those fields exists in the cache and encode them if possible. Furthermore, the user can also implement additional processing such as split or regex to only attempt caching parts of a string.

SQL queries. The pre-processor reads a SQL query and extracts the template and parameters of it (similar to the example in §3.2). This could be applied to distributed databases that employ statement-based replication, where the log commands are SQL queries and all replicas execute them according to the log order for consistency. This is used in MySQL NDB cluster where the default `MIXED` replication format is statement-based by default [6].

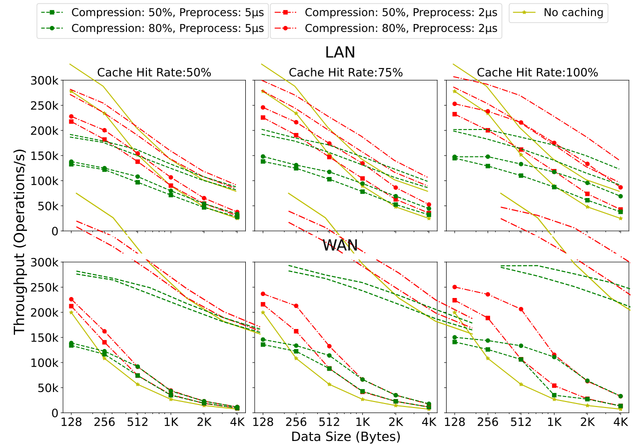

Figure 3: Synthetic benchmark in LAN and WAN.

5 EVALUATION

Our evaluation uses a synthetic benchmark (§5.1) to generalize the performance impact by UniCache, and a real-world benchmark using two CSV and one SQL dataset (§5.2).

Experimental setup. We performed the evaluation on Google Cloud, using a cluster of three c2-standard-8 instances with 8 vCPUs, 32GB of memory, and 10 Gbps network bandwidth. An additional client instance was used to propose commands and measure the end-to-end performance. The client generates a workload of 100k concurrent requests and the experiment is completed when it has received the commit acknowledgment for all records of the given dataset. Each experiment was repeated 10 times. Furthermore, we used two different network settings:

- **LAN:** all instances are in `europe-west1`. The RTT is 0.2ms.
- **WAN:** the client and the primary are in the same region `europe-west1`. The two backups are located in `us-central1` and `asia-northeast1` with RTT of 102ms and 224ms respectively. The RTT between the backups is 144ms. UniCache was implemented in OmniPaxos[2] where the log was stored in memory. The actor framework Kompact was used for message-passing over TCP.

5.1 Synthetic Benchmark

To evaluate the general performance of UniCache, we implemented a synthetic benchmark of 20 million requests that are proposed under configurations with varying data size, compression rate, cache hit rate, and pre-processing time. The data are byte vectors with the size 128-4096. The other parameters are based on typical values we found while working with the real implementations from §4. The pre-processing overhead is simulated by inducing idle time.

[2] https://github.com/haraldng/omnipaxos

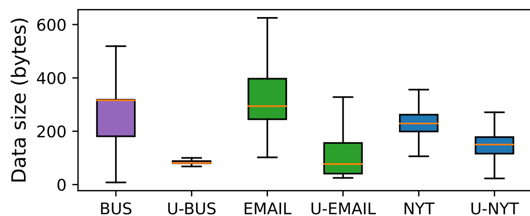

Figure 4: Data sizes before and after compression (excl. outliers). The boxes prefixed with "U-" are after compression.

What is the pre-processing overhead in UniCache? As seen in Figure 3, UniCache did not show any performance improvement with small data in the LAN setting. With low network latency and data sizes of 128 and 256 bytes, the transmission does not constitute a bottleneck. Instead, the pre-processing overhead incurred by UniCache degrades performance. Depending on the compression rate, a pre-processing time of 2μs resulted in up to 22% lower throughput. With 5μs, the corresponding result is 52%. The performance degradation is due to the pre-processing being in the critical path and executed for one command at a time by the primary. This also affects the batching. In our prototype, batching is based on time (1ms), and a longer pre-processing delay results in smaller batches.

