RESEARCH-ARTICLE

Asynchronous Replication Strategies for a Real-Time DBMS

V SRINIVASAN

ANDREW GOODING

SUNIL SAYYAPARAJU

THOMAS LOPATIC

KEVIN PORTER

ASHISH KRISHNADEO SHINDE

View all

Published: 22 June 2025

Citation in BibTeX format

SIGMOD/PODS '25: International Conference on Management of Data
June 22 - 27, 2025
Berlin, Germany

Conference Sponsors:
SIGMOD

SIGMOD/PODS '25: Companion of the 2025 International Conference on Management of Data (June 2025)
https://doi.org/10.1145/3722212.3724429
ISBN: 9798400715648

Asynchronous Replication Strategies for a Real-Time DBMS

V. Srinivasan
srini@aerospike.com
Aerospike
Mountain View, CA, USA

Andrew Gooding
andy@aerospike.com
Aerospike
Mountain View, CA, USA

Sunil Sayyaparaju
sunil@aerospike.com
Aerospike
Bengaluru, India

Thomas Lopatic
thomas@aerospike.com
Aerospike
Berlin, Germany

Kevin Porter
kporter@aerospike.com
Aerospike
Mountain View, CA, USA

Ashish Krishnadeo Shinde
ashish@aerospike.com
Aerospike
Bengaluru, India

Sri Varun Poluri
varun@aerospike.com
Aerospike
Hyderabad, India

B. Narendran
bnarendran@aerospike.com
Aerospike
Warren, NJ, USA

Daudkhan Pathan
dpathan@aerospike.com
Aerospike
Pune, India

Srinivasan Seshadri
sseshadri@aerospike.com
Aerospike
Mountain View, CA, USA

Abstract

Real-time mission-critical services that handle large-scale consumer facing applications typically use a real-time DBMS that can provide both high performance as well as high availability. The only sure way to ensure that such services can survive catastrophic infrastructure failures is to deploy the DBMS as well as the applications themselves on multiple sites that are hundreds (or even thousands) of miles apart from each other.

Although synchronous replication across sites can be used for some consumer services, most high-performance consumer applications cannot tolerate the high write latencies required for synchronizing writes across far-apart sites. Therefore, the use of asynchronous replication between geographically distributed sites becomes the only practical option for these applications. This means that any changes in the data must be replicated quickly across sites to keep the exposure window as low as possible in the case of a major site failure event.

In this paper, we describe a resource-optimized, asynchronous replication strategy called XDR that we developed for use with a real-time DBMS (Aerospike) that supports very high read and write throughput. XDR is resource optimized because it is co-designed within the same process as the core database. This enables replication between sites to proceed at a high throughput despite the high latency between sites that are thousands of miles apart, while not impacting the latency and throughput of the source transactions. Furthermore, this co-design allows for efficient implementation of several other features within XDR such as i) filtering data during replication and ii) allowing multiple sites to update the same record simultaneously while ensuring the sites converge on the content of the records quickly. XDR has been used to deploy more than 100 large-scale customer applications worldwide.

CCS Concepts

• **Information systems → Parallel and distributed DBMSs**.

Keywords

Asynchronous replication, Change Data Capture, Disaster Recovery, Hot Standby, Real Time DBMS

ACM Reference Format:
V. Srinivasan, Andrew Gooding, Sunil Sayyaparaju, Thomas Lopatic, Kevin Porter, Ashish Krishnadeo Shinde, Sri Varun Poluri, B. Narendran, Daudkhan Pathan, and Srinivasan Seshadri. 2025. Asynchronous Replication Strategies for a Real-Time DBMS. In *Companion of the 2025 International Conference on Management of Data (SIGMOD-Companion '25), June 22–27, 2025, Berlin, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3722212.3724429

1 Introduction

Geographically distributed deployments are widely used for real-time services that serve a large number of consumers, from millions to billions. These services require stringent Service Level Agreements to ensure both low, predictable latency and round-the-clock availability to support a seamless, real-time user experience. The goal is to deliver the same high-quality experience to both the first and billionth customer while ensuring uninterrupted service. Replicating data across geographically dispersed sites is a key design pattern in mission-critical, real-time services, such as financial systems, e-commerce platforms, telecommunications, and ticket reservation systems.

SIGMOD-Companion '25, Berlin, Germany

ACM ISBN 979-8-4007-1564-8/2025/06
https://doi.org/10.1145/3722212.3724429

In this paper, we describe the design of Aerospike's asynchronous replication, known as XDR[1]. The Aerospike database system [19–22] scales to millions of transactions a second in a cluster even while guaranteeing a P99 latency of single-digit milliseconds. Our goal is to replicate data from a source cluster to a destination cluster which are geographically far apart. The round-trip network latency between such distant clusters is typically between 70 and 200 milliseconds. ***The design goal for XDR was to minimize the lag between the records in the two clusters without impacting the throughput or latency of the transactions at the source cluster.*** Note that, synchronous replication, by contrast, would impact the write latency of transactions on a source cluster whose nodes (or racks) are stretched across geographically distant sites, as multiple replicas must be updated before confirming the write to the application[2].

There are several use cases that benefit from an asynchronous replication architecture. The first is disaster recovery that enables consumer-facing internet or mobile services to stay up during catastrophic failures. The second is the usage of different clusters at the edge and core of a digital infrastructure. This allows consumer applications and their respective data to be located close to the end user location to provide the best possible user experience. Finally, asynchronous replication is often used to backup (both a full backup and an incremental backup), to get a point-in-time-recovery (PITR) solution for recovering from data corruption or just to create another cluster similar to the existing cluster for production use or for testing.

To illustrate the benefits of asynchronous replication in a real-world scenario, consider fraud detection in payment systems [8], where machine learning and AI algorithms are applied to generate fraud scores in real time (within approximately 100 ms). Hundreds of reads, several of them in parallel, from the database are typically required to calculate a fraud score, along with dozens of writes to record recent transaction history. For fraud prevention, the AI model must have access to recent behavioral data to enable timely action, making low-latency reads and writes essential. Use cases requiring similar support for real-time data access span areas such as risk management, recommendation engines, and real-time bidding for advertisements. While these applications require efficient access to recent data, they can generally tolerate slightly outdated information during failures. Consequently, relaxing consistency requirements is acceptable in favor of reduced latency, higher throughput, and robust availability, making asynchronous replication the preferred approach in these contexts.

Asynchronous replication in database systems is well-established in relational databases and more recently in NoSQL and NewSQL systems (we discuss this in detail in Section 9). Typically, these systems employ some form of operation logging on the source cluster, with logs then replayed on the destination cluster. This method is reliable and ensures that all modifications on the source are transmitted in a consistent, ordered manner to the destination cluster. Operational logging requires lot of resources, at the minimum it requires writing the logs to disk at each local node, then another process reads in the logs and then transmits the log to the destination.

In order to minimize the lag, we had to significantly reduce the resource consumption (in terms of CPU, memory, disk, and network) associated with logging each operation individually[3] and transmitting each one to the destination cluster. As described above, the Aerospike database is able to achieve very high throughput while keeping latencies sub-ms and this is largely an artifact of having been engineered and built in a very resource efficient manner. XDR builds on the core database by tightly coupling itself to the core database - it is co-designed to be within the same process as the database. This co-design approach eliminates many redundant data transfers between operating system processes and reduces unnecessary memory and disk copying. As an aside, because Aerospike is a NoSQL database, most of our applications have carefully modeled their data and de-normalized to ensure all operations are on a single record. Aerospike supports atomic transactions that can perform any number of operations on a record and XDR as described in this paper applies to such Single Record Transactions. Aerospike is currently building out capabilities for MRT (Multi Record Transactions) and XDR will evolve to support MRT in the future.

Based on our design goal of minimizing lag between the source and the destination cluster, we do not have to transmit every change to a record that is getting updated rapidly (a hotkey). We could skip intermediate updates if a more recent change was available when preparing updates for transmission. This approach is sufficient to ensure the lag at the destination cluster is not high. We allow the degree to which we are skipping over intermediate updates to be configurable and flexible, all the way to not skipping any intermediate update.

In addition, XDR supports a variety of other features that are critical in an enterprise setting. All of these features also are resource efficient as a result of the above co-design approach.

(1) XDR supports a range of filtering options, allowing specific fields of specific records to be selectively replicated to the destination. These filters are applied within the database process as early as possible, leading to optimized resource utilization compared to operation logging.
(2) XDR allows both source and destination clusters to be active simultaneously. They could be simultaneously updating mutually exclusive fields or even be updating the same field. XDR has built in mechanisms to ensure non-conflicting changes are merged and in the event of conflicting changes, exactly one of the changes prevails in both clusters.
(3) XDR is not limited to replication across Aerospike clusters. Aerospike has outbound connectors that relay changes to the records that XDR is transmitting to other databases and message queues (e.g., Kafka).
(4) XDR has a feature called Rewind which allows XDR to selectively transmit records that have changed after a specified time. This feature enables incremental backup of a source

[1]XDR was initially named to represent cross(**X**) **D**ata center **R**eplication. However, with the widespread adoption of cloud environments, XDR is also used as the replication system across cloud regions or cloud availability zones.

[2]Our synchronous replication algorithm for achieving strong consistency is discussed in detail in [22]. This approach is utilized by customers who prioritize consistency over availability and are willing to tolerate increased write latencies (up to 200ms instead of sub-ms) to guarantee consistency.

[3]Our first implementation of Asynchronous Replication used some form of logging (to track which records had changed) which we abandoned in favor of XDR described in this paper.

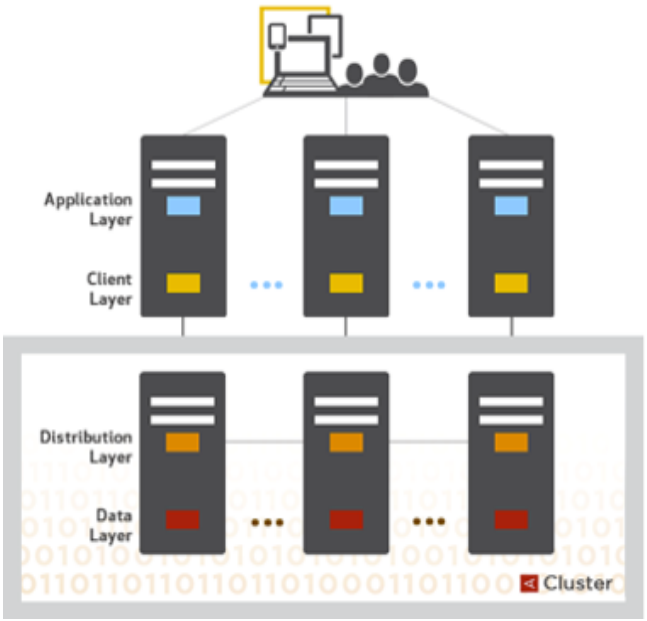

Figure 1: Aerospike Cluster Architecture

cluster over time. It can also be used for duplicating an existing cluster by setting up a new cluster and transferring all data from the existing cluster to the new one without any interruption in service.

The rest of this paper is organized as follows: Section 2 is a short introduction to Aerospike with a particular focus on what is relevant to XDR. Section 3 outlines the different replication topologies used in Aerospike deployments. Section 4 describes the core design of XDR. Section 5 outlines several advanced features of XDR. Section 6 outlines how XDR is used for several other applications. Section 7 talks to the correctness guarantees of XDR. We present some performance results in Section 8. We discuss related work at the end after discussing XDR in Section 9 and conclude in Section 10.

2 Aerospike Cluster Architecture

The Aerospike database cluster [22] will be used as the building block for the asynchronous replication strategies described in this paper. The Aerospike real-time database cluster architecture is illustrated in Figure 1 and we highlight a few select characteristics below relevant to this paper.

- A shared nothing system where every node is identical.
- The records in Aerospike belong to container units called namespaces[4]. All further description is about data and records in a namespace.
- A hash-based data partitioning scheme that avoids hotspots.
- An intelligent client is aware of the data placement within the cluster and thus providing one-hop data access to the data.
- A primary index is used to keep track of the physical location of a record for each key.
- We will assume synchronous replication of the data within a cluster throughout the paper so we can understand the guarantees provided by XDR clearly. Note asynchronous replication within a cluster provides few guarantees within the cluster to begin with.

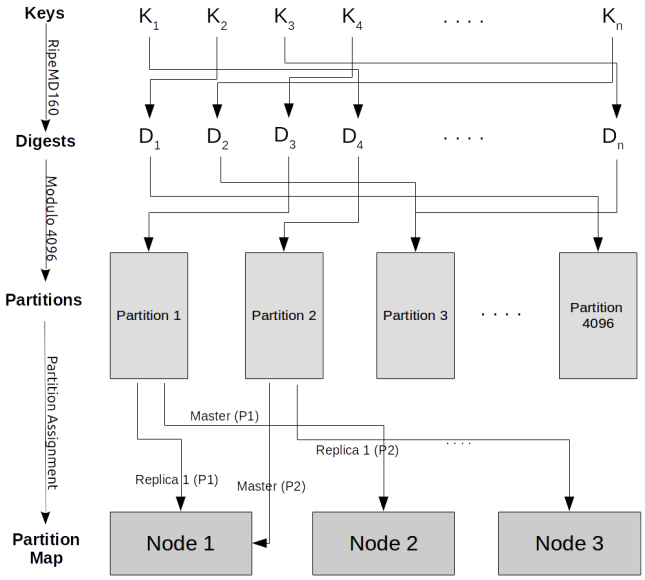

Figure 2: Data Partitioning

There are a few important concepts that are helpful to understand a bit deeper to appreciate the details of XDR. They are described below.

2.1 Data Distribution

Aerospike uses a data partitioning scheme that distributes data across nodes in a uniform manner (Figure 2). A record's primary key is hashed into a 160-bit digest using the RIPEMD algorithm, which is extremely robust against collisions [11]. The digest space is partitioned into 4096 non- overlapping 'partitions. A partition is the primary unit of data distribution. Records are assigned to partitions based on the primary key's hash digest. Even if the distribution of keys in the key space is skewed, the distribution of keys in the digest space and therefore in the partition space is uniform.

Partitions are assigned to nodes at random using Rendezvous Hashing [25]. Figure 3a shows the partition assignment for a 5-node cluster with a replication factor of 3. The partition assignment is obtained as follows:

(1) For each partition, and for each node, the hash value[5] of the combination of partition-id and node-id is computed.
(2) For each partition, the hash values computed above are sorted to obtain a sorted node list - this corresponds to a row in Figure 3a.

[4]These are similar to table spaces and there are containers within namespaces called sets which are similar to tables.

[5]Any good hash function can be used.

Partition	Master	Replica 1	Replica 2	Unused	Unused
P1	N5	N1	N3	N2	N4
P2	N2	N4	N5	N3	N1
P3	N1	N3	N2	N5	N4
....	....	....	....	....	....