What improvements does Unicache offer in different network environments? With data size of 512 bytes and a pre-processing time of 2μs, UniCache recorded 15% higher throughput when the cache hit rate is 75%. In the ideal case of 100% hit rate, the improvement is 42%. Thus, with data size of 512 bytes and a cache hit rate in the range of 75% or higher, the throughput can increase by 15-42%. The benefits of UniCache are more evident in larger data sizes. With 1024 bytes and 2μs pre-processing time, UniCache increases throughput by up to 96%. Furthermore, with data sizes of 2048 and 4096 bytes, UniCache is on par or outperforms baselines in all configurations, including when the pre-processing time is 5μs.

In the WAN setting (see Figure 3), UniCache outperforms the baseline implementation in all scenarios except for when the data size is 128 bytes and the pre-processing time is 5μs. We already observe an improvement by 6-25% with 128 bytes of data and pre-processing time of 2μs. Using data sizes of 256 and 512 bytes, the throughput increase is between 8-117% and 31-264% respectively. The benefits of sending compressed data over WAN become more evident as the data size gets even larger. With 1024, 2048, and 4096 bytes, UniCache improves performance by 30-354%.

From the synthetic benchmark, we can conclude that the pre-processing overhead incurred by UniCache causes performance degradation in the LAN setting with data sizes 256 bytes or lower. The reason is that the pre-processing overhead is not amortized by the transmission with small data and low network latency. However, in the WAN setting, the data transmission is a bottleneck and the pre-processing overhead is adequately amortized. As a result, UniCache recorded up to 4.5x higher throughput in WAN. Another observation we make is that, contrary to the results in LAN, the performance of UniCache in WAN is dominated by the compression rate rather than pre-processing time.

5.2 Real-world Benchmarks

We evaluated UniCache with three datasets from real applications:

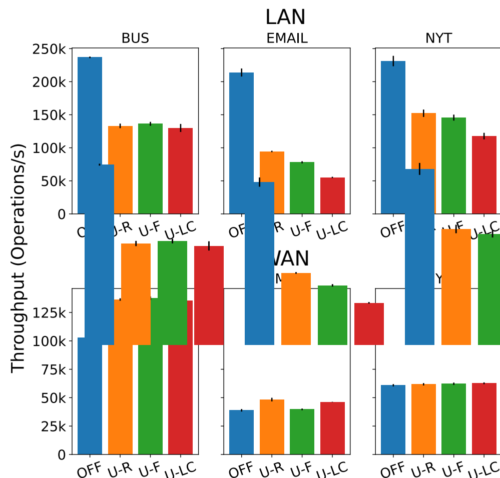

Figure 5: Real-world benchmark with different eviction policies. OFF: UniCache turned off, U-R, U-F, U-LC: UniCache with LRU, LFU, and LeCar respectively.

Enron emails [9]: CSV records of email headers and metadata. UniCache reads and attempts to cache the fields such as folder names, sender and receiver addresses, and words in the subject.
New York Times articles [7]: The metadata of all New York Times articles between 2020 and 2022 in CSV format. The fields that get pre-processed by UniCache are the web url, headlines, authors, and article metadata (e.g., category, tags, keywords).
BusTracker [31]: SQL queries from a mobile application that updates bus location information, finds users nearby bus stops, and gets route information. UniCache extracts the query template using regex operations and encodes any template that is cached. The idea is similar to using prepared statements, but without requiring the developer to know the queries a priori.

The average pre-processing time for the email records was 8μs which resulted in an average compression of 66%. The same compression rate was recorded for the bustracker queries, but with a lower execution time of 4μs. The pre-processing time is higher for the email records as they require more processing such as splitting by whitespace on the subject field to cache individual words. The pre-processing time in the articles dataset was 4μs and the compression rate was 38%. The lower compression rate is due to the wide range of words that can be used for titles and tags in news articles. The distribution of data sizes before and after compression can be seen in Figure 4. In all datasets, the cache used less than 500kb memory.