(a) Partition assignment with replication factor 3

P2	N2	N4	N3	N1	

(b) P2 succession list when N5 goes down

P2	N2	N4	N5	N3	N1

(c) P2 succession list when N5 comes up again

Figure 3: Mapping Partitions to Nodes

(3) For our case of replication factor of three, Aerospike designates one node as a master and the rest as replicas. The first node in the sorted list contains the master partition and the next two nodes in the sorted list have the replica partitions. Only the first three columns in the partition map are used; the last two columns are unused.

The most important aspect of the partition map is in how the map may change when nodes are added or removed from a cluster. We will briefly describe this now. Consider the case where the node N5 goes down. Let us consider partition P2 for which N5 hosts a copy of the partition. N5 is removed from P2's sorted node list causing a left shift for all subsequent nodes as shown in Figure 3(b). As a resilt of the left shift, N3 would take N5's place and records in P2 will be migrated (copied) to N3. Once N5 returns and becomes part of the cluster again, it would simply regain its position in each partition's sorted node list, as shown in Figure 3(c) (for partition P2). If N5 did not host a copy of a partition (e.g., P3), this partition would not require data migration. Adding a brand-new node to the cluster would have the effect of inserting this node at some position in each partition's sorted node lists, and, therefore, result in the right shift of the subsequent nodes for each partition. Assignments to the left of the new node are unaffected.

2.2 Primary Index

The primary index is used to keep track of the physical location of a record given its key. The primary index is implemented as an R-B Tree and there is one such tree for each of the 4096 partitions in the key space. One of the most popular Aerospike deployments is when the primary index is in memory while the data is on disk. As an aside, note that the primary index is far more compact than typical records and given that many instances in the cloud as well as in data centers have typical memory to disk ratios of at least 1:50, it turns out that this is a viable architecture for a large class of applications. However, the primary index can also be on disk but many of the design decisions piggy back on the speed of index operations when it is memory resident.

As a normal part of write transactions, Aerospike keeps a last update time (LUT) with the record. The primary index of Aerospike also contains the LUT of the data items as well as the digest and can be used to search for all data items that have an LUT greater than a certain specified time. Such a query can be used to efficiently retrieve the list of the digests of all the records that have been changed after a specified time. One important detail about LUTs is that we do not require globally synchronized clocks within a cluster or across clusters. LUTs are maintained local to a node but our strong consistency algorithms make sure that LUTs on the versions of a record are monotonic when writing a record. In particular, if a replica with a clock that is behind a failed master takes over a partition, it will ensure that any subsequent write to a record within the partition always pushes the LUT forward as it has the LUT of the older version by virtue of strong consistency.

With this background about the basics of an Aerospike cluster, we can now proceed to describe the strategies we have developed for asynchronous replication across sites that are far apart geographically.

3 Replication Architecture

In this paper, we focus on inter-cluster replication. As outlined in [22], we have implemented both synchronous and asynchronous replication to achieve intra-cluster replication. We will see shortly that intra-cluster replication strategy is not a good base on which to build the inter-cluster replication strategies.The goal of intra-cluster replication is to achieve fault-tolerance to node failures within the cluster. As a result, these protocols heavily depend on frequent heartbeat messages to know the state of all nodes in a cluster at all points of time. In contrast, the goal of inter-cluster replication is to leave the operational independence of a cluster intact. In fact, two different departments or organizations may be managing or operating the source and the destination cluster respectively in an enterprise. It is definitely not practical in these situations to intertwine the operational fate of the two clusters. As a result, in our inter-cluster replication, the nodes of the source cluster act as a client to the destination cluster and leave the clusters operationally independent. Figure 4 demonstrates two clusters that are using synchronous replication within the clusters and asynchronous replication across clusters.

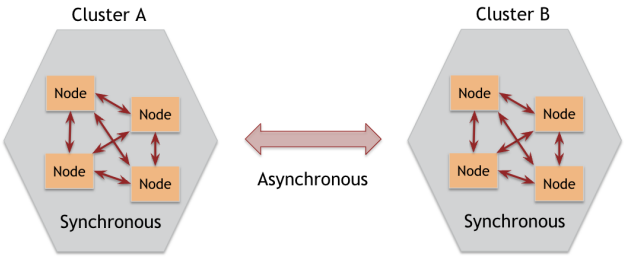

Figure 4: Inter-Cluster Replication Is Asynchronous

Our system supports both active-passive and active-active replication described below:

- **Active Passive Replication:** In Figure 5, the active cluster, A, accepts application writes, and sends these to the passive cluster, B, where it's applied locally. The active-passive behavior is indicated by the one-way arrows linking the clusters. There are no application writes directly written to

Figure 5: Active-Passive Replication

Figure 6: Active-Active Replication

cluster B. Applications can read from the passive cluster, but the latest data will not be available in the passive cluster until the writes to the active cluster have been received and applied to the passive database cluster B. The time it takes for a committed write in the active cluster to arrive at the passive cluster is called the lag.

- **Active Active Replication:** In an active-active setup, both the clusters in the system can be accepting application writes and shipping these to each other. A simple two cluster active-active system is shown in Figure 6. In this case, there are two clusters, A and B. Both these clusters are taking writes from the application, and they ship their data writes to the other cluster. Note that if the same data item is modified around the same time in both clusters, there is a potential for data conflicts to occur and our XDR has mechanisms to ensure the same write wins in both the clusters eventually.

In real-life situations, replication topologies can be quite complex involving multiple clusters. Figure 7 shows a three node chain of clusters where the updates are forwarded along the chain. This results in a cumulative delay to ship data to clusters that are many hops from the active cluster. Figure 8 shows an example of a star topology which has one active cluster A that ships all its writes in parallel to the other clusters. A minimum of three clusters are required to create a star topology, one active cluster (A) and at least two passive clusters (e.g., D and E).

Figure 9 illustrates an example where there are four active clusters (A, B, C, and D). Each of these clusters are connected to the three other clusters with bi-directional links. We refer to this as a mesh topology. We have found star and mesh topologies to be the most used in complex real-world deployments. Additionally, there is a link from the active cluster D to a passive cluster E. Each of the active clusters (A, B, C, and D) allow application writes while the passive cluster E only gets changes from cluster D through the one-way replication link. Since D is connected to all other active clusters, E should eventually receive all the application data writes that occur at any of the active clusters.

Figure 7: Linear Chained Active-Passive

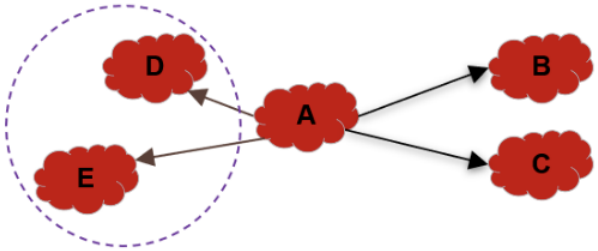

Figure 8: Star Topology

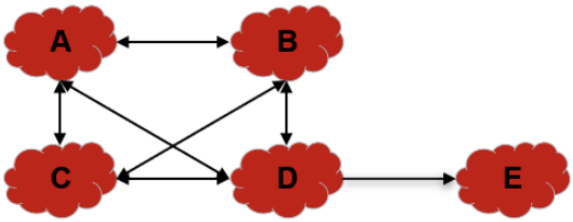

Figure 9: Complex Replication Topology

4 XDR Design

XDR is designed as an integral part of the Aerospike database process on each node. Each namespace can be configured to be replicated to any number of destinations. Different destinations could have widely different network connectivity and different response time characteristics. This will lead to different lags for each of these destinations and it is best that the operation of XDR for a given namespace and destination is completely independent of other namespace and destination combinations. We focus on a particular namespace and destination for the rest of the paper. The destination need not be an Aerospike cluster (or even a cluster for that matter) but we will describe the main algorithm for the case where the destination is an Aerospike cluster (this helps the co-design extract the maximum benefit).

An Aerospike client instance is used by XDR (XDR client henceforth) to transfer data to the nodes of the remote cluster. The XDR write to the remote cluster is identical to an application write to that cluster except for one extra bit identifying that the write is from an XDR client. Thus, XDR piggy backs on all the functionality of the Aerospike client-server protocols in communicating with the remote cluster. This technique of using the vanilla client for asynchronous replication has a key benefit in terms of node and cluster configuration. Individual clusters in a replication topology are decoupled. Therefore, the source cluster can have nodes with different capabilities as well as a different number of nodes than

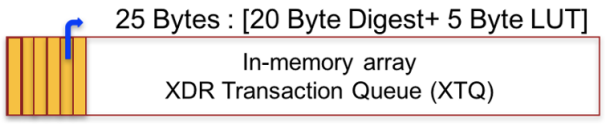

Figure 10: XDR Transaction Queue: XTQ

the destination cluster and vice versa. This allows for extremely flexible deployment configurations in the real-world and the connected clusters in the same deployment can be deployed in widely different environments, e.g. bare metal data centers, private clouds, or public clouds. This ensures that these two clusters are completely operationally independent.

Recall that the unit of data distribution and replication is a partition. As a result, one of the key design decisions is that XDR design is also partition-centric - this means that we can focus on what XDR needs to do to replicate records in a partition. There are very few places the partitions interact with each other from XDR logic perspective and we will point out when such an interaction is required.

Aerospike database process has several threads, some of which are dedicated to specific special-purpose tasks but the main work is performed by a pool of service threads. The writes from a client are typically processed by a service thread and the XDR writes to destination are also performed by service threads. When the destination is an Aerospike cluster, one specific service thread takes the responsibility of shipping changes to records in a partition and establishes a connection with the unique destination node that is the current master node for this partition. The network latency between the XDR client and the destination server nodes are large (in the tens to hundreds of milliseconds). Therefore, XDR write requests are pipelined on each connection. Further, there could be several other connections from this source node (corresponding to other partitions at this source node) established by other service threads to the same destination node. The pipelining within a partition and parallelism across partitions enables XDR to minimize lag even if the latency between the source and the destination cluster is high.

4.1 XDR: Normal Operations

We will now describe how XDR operates under normal conditions (no failures anywhere in the source or the destination cluster or the network connecting them). As mentioned above, our description will focus on a single partition (and a namespace, destination combination).

XDR maintains an in-memory FIFO queue called the XDR Transaction Queue (XTQ henceforth). An incoming write (either from a regular client or an XDR client for which this node is the destination node) is handled by a service thread inside the Aerospike process. For each write, the service threads adds the record's digest and LUT to the tail of XTQ as shown in Figure 10. The entries are dequeued by a special thread in the system called the DC thread. The DC thread periodically (typically every 100ms) picks a group of entries (upto a maximum based on how many are in the queue) from the head of XTQ (oldest LUTs) and schedules them to be processed by the service thread responsible for shipping this partition. The DC thread then goes to sleep for the next period (as mentioned before typically 100ms). That service thread reads the group of records from storage and starts sending these to the destination node in a pipelined manner (this is the reason for reading a group of records and then sending them one after another without waiting for acknowledgements). Under normal circumstances, all of these records are successfully shipped and when the DC thread wakes up it continues with the next group.

As mentioned in Section 1, recall that one of the key conceptual ways we are able to keep the lag between the source and the destination cluster small is by not insisting that every update to a record is transmitted to the destination. In practice, several of our hotkeys are updated several times during the order of hundred milliseconds. One optimization for hotkeys is that we do not add a digest to XTQ if that digest is already in XTQ and its LUT is within 100ms of the current record's LUT. As should be clear from the above description, when this digest does get picked up by the DC thread and a read scheduled, the latest version will automatically be read and previous versions would not have been shipped.

4.2 XDR: Node Failures in Source Cluster

We will now describe what additions we need to make to the XDR system to gracefully and efficiently handle node failures in the source cluster. First, the nodes that have a replica of a partition will receive replica writes through the strong consistency protocol within these cluster. Based on these replica writes, the node will also maintain a XTQ for those partitions to enable quick take over of XDR transmission if the master node fails. However, we need to decide how and when to purge the XTQ as well as decide from what point to start shipping to the destination if the master fails. To make this process efficient, we introduce the notion of LST (last ship time) of a partition. LST satisfies the invariant that all records with a LUT smaller than the LST have already been shipped to the destination. The DC thread updates LST appropriately based on which records have been successfully shipped as part of each group it processes during a period.

The last ship time, LST, is sent every two seconds from the node with the master copy of the partition to the replica nodes. First, this enables the XTQ at the replica node to purge entries whose LUT is smaller than the LST. Second, when the node owning the master partition goes down, the replica will take over immediately but may re-ship some updates due to the lag in LST between master and replica nodes. When the original node comes back, the replica that is acting as master will start sending LST updates every two seconds to the original master until the original master has caught up and can safely take over as the master again. If multiple nodes become disconnected from the cluster, it is possible for a partition to become unavailable (our cluster is supposed to be strongly consistent), even though other partitions continue to ship. When a partition becomes unavailable, it is possible that the LST being shared between the master and replica nodes for that partition is completely lost (if all nodes with replicas go down). In thus case, the system will have to revert to the cluster-wide global LST (this is the earliest of all the LST values maintained per partition for this namespace, destination cluster pair), which is also persisted to disk periodically. The global last ship time will stop advancing when the cluster has

any partitions unavailable. This is to ensure that when the missing nodes join the cluster, they can restart shipping from the global LST value (local values may have been lost for this partition). A lot of re-shipping may happen, but the system ensures that it never misses shipping a previously unshipped record.

4.3 XDR: Failure to Communicate with Destination Node

There are several reasons due to which a record shipped from the source node may not be acknowledged back from the destination node. There could be a problem with the network, the destination node, or the write to this record or partition at the destination node. Usually, many of these are very transient failures (of the order of a few seconds at the most). If there is such a failure, the record and LUT are added to a retry queue and not back to XTQ. Every XTQ has a companion retry queue associated with it. The DC thread first picks up all the records from the retry queue and then fills up the rest of the next group from XTQ during a period. The service thread for the partition will process this group as before and update LST appropriately based on which records have been successfully shipped as part of this group.

4.3.1 Throttling: If the retry queue keeps getting bigger, it is an indication of a more persistent failure in communicating with the destination. After a certain threshold (of the size of retry queue) we start throttling the throughput allocated for this partition to adapt to this failure. Each partition has a certain maximum throughput customers configure based on their environment (for a given namespace and a destination combination). We throttle this all the way down to one record per second if required in steps (essentially an exponential backoff).

If XDR continues to throttle for a long time while the incoming writes at the source cluster continue at a faster rate than XDR is able to ship, XTQ will keep growing until it hits its maximum size (typically 16K entries). On reaching the maximum size, XDR will enter recovery mode (described next) for the specific partition. Other unaffected partitions can continue normal processing.

4.3.2 Recovery Mode: When the DC thread detects that XTQ has filled up (reached its maximum size limit) it signals to another thread called the Recovery thread that a recovery needs to happen for this partition (by setting a shared variable called ToBeRecovered to True). The Recovery thread polls for this variable and gets into motion as described below when the variable is True. The DC thread, after signaling the recovery thread, empties XTQ and subsequent writes to the partition will start filling XTQ again.