As seen in Figure 5, UniCache recorded a performance decrease of more than 35% in the LAN for all datasets. This aligns with the findings from the synthetic benchmark. The transmission of data smaller than 512 bytes is not a bottleneck in LAN and the pre-processing overhead reduces performance instead. In the WAN setting, the high compression rates in the email and bustracker datasets yielded performance improvements by UniCache. With the LRU eviction policy in the email dataset, 24% higher throughput was recorded. The most improvement was found in the bustracker dataset where UniCache recorded an improvement of 33%. This is because queries would consistently be compressed to a high degree (see Figure 4). Lastly, the lower compression rate in the articles dataset resulted in a similar performance as the baseline.

In general, the results using real-world data reflect the findings from the synthetic benchmark. A high compression rate can significantly improve performance in WAN where the latency is high. Whereas in the LAN, the network transmission is not a bottleneck and leads to non-amortized compression overhead.

6 DISCUSSION AND PROSPECTS

When is UniCache beneficial? UniCache can evidently provide good I/O savings for application data of varying skew and wide value range and granularity. We foresee its highest potential within multi-datacenter replicated cloud services where cross-region traffic dominates the performance and operational costs. In such settings, UniCache can be beneficial even when the compression rate is high, but not enough to boost performance (e.g., NYT articles dataset in §5.2).

Parallel Pre-processing. The current prototype of UniCache employs a naive sequential pre-processing logic. However, common parallelization methods can be used to minimize its critical path footprint. Deployments with persistent storage enforce flushing data before proceeding in the protocol for safety reasons. UniCache could employ its pre-processing during this flushing phase instead. Pipelining is also feasible by chaining the pre-processing of consecutive commands without violating the determinism of the cache.

Auto-tunable UniCache. As shown by the synthetic benchmark, the overhead of pre-processing can bring performance penalties when there is a high rate of cache misses. To that end, a possible improvement would be to dynamically enable and disable UniCache according to a threshold of cache hit rate. In this way, the replicated state machine middleware can tune itself to optimize for better throughput based on its measured hit rate.

BFT UniCache. UniCache could potentially be applied to Byzantine fault-tolerant protocols such as PBFT [18] that also guarantees safety via a consistent log. BFT protocols typically require not just the primary, but all replicas to transmit application data, and UniCache could thus reduce the network I/O for more servers. However, we see two practical challenges: 1) The application domain of BFT protocols often focuses on unique data (nonces, hashes, etc.) that cannot be cached, and 2) BFT protocols have a longer commit path with more message round-trips that might neglect the effects of UniCache. Nonetheless, BFT UniCache remains an interesting prospect for future work.

7 RELATED WORK

Log replication protocols such as Raft [32], VR [29], Zab [25], and Paxos [28] derivatives are prevalent in both academia and industry. The primary bottleneck in these protocols is a well-studied problem: Fast Paxos [27] relies on the client to disseminate the data which pushes the network I/O to the client instead of the primary. Fault-tolerance is also reduced as larger quorums are required for safety. Custom transmission schemes [38] and network overlays incur higher commit latency and decrease fault-tolerance as any failures along the dissemination path will affect progress. Erasure coding [36] avoids the increased latency but leads to backups having incomplete state which affects practicality for applications. Contrary to UniCache, these approaches cannot be generalized as the underlying system execution and normal behavior are altered.

UniCache aligns with the recent trend of using data- and application workload-driven optimized replication. Skyros [21] exploits the application API to reduce commit latency by deferring the replication of operations that do not need to be materialized instantly. ResilientDB [22] considers the given network setting to reduce cross-region traffic by using topology-aware replication with PBFT [18]. Another technique employed by CAPER [15] and SharPer [16] is sharded replication for different applications and objects which enables more parallelism, and could also be used in conjunction with UniCache to boost performance further.