The high level idea of the recovery mode is to use the fact that Aerospike stores the LUT of the record in primary index and typically has the primary index in memory. This allows for a quick scan of the primary index to locate records that have changed after a certain time (such as the LST of the partition). We read a group of records that have changed after LST and then ship them using the service thread specific to this partition as we did during normal processing. So essentially, this bestows on XDR a **Rewind** capability that has proven to be useful in many other contexts - XDR is efficiently able to rewind and replay changed records from a point of time in the past.

Algorithm 1 outlines the recovery algorithm. We first reset the shared variable ToBeRecovered (in Line 1) (to indicate the recovery process has started a recovery). We then note down the current time in Line 4 which we use in Line 11 to set the LST of this partition. The loop starting at Line 7 implements the above primary index scan we mentioned above. We do not update the LST inside the loop starting at Line 7. XTQ can fill up again while the recovery mode is in progress and the shared variable ToBeRecovered tracks that state. If the DC thread has not detected a XTQ overflow the recovery thread exits Algorithm 1. If there was an overflow, the recovery thread goes back into recovery mode and executes Algorithm 1 again. This will terminate as with each round of the recovery the LST advances and XDR catches up soon. Note how the work done during the recovery is proportional to just the number of records that got updated after LST and not the number of actual updates (a record can get updated multiple times).

Algorithm 1 Recovery Thread Algorithm

```
1  ToBeRecovered = False
2  // reset this variable
3  // XTQ can become full again while in recovery mode
4  RecoveryStartTime = Current Time
5  // Note that LST is not advanced until
6  // the following loop completes
7  For each digest in primary index
8      if LUT of digest > LST
9          Schedule Read of record
10         /* Service thread will pick up */
11 LST = RecoveryStartTime
12 If !ToBeRecovered
13     // ToBeRecovered is set to true by the
14     DC thread if XTQ becomes full again */
15     Exit Recovery Mode
16 Else
17     Call Algorithm Recovery Mode Again
```

There is yet another situation when a node may get into recovery mode. Let node A be a node that has been disconnected from the cluster and let P be a partition for which A is the original master. Let B be the new master for P when A was disconnected. When A rejoins the cluster, A will start receiving the LST for P every two seconds from B. Further, A will rejoin the cluster with an empty XTQ. A will receive migrated data (updates it missed when it was disconnected) from B (this is not added to XTQ) as well as replica writes from B (this is added to XTQ) for new writes to records in P until A is ready to take over as master. After the master handoff from B to A, A will resume shipping for P. At master handoff time, if the oldest LUT in XTQ is bigger (later) than the latest LST obtained from B, A will go into recovery mode for this partition to ensure that no data is missed for shipping purposes.

As an aside, it is important to note that very early in the life of XDR we actually used all service threads for shipping records from a partition. This had two downsides: 1) We opened up many connections between this source node and the nodes in the destination cluster (each service thread on this node would have an

open connection to every destination node[6] that it could end up communicating with) and in many situations this caused us to run out of file descriptors and more importantly 2) spraying the records of a partition across all service threads especially during recovery mode meant the service threads came in the way of each other as there are some partition level locks (and some lower level locks called sprig locks) that become hotspots. This led us to come up with this notion of a partition having an affinity to a specific service thread. This solved both of the issues above - the proliferation of file descriptors as well as the threads contending for the same locks. This led to a dramatic speedup of the recovery of a partition.

4.3.3 Many Partitions in Recovery Mode: Although most of our description so far has been on a per-partition basis as there is not much impact of one partition on another typically, a situation where the network connectivity between the source and destination cluster is very slow or completely down is a special case worth considering.

When many partitions at a node all have to go into a recovery mode we get to an interesting tradeoff. On the one hand we want to maximize the available network bandwidth across many partitions but on the other hand it would be good to start and complete the recovery of a single partition as fast as possible so we do not have many rounds of recovery for a given partition. We pick a subset of partitions that we will simultaneously recover to balance the above tradeoff based on many factors that we will not go into due to space constraints.

5 Advanced Features of XDR

In this section, we will look at a few advanced features of XDR that emerged as a result of real-life use cases that were brought to us by our customers. We will discuss them one after the other in turn.

5.1 Ship At Least Once in a Time Window and Ship All

The first feature request is related to our conceptual optimization of not shipping every version of a record. It follows from Section 4 that if the latest version of a record has not been shipped to the destination, our current algorithm still allows that record to be updated successfully at the source. We discuss mechanisms by which we do not allow the source record to be updated unless a somewhat recent version of the record has already been shipped. This was useful for the applications to give some guarantees on how far apart the source and the destination cluster could be in the event of failures.

There were two broad flavors of the guarantee that our customers wanted on how far apart the source and destination cluster could be on a record. We discuss them next.

5.1.1 Ship At Least Once in a Time Window: Many of our customers wanted to have a guarantee that if there were any updates in a pre-defined time window then at least one of those updates made it to the destination. The time window is typically of the order of seconds or minutes. We will now describe how XDR is augmented to ship at least once in a time window. The crux of the idea is outlined in Algorithm 2.

Algorithm 2 Ship At Least Once in a Time Window

```
If not in recovery mode
    If digest not in XTQ/Retry Queue
        // No other write in this
        // time window is in progress
        Add <digest, LUT> to XTQ
        Allow the write to complete
    else
        // pv = prev_version of digest
        // cv = curr_version of digest
        // pv is in XTQ/Retry Queue
        If pv and cv are in same interval
            Skip adding cv to XTQ
            Allow write to complete
        else
           // pv is in XTQ/Retry Q
           // pv and cv  not from same interval
           This write waits for pv to be shipped
else
    //XDR in recovery mode
    If prev_lut (of this digest) < LST
        // previous version has shipped
        Add <digest, LUT> to XTQ
        Allow the write to complete
    else
        // Recovery in progress and not caught up yet
        Reject this write
        // Usually transactions have very short timeouts
        // and if you have to wait for the recovery to
        // complete it is better to reject the write quickly
```

5.1.2 Ship All. Ship All versions tightens the guarantee even more by ensuring that each version reaches the destination before the next version is shipped. The strictest interpretation of PITR (see Section 6.2) is that we should be able to roll back the database to exactly what it was at some point of time in the past. For these situations we need to use Ship All to be able to guarantee this. Algorithm 2 can be modified to guarantee Ship All.

5.2 Bin Shipping

A second feature request is that our system ship the least amount of information possible between clusters. As mentioned before, being a NoSQL system, a lot of our customers had fairly large denormalized records and specific writes often changed only a few bins of the entire record. The network traffic costs (monetary) in cloud deployments where these clusters are deployed across different regions of a cloud provider or even across two different cloud providers or across a cloud provider and a private cloud infrastructure can get overwhelming fairly quickly.

We came up with the notion of a bin-policy that conveys to XDR whether the entire record or specific bins (such as a combination of specific named bins and/or bins that changed as a result of this

[6]Sharing network connections across service threads would lead to unnecessary lock contention.

write) need to be shipped. To support this feature, for each bin that needs to be shipped across clusters, we need to maintain LUTs at a bin level. When a record is ready to be shipped by the service thread, all the bins that have LUTs bigger than LST will need to be shipped. The record level LUT will continue to be maintained and will be greater than or equal to all the bin level LUTs for the record (there can be bins that are not being shipped that will therefore not have bin LUTs and may have been updated last). One optimization worth mentioning is that we do not maintain a bin LUT if it is the same as record LUT. In particular, therefore, if we want to ship only two named bins (say bin A and bin B) if they have changed and all writes on the source cluster only update both bins together then we do not have to store bin LUTs for A and B ever (although we only ship bins A and B). The cost of extra bytes for these bin LUTs in the storage for the source cluster was well worth for most customers who cared about their network traffic costs.

5.3 Convergence

The third feature request is related to the desire of some customers to allow both the source and the destination clusters to be updating the same record (and have XDR perform bi-directional replication). Such an active-active deployment could result in the following different situations:

- Active-Active is often used to allows clusters to work on mutually exclusive set of records (such as one cluster works on all users who live in the USA while another works on all users who live in Europe). In these cases, active-active poses no major challenges.
- In many other situations, the clusters update mutually exclusive bins of the same record by design. For example, since many of our records are de-normalized the clusters may update different entities represented inside the same record. XDR provides the ability to merge these updates and this enables one of the clusters be ready to take over in case the other cluster fails.
- There are yet other situations where customers even want the ability to update the same bin in both the clusters and have conflicts resolved automatically.

The bin-LUT introduced in Section 5.2 allows us to compare two versions of a bin for the same record and decide which one is a later version in terms of the LUT value. This is essentially a Last-Write-Wins (LWW) approach using LUTs on the respective clusters to decide which is the last write. Recall that we do not insist on LUTs to be synchronized across clusters. The goal of comparing LUTs and ensuring the larger LUT wins is to make a consistent decision on both the clusters, not to pick the latest write with absolute certainty. Note that, as of the time of this publication, we do not support CRDTs (Conflict Free Replicated Data Types) which provide a much better solution to this problem in many cases.

5.4 Filtering

The next big request has been to bring to bear the rich expression filtering capabilities supported by the data model in Aerospike to XDR to help XDR decide what records (and fields through bin-shipping described in Section 5.2) should actually be shipped. So we are essentially applying our querying capabilities on streaming data (comprised of updates to the database). The expressive power of the expression language filtering has helped many enterprises also stay in compliance with data locality regulations such as GDPR, DPDPA etc. In addition, for purely internal organization compliance or performance reasons many enterprises carefully choose what data gets replicated. The co-design of XDR and Aerospike database means there is no extra copying into another process or machine, writing to disk, rereading it from disk and then applying the filters in a different tool - all of these add to overheads in terms of hardware and the overhead of managing multiple tools.

6 Other Applications of XDR

We have so far described XDR in the context of minimizing lag between a source cluster and a destination cluster. The XDR machinery however generalizes to many other uses cases in real-life deployments. We discuss a few below.

6.1 Bringing Up New Clusters

The primary design goal of XDR was to keep two existing clusters in sync but many of our customers use XDR to just bring up a new cluster. They may bring up a new cluster to temporarily increase production capacity in anticipation of a one time spike due to a holiday sale or increase capacity permanently by adding another cluster (such as another edge cluster in the edge and core architecture). Many times, enterprises want to just test our their latest application update and want as close to the production data on a cluster as they can get.

Bringing up a new cluster will employ the Rewind capability but will start from the beginning of time in this case. A new destination is set up at the source cluster at a certain point of time, say 't'. Filtering is applied to filter out records that have a LUT greater than t. At some point the destination would have caught up with all records upto 't' and the destination can be decommissioned. The throughput of XDR can be capped for this destination and essentially "steal" resources in the background to bring up the new cluster.

6.2 Backup

It is easy to see based on Section 6.1 how XDR in conjunction with rewind and filtering can be used to backup as of a certain point of time. A variation of this is that we can also do incremental backups - these have a start time (the time at which we stopped the previous incremental backup) and a stop time. Again, a combination of XDR Rewind and filters would achieve this purpose. The incremental backups can be arranged when there is low traffic on the source cluster.

6.3 Fixing XDR Configuration Errors

XDR Rewind is often used for correcting configuration mistakes that lead to missed records or incorrect records.

6.4 Change Data Capture

Aerospike has created several connectors that make the process of capturing the changes on a source cluster and exporting them very simple. Kafka, Pulsar, Elastic Search are some examples of outbound connectors. These connectors are destinations that receive

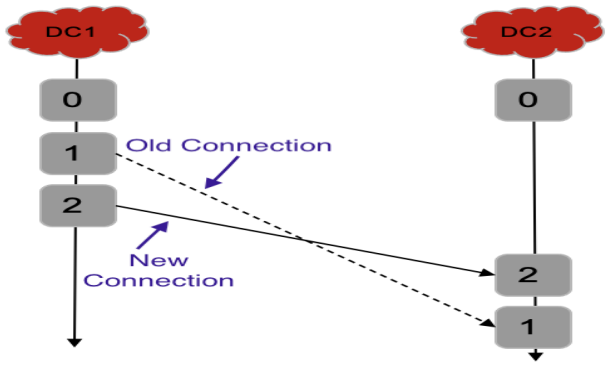

Figure 11: Out of Order Packets

the source XDR stream and then connect to some other system on the other side.

Ship All (Section 5.1.2) is particularly interesting as this allows every change to be captured at a destination and that can be acted upon further - this could be used to maintain real time aggregate information in a streaming fashion, or just plain file away the changes for audit purposes or build a PITR system.

7 XDR Correctness Guarantees

The most basic version of XDR described in Section 4 where we are allowed to skip arbitrary number of updates to a record is the default mode for XDR. It is also the highest performant and is used by most of our customers. With a very high degree of probability, it guarantees eventual consistency. Our usage of the term eventual consistency is that if no new updates are made to a given record on the source cluster, eventually all accesses to that record on the destination cluster will return the latest value at the source cluster.

If there were no failures ever in the system, then as designed XDR will guarantee eventual consistency. The reason for this is that XDR always ships the latest version when it knows there has been a change to a record and the service thread responsible for shipping a record has exactly one connection with the destination node and so all updates are sent in a FIFO order.

However, if the connection to the destination were to break and a new connection were to be established (either between the same source and destination node pair or one or both change due to master ownership change) then there is a small chance that an older packet containing an older update may get to the destination after a newer packet with a newer update has been accepted (see Figure 11). In this case, by default, the older update will overwrite the newer update. Of course, when there is an update again things will get back in sync again and this works fine for many applications for whom these are features for a ML model. These applications want to have minimal overheads during normal processing while minimizing lag - paying a small penalty in terms of lag during outlier conditions for a few records is a perfect design choice for them.

Having said that, there are at least two ways in which customers who care about versions at the destination being ordered and therefore eventual consistency (under all failure scenarios) can get it from Aerospike. The first is by using our convergence option (in an active-passive setting), this automatically ships the bin LUTs (all LUTs are from a single cluster and the master ensures LUTs on successive versions of a record are monotonic even if the clocks are not synchronized across nodes of a cluster) and the destination cluster will not overwrite a newer version of the bin for the record. The second is by using Ship All which guarantees that every version makes it one after the other to the destination. Table 1 summarizes all these scenarios.

8 Performance Results

We have been running XDR in production in several enterprises and we wanted to re-establish through these experiments what we have heard many times over from our customers. They are pleasantly surprised when Aerospike first comfortably manages their workload consistently at a much lower latency and a much higher throughput on the same infrastructure on which they are currently running their databases. Even more surprising is the fact that adding XDR to the mix to have a destination cluster ready for disaster recovery hardly adds any overhead to the existing Aerospike source cluster.

Our experimental set up was as follows: We run a mixed workload of 50% reads and 50% writes against an Aerospike cluster S (S for source) of three (and six in the next set of experiments) c7i.2xlarge instances on AWS. The entire database was in memory as our main objective was to obtain a high node throughput at low latency as the baseline and then analyze what would happen when we added XDR. Furthermore, having data in memory is worse than having it on disk for XDR. The reason for this is that when data is on disk XDR most often finds the data it needs to ship in memory (Aerospike has a notion of a post write queue that holds recently written write blocks in memory for precisely this reason). In contrast, our reads are random and always end up going to the disk most of the time unless it is found in the post write queue (not at all common). The writes of course have to go to the disk. So, by holding data in memory rather than disk we are favoring normal transactions over XDR.