8 CONCLUSION

UniCache is a transparent approach to tackle the primary bottleneck problem in log replication protocols. It uses a universally consistent cache at each server that is used for lossless compression of frequently replicated data. This reduces the data transmission in the protocol without making trade-offs in fault-tolerance or practicality as in other approaches. As shown in the evaluation, UniCache can achieve significant performance improvements, especially in high latency environments such as WAN.

ACKNOWLEDGMENTS

This work has been supported by the Swedish Foundation of Strategic Research (Grant No.: BD15-0006), RISE AI, Google Cloud Research Credits Program, and the Wallenberg AI: Autonomous Systems and Software Program (Data-Bound NEST Project).

REFERENCES

[1] 2022. *Apache Flink.* Retrieved 2022-09-11 from https://flink.apache.org

[2] 2022. *Apache Kafka.* Retrieved 2022-09-11 from https://kafka.apache.org

[3] 2022. *Apache Spark.* Retrieved 2022-09-11 from https://spark.apache.org

[4] 2022. *Dragonboat - A Multi-Group Raft library in Go.* Retrieved 2022-12-15 from https://github.com/lni/dragonboat

[5] 2022. *MongoDB Performance Scale.* Retrieved 2022-08-01 from https://www.mongodb.com/mongodb-scale

[6] 2022. *MySQL 8.0 Reference Manual: Replication Formats.* Retrieved 2022-09-27 from https://dev.mysql.com/doc/refman/8.0/en/replication-formats.html

[7] 2022. *New York Times Articles Metadata 2020-2022.* Retrieved 2022-12-15 from https://www.kaggle.com/datasets/cascaschatz/new-york-times-articles-metadata-2020-2022

[8] 2022. *RethinkDB.* Retrieved 2022-09-11 from https://rethinkdb.com

[9] 2022. *Structured Enron Dataset.* Retrieved 2022-12-15 from https://www.kaggle.com/datasets/rcmonteiro/structured-enron-dataset

[10] 2022. *TigerBeetle.* Retrieved 2022-12-15 from https://tigerbeetle.com/

[11] 2022. *TiKV.* Retrieved 2022-12-15 from https://tikv.org/

[12] 2022. *TiKV Adopters.* Retrieved 2022-08-01 from https://tikv.org/adopters/

[13] 2022. *Typical command sizes in log replication systems.* Retrieved 2022-12-23 from https://gist.github.com/haraldng/09b31a612506e69691cf96cffcaf4e08

[14] 2022. *yugabyteDB Raft replication.* Retrieved 2022-07-29 from https://docs.yugabyte.com/preview/architecture/docdb-replication/replication/#raft-replication

[15] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2019. CAPER: A Cross-Application Permissioned Blockchain. *Proc. VLDB Endow.* 12, 11 (jul 2019), 1385–1398. https://doi.org/10.14778/3342263.3342275

[16] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2021. SharPer: Sharding Permissioned Blockchains Over Network Clusters. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21).* Association for Computing Machinery, New York, NY, USA, 76–88. https://doi.org/10.1145/3448016.3452807

[17] Mike Burrows. 2006. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation.* 335–350.

[18] Miguel Castro, Barbara Liskov, et al. 1999. Practical byzantine fault tolerance. In *OsDI*, Vol. 99. 173–186.

[19] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.

[20] etcd 2022. *etcd.* Retrieved 2022-12-15 from https://etcd.io/

[21] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2021. Exploiting nil-externality for fast replicated storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles.* 440–456.