We loaded the CPUs until they were at 50% utilization and that resulted in us pushing 135K read TPS and 135K write TPS for a total of 270K TPS through the above system. We normally recommend 50% CPU utilization as the default operating point to have enough headroom for handling node failures within the cluster and other background housekeeping jobs such as de-fragmentation etc. We also pushed the utilization to 75% and the total throughput went from 270K to about 400K TPS as would be expected. The read and write latencies in all these cases were sub-ms and we verified the same trend we report at 50% holds.

We then added a destination cluster (D) that was identical to the source cluster S but on the opposite coast in the US. S was in us-west-1 region while D was in us-east-1 region. We observed latencies between 70 and 75ms during our experiments between these two regions.

We enabled XDR from S to D and report on metrics during normal mode and also when we deliberately set XDR to rewind by different amounts of time (this simulates from a XDR point of view a network outage between S and D of that duration). We also

Table 1: Correctness Guarantees

Failure Scenarios	Basic XDR (Section 4)	Convergence (Section 5.3)	Ship All (Section 5.1.2)
No Failures	Eventually Consistent	Eventually Consistent	Eventually Consistent
TCP Connection Failure	Older version possible	Evebtually Consistent	Eventually Consistent
Source Master Failure	Older version possible	Eventually Consistent	Eventually Consistent
Destination Master Failure	Older version possible	Eventually Consistent	Eventually Consistent

Table 2: Impact of XDR on Source Transactions and XDR Throughput (3 node cluster)

XDR Rewind in Secs	Read		Write		XDR TPS
	TPS	Outlier Latency %age	TPS	Outlier Latency %age	
No XDR	135558	0	135192	0	NA
XDR Normal	131482	0.01	132167	0.1	129939
10	131757	0.01	131424	0.13	120702
30	129420	0.01	129372	0.27	114408
60	130183	0.01	129372	0.28	113040
∞	127702	0.01	128395	0.29	118701

Table 3: Impact of XDR on Source Transactions and XDR Throughput (6 node cluster)

XDR Rewind in Secs	Read		Write		XDR TPS
	TPS	Outlier Latency %age	TPS	Outlier Latency %age	
No XDR	272021	0	270547	0.01	NA
XDR Normal	264386	0	264077	0.07	260334
10	263302	0.01	264203	0.17	259908
30	265931	0.01	266247	0.31	259556
60	267145	0.01	267487	0.32	259650
∞	267705	0.01	267351	0.36	259602

rewind all the way to the beginning of time to simulate a very long network outage. Table 2 lists the source throughput and latency as XDR does more and more work, as well as XDR throughput. This clearly shows that XDR has very little impact on the source cluster transactions latency and throughput as per the design goal. The column outlier latency percentage measures the percentage of transactions that have a latency over a milli-second. So 0.01 means 0.01% of all transactions have a latency greater than 1 ms. We also measured the CPU load when XDR was running and in all these cases the load went up from 50% to 52.5% demonstrating the real value of our co-design philosophy.

In all our experiments, when XDR was in normal mode the lag between D and S was less than a second (we measure lag in increments of seconds and, therefore, lag was zero). XDR Throughput under normal conditions is a tad bit lower than the write throughput that just demonstrates that not all updates get shipped as per the design. As the XDR rewind time increases, XDR throughput drops even further due to the fact that we skip even more intermediate updates. The XDR TPS when we rewind all the way back to the beginning (denoted by row with the value ∞) rises back up a notch - this is due to the fact that there are a lot of records that need to be shipped and actually transiently the XDR TPS goes up way much higher than the steady state of about 120K TPS.

The time to recover back to normal mode from the rewind is of the order of 20 to 30 seconds for all cases except when we rewind all the way to the beginning. It was about 50 seconds when we rewound all the way to the beginning. This time to catch up when we rewind all the way back is only a function of the size of the database - in this experiment we used just 10 million records to be able to repeat our experiments several times. However, we have validated this fact that the time to catch up when we rewind all the way back is purely a function of the database size (and the network throughput) several times while debugging live in our customer's environment and helping them create backup clusters etc.

We reran the entire experiment for a 6 node cluster to establish that as the Aerospike cluster scales and the throughput scales, XDR also scales perfectly in all aspects. Table 3 shows the source transaction metrics when we scale the cluster to six nodes. Note how the read and write throughput doubles when the number of node doubles as expected. The outlier latency percentage remain largely the same as the number of nodes scale - again a typical behavior of an Aerospike cluster. The XDR throughput is just a tad short of the write throughput of the cluster and the recovery times behave very similarly to the 3 node case with a small change - since the same 10 million records are spread over six nodes the time to rewind from the beginning drops to about 30 seconds.

9 Related Work

Asynchronous Replication is supported by a variety of NoSQL [1, 2, 4, 6, 9, 17, 23], NewSQL [3, 24, 26] and relational database systems [7, 13, 16, 18, 27].

The most well understood and widely used asynchronous replication mechanism by almost all of the NewSQL [5] and Relational databases [7, 13, 16, 18] uses some form of redo logs or operational logs that is replayed at the destination cluster. The focus in these systems is to ensure transactional (consisting of multiple records) consistency across the source and destination clusters. As a result, these systems usually write logs to local storage and then ship those logs to the destination which is replayed. Every update that needs to be reflected at the destination has to be shipped and these updates have to be maintained at the local disks or a shared network device (which further increases the latency).

Our first attempt at asynchronous replication was a variation of this in which we logged the digests that have been modified (and not the entire changes to a record) and then read the corresponding record just before shipping to the destination. Thus, this system also skipped intermediate writes to a record. This worked well up to a point but using a single digest log per node meant that we could not reclaim log space in these logs until the slowest of the multiple destinations was able to consume the digests in that space. We had not implemented XDR Rewind in our first attempt and therefore had to keep the digest log until it was consumed by all destinations. In fact, one of the early customers of our asynchronous replication, almost a decade ago, used dozens of destination clusters which were fed by just two clusters in the core of their network data center.

In contrast, XDR quickly performs replication from the source node memory to the destination node and XTQ is maintained per destination so the destinations do not collide with each other at the source. XDR uses the Rewind mechanism in case XTQ fills up which has proved to be extremely efficient.

The NoSQL systems, in contrast to the NewSQL and relational database vendors do not attempt to keep the destination cluster transactionally consistent [12]. Some NoSQL systems (e.g., [4]), like XDR, attempt to replicate from memory to memory (but still store every update in memory) under normal circumstances but when there are failures they resort to keeping a log of updates on the disk in contrast to our approach of XDR Rewind which adds no other storage overhead. In particular there is no overhead per destination or overhead that grows as a function of the slowest destination.

It is important to reiterate that one of the key reasons for the efficiency of XDR is that our design goal was to keep the lag between the source and the destination cluster as small as possible. This allows us to skip shipping intermediate versions as they do not have any role at the destination once the next version has been created. The decision of what records get skipped are made independently for each record. Needless to say, such a design goal can be co-opted into any database system (Relational or NoSQL) if the applications can live with the consequences of this decision. One of the most important consequence is that a snapshot of the destination may not be transactionally consistent (may contain some writes of a transaction but not others).