[22] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. 2020. ResilientDB: Global Scale Resilient Blockchain Fabric. 13, 6 (mar 2020), 868–883. https://doi.org/10.14778/3380750.3380757

[23] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.
[24] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems.. In *USENIX annual technical conference*, Vol. 8.
[25] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. 2011. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*. 245–256. https://doi.org/10.1109/DSN.2011.5958223
[26] Leslie Lamport. 1984. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 6, 2 (1984), 254–280.
[27] Leslie Lamport. 2006. Fast paxos. *Distributed Computing* 19, 2 (2006), 79–103.
[28] Leslie Lamport et al. 2001. Paxos made simple. *ACM Sigact News* 32, 4 (2001), 18–25.
[29] Barbara Liskov and James Cowling. 2012. Viewstamped replication revisited. (2012).
[30] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. 2018. Query-Based Workload Forecasting for Self-Driving Database Management Systems. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 631–645. https://doi.org/10.1145/3183713.3196908
[31] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J Gordon. 2018. Query-based workload forecasting for self-driving database management systems. In *Proceedings of the 2018 International Conference on Management of Data*. 631–645.
[32] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 305–319. https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro
[33] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. 2014. E-Store: Fine-Grained Elastic Partitioning for Distributed Transaction Processing Systems. *Proc. VLDB Endow.* 8, 3 (nov 2014), 245–256. https://doi.org/10.14778/2735508.2735514
[34] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. 2020. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1493–1509.
[35] Giuseppe Vietri, Liana V. Rodriguez, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. [n.d.]. Driving Cache Replacement with ML-based LeCaR, booktitle = 10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18), year = 2018, address = Boston, MA, url = https://www.usenix.org/conference/hotstorage18/presentation/vietri, publisher = USENIX Association, month = jul,.
[36] Zizhong Wang, Tongliang Li, Haixia Wang, Airan Shao, Yunren Bai, Shengming Cai, Zihan Xu, and Dongsheng Wang. 2020. CRaft: An Erasure-coding-supported Version of Raft for Reducing Storage Cost and Network Cost. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 297–308. https://www.usenix.org/conference/fast20/presentation/wang-zizhong
[37] Michael Whittaker, Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, Neil Giridharan, Joseph M Hellerstein, Heidi Howard, Ion Stoica, and Adriana Szekeres. 2020. Scaling Replicated State Machines with Compartmentalization [Technical Report]. *arXiv preprint arXiv:2012.15762* (2020).
[38] Siyuan Zhou and Shuai Mu. 2021. Fault-Tolerant Replication with Pull-Based Consensus in MongoDB. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 687–703. https://www.usenix.org/conference/nsdi21/presentation/zhou

A PSEUDO CODE

Interface and pseudo code for UniCache. Modifications to the log replication protocol are marked in blue.

```
/* Interface
1:  Fun PreProcess(C) → (List⟨data⟩, List⟨data⟩)
2:  Fun Recreate(List⟨data⟩, List⟨data⟩) → C
/* Primary Code
3:  Upon ⟨Propose | C⟩
     s.t. state = (PRIMARY, ACCEPT)  from client
4:    log ← log ⊕ C;
5:    accepted[self] ← |log|;
6:    items ← [];
7:    (encodable, not_encodable) ← PreProcess(C);
      foreach e ∈ encodable do
8:      if SOME⟨index⟩ = TryEncode(e) then
9:        items ← items ⊕ ENCODED⟨index⟩;
        else
10:       items ← items ⊕ DECODED⟨e⟩;
      foreach b ∈ promised_backups do
11:     send ⟨AcceptEncoded | n, items, not_encodable⟩ to b;
/* Backup Code
12: Upon ⟨AcceptEncoded | n, items, not_encoded⟩ from p
     s.t. state = (Backup, ACCEPT)
13:   if n_promised = n then
14:     decoded ← [];
        foreach i ∈ items do
15:       if ENCODED⟨index⟩ = i then
16:         data ← Decode(index);
17:         decoded ← decoded ⊕ data;
18:       if DECODED⟨data⟩ = i then
19:         decoded ← decoded ⊕ data;
20:     C ← Recreate(decoded, not_encoded);
21:     log ← log ⊕ C;
22:     idx ← |log|;
23:     send ⟨Accepted | n, idx⟩ to p;
/* Both Primary and Backup Code
24: Upon ⟨Decide | n, idx⟩ s.t. n_promised = n
25:   idx_decided ← idx;
26:   UpdateCache(idx)
```