The authors in [15] propose a dynamic mechanism for moving from synchronous replication to asynchronous replication if the write latency at the source cluster is getting too big relative to the SLAs desired. They call this semi-synchronous replication. Although, this has some appeal as a conceptual mechanism, it is not very useful in practice. It neither can guarantee strong consistency nor the low latency associated with asynchronous replication. Note, that semi-synchronous replication is used in a very different context to just mean quorum consensus (i.e., do not have to write to all replicas but neither are we satisfied with writing to only one replica) in [10, 14].

10 Conclusions

We have introduced XDR, an innovative approach to asynchronous replication in a database system, co-designed within the same process as the core database. The key outcome of this co-design is that updated records are transferred in a single hop—from the database process on the source node directly to the database process on the destination node, thus, eliminating intermediate disk copies and inter-process communication. This architecture allows XDR to preserve both the ultra-low latency characteristics of the source cluster (average latency is sub-millisecond, with p99 latency in the single-digit milliseconds) and the high throughput characteristics, processing millions of transactions per second. On top of not impacting transaction performance on the source cluster XDR supports high-throughput updates to remote clusters located hundreds of milliseconds of network latency away while minimizing the lag. In contrast, replication approaches based on logs cannot achieve these performance characteristics due to the overhead introduced by additional processes, memory copies, and disk operations. Furthermore, XDR's ability to handle hotspots by skipping intermediate updates while still minimizing lag provides a significant performance advantage. The co-design approach also enables efficient implementation of advanced features, such as i) Filtering, ii) Convergence iii) Ability to control how far apart the records in the source and destination cluster can drift and iv) Rewind. These features have enabled diverse and practical use cases for XDR in real-world deployments. We believe that this level of performance and flexibility would not have been achievable without the co-designed architecture.

Acknowledgments

We thank Piyush Gupta for contributing several diagrams used in this paper. We also thank Akash Kumar and Suresh Togas for helping with the performance experiments.

References

[1] Shannon Bradshaw, Eoin Brazil, and Kristina Chodorow. 2019. *MongoDB: the definitive guide: powerful and scalable data storage.* O'Reilly Media.
[2] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.
[3] 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.
[4] Couchbase. 2023. *WhitePaper: High Availability and Disaster Recovery for Globally Distirbuted Data.* Technical Report. CouchBase. https://info.couchbase.com/rs/302-GJY-034/images/High_Availability_Disaster_Recovery_Globally_Distributed_Data_XDCR_.pdf Accessed: 2024-11-08.
[5] Cockroach DB. 2024. *Physical Cluster Replication.* Technical Report. Cockroach Labs. https://www.cockroachlabs.com/docs/stable/physical-cluster-replication-overview Accessed: 2024-11-08.
[6] Mostafa Elhemali, Niall Gallagher, Nicholas Gordon, Joseph Idziorek, Richard Krog, Colin Lazier, Erben Mo, Akhilesh Mritunjai, Somu Perianayagam, Tim Rath, Swaminathan Sivasubramanian, James Christopher Sorenson III, Sroaj Sosothikul, Doug Terry, and Akshat Vig. 2022. Amazon DynamoDB: A scalable, predictably performant, and fully managed NoSQL database service. In *USENIX ATC 2022.* https://www.amazon.science/publications/amazon-dynamodb-a-scalable-predictably-performant-and-fully-managed-nosql-database-service
[7] Bettina Kemme and Gustavo Alonso. 2010. Database replication: a tale of research across communities. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 5–12.
[8] Mikhail Kourjanski. 2018. ML Data Pipelines for Real-Time Fraud Prevention. In *QCon.* https://www.infoq.com/presentations/paypal-ml-fraud-prevention-2018/
[9] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS operating systems review* 44, 2 (2010), 35–40.
[10] MariaDB. 2024. *Semi Synchronous Replication.* Technical Report. MariaDB. https://mariadb.com/kb/en/semisynchronous-replication/ Accessed: 2024-11-08.
[11] Florian Mendel, Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen. 2006. On the collision resistance of RIPEMD-160. In *Proceedings of the 9th international conference on Information Security.*
[12] Microsoft. 2024. *Global Tables - multi region replication for Dynamo DB.* Technical Report. Microsoft. https://docs.aws.amazon.com/amazondynamodb/latest/

developerguide/GlobalTables.html Accessed: 2024-11-08.
[13] Microsoft. 2024. *SQL Server Replication, Microsoft Website.* Technical Report. Microsoft. https://learn.microsoft.com/en-us/sql/relational-databases/replication/sql-server-replication Accessed: 2024-11-08.
[14] MySQL. 2024. *MySQL Reference Manual.* Technical Report. MySQL. https://dev.mysql.com/doc/refman/8.4/en/replication-semisync.html Accessed: 2024-11-08.
[15] Assaf Natanzon and Eitan Bachmat. 2013. Dynamic synchronous/asynchronous replication. *ACM Transactions on Storage (TOS)* 9, 3 (2013), 1–19.
[16] Oracle. 2024. *Oracle Goldengate.* Technical Report. Oracle. https://www.oracle.com/integration/goldengate/ Accessed: 2024-11-08.
[17] Conor Power, Hiren Patel, Alekh Jindal, Jyoti Leeka, Bob Jenkins, Michael Rys, Ed Triou, Dexin Zhu, Lucky Katahanas, Chakrapani Bhat Talapady, et al. 2021. The cosmos big data platform at microsoft: Over a decade of progress and a decade to look forward. *Proceedings of the VLDB Endowment* 14, 12 (2021), 3148–3161.
[18] Hans-Jürgen Schönig. 2015. *PostgreSQL Replication.* Packt Publishing Ltd.
[19] V. Srinivasan and B. Bulkowski. 2012. Citrusleaf: A Real-Time NoSQL DB which Preserves ACID. In *Proceedings of the VLDB Endownment.*
[20] V. Srinivasan, Brian Bulkowski, Sunil Sayyaparaju Wei-Ling Chu, Andrew Gooding, Rajkumar Iyer, Ashish Shinde, and Thomas Lopatic. 2016. Aerospike: Architecture of a Real-Time Operational DBMS. In *Proceedings of the VLDB Endowment*, Vol. 9.
[21] V. Srinivasan, Tim Faulkes, Albert Autin, and Paige Roberts. 2024. *Aerospike: Up and Running.* O'Reilly Media.
[22] V. Srinivasan, Andrew Gooding, Sunil Sayyaparaju, Thomas Lopatic, Kevin Porter, Ashish Shinde, and B. Narendran. 2023. Techniques and Efficiencies from Building a Real-Time DBMS. *Proc. VLDB Endow.* 16, 12 (Aug. 2023), 3676–3688. doi:10.14778/3611540.3611556
[23] Nishant Suneja. 2019. Scylladb optimizes database architecture to maximize hardware performance. *IEEE Software* 36, 04 (2019), 96–100.
[24] 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.
[25] David G. Thaler and Chinya V. Ravishankar. 1996. *A Name Based Mapping Scheme for Rendezvous.* Technical Report. University of Michigan, Ann Arbor, Michigan.
[26] Nathan VanBenschoten, Arul Ajmani, Marcus Gartner, Andrei Matei, Aayush Shah, Irfan Sharif, Alexander Shraer, Adam Storm, Rebecca Taft, Oliver Tan, et al. 2022. Enabling the next generation of multi-region applications with cockroachdb. In *Proceedings of the 2022 International Conference on Management of Data.* 2312–2325.
[27] Lik Wong, Nimar S Arora, Lei Gao, Thuvan Hoang, and Jingwei Wu. 2009. Oracle streams: A high performance implementation for near real time asynchronous replication. In *2009 IEEE 25th International Conference on Data Engineering.* IEEE, 1363–1374.