UNIVERSITY OF CALIFORNIA,
IRVINE

Active Replication in AsterixDB

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Computer Science

by

Akshay Manchale Sridhar

Thesis Committee:
Professor Michael J. Carey, Chair
Professor Chen Li
Assistant Professor Ardalan Amiri Sani

2017

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to first thank my thesis advisor, mentor, and a friend, Professor Michael Carey, who has been immensely supportive from the beginning. This thesis would not have been possible without him steering me to ask the right questions while giving me the freedom to answer those questions. His humility, passion, knowledge and dedication have shown me how to be a good researcher.

I would like to thank the AsterixDB community for their support whenever I needed help. I would like to thank Ian Maxon, for being a good friend, and his continuous support and inputs have been truly valuable in understanding and working with AsterixDB.

I would to thank my committee members Professors Chen Li and Adralan Amiri Sani for reviewing my thesis.

The work reported in this thesis has been supported in part by NSF IIS award 1059436 and NSF CNS award 1305430. In addition, I would like to acknowledge the Donald Bren Foundation for its generosity in contributing to the scientific investigation that has driven my research and academic goals.

ABSTRACT OF THE THESIS

Active Replication in AsterixDB

By

Akshay Manchale Sridhar

Master of Science in Computer Science

University of California, Irvine, 2017

Professor Michael J. Carey, Chair

AsterixDB is a Big Data Management System (BDMS) designed to manage data on clusters of commodity hardware. In any distributed system, failures of hardware and/or software components in the system are an eventual certainty and a choice has to be made between consistency and availability. AsterixDB, being a CP system (as opposed to an AP system), sacrifices availability when there is a partition in the network and seeks to maintain data consistency.

In this thesis, we describe a replication protocol called Active Replication that eagerly replicates the state of a node in the cluster to one or more replicas and provides a Mean Time to Recovery (MTTR) close to the time needed to detect a given failure. The protocol works without sacrificing the consistency of the system by exploiting properties of AsterixDB's record-oriented transactional model and the lifecyle of its Log-Structured Merge-trees. We describe the implementation of the Active Replication protocol as well as the fault-tolerance mechanism built on top of the protocol. We evaluate the performance of the protocol and show how to achieve a low MTTR with a 10–25% decrease in ingestion throughput.

Chapter 1

Introduction

Distributed data management systems have overtaken centralized RDBMS systems in recent years due to the volume of data involved. Many systems have been developed in recent years to handle the scale of data that we see today. However, scaling out horizontally by adding more systems (nodes) connected by a network introduces new failure models that have to be carefully handled. Additionally, there needs to be some coordination between the systems to make them behave like a single system. Much like software, hardware is not free from faults and can fail without fair warning. These failures are more common in distributed systems due to the number of components involved and the fact that they are physically separated from each other and connected by unreliable networks. For instance, the mean time to failure (MTTF) for a disk to fail in a cluster of systems is much higher than the MTTF per driver mentioned in the device specification [1]. Certain applications demand SLAs that have very stringent requirements on being available in case of failures, and how we recover from failures is as important a design question as the rest of the system.

Introducing redundancy is a common approach to handling failures and to remaining available. Redundancy can be employed either in the hardware, software or both. In data management systems, replication is a common approach to handling failures by duplicating the underlying data in multiple physical systems. Duplicating the state of a system comes with the costs of increased communication overheads and synchronization to maintain a consistent state. Data replication protocols are complex and can be implemented in multiple layers of a system. System design decisions have to make tradeoffs between availability and consistency, throughput and latency etc. Over the years, many data replication methods have been developed [2, 3], each with its own advantages and disadvantages [4].

In this thesis, we have implemented a new data replication protocol in AsterixDB that keeps hot-standbys ready to take over the responsibilities of a node in the cluster and that provides a mean time to recovery (MTTR) that is very close to the time required to detect a failure. We do this with an acceptable reduction in the overall throughput of the system by exploiting the underlying data partitioning, isolation model, and properties of Log-Structured Merge Trees [5] without compromising the consistency of data.

The rest of this thesis is structured as follows: Chapter 2 describes related work and approaches in popular Big Data Management Systems. Chapter 3 provides the background of AsterixDB and its existing data replication protocol. Chapter 4 details the design and implementation of the new Active Replication protocol and its design for fault-tolerance.

Chapter 5 evaluates the performance of the protocol’s implementation. Chapter 6 concludes the thesis and discusses avenues for continued work on Active Replication.

Chapter 2

Related Work

2.1 Data Replication in Centralized Databases

Centralized data management systems have replication protocols that are designed to duplicate the data and the system state in a different geographical location that can take over when there is a failure of the primary database [6]. The consistency requirements of the primary and the backup copy leads to two possible replication strategies in centralized databases: 1-safe and 2-safe [7]. 1-safe algorithms apply all changes to the primary copy and then asynchronously replicate the new physical data or replay the logical operations on the replica. However, this means that when there is a failure of the primary copy after a transaction commits, the replica may not have received the information about the last few transactions and it will have no way to recover that data, leading to an inconsistency in the system. This is an acceptable tradeoff for certain applications that do not have strict requirements on the consistency of the data. Since the primary system does not have to coordinate with the replica for every operation, the latency involved in updating 1-safe data is close to the case with no replication since an operation does not need to wait on the replica(s) to make progress. In contrast, 2-safe algorithms make progress by synchronously updating both the primary and the replica for every committed operation. 2-safe algorithms ensure that any side effect visible in the primary site will always be visible in the backup site. This requires coordination between the primary and replica for every operation, so the minimum latency of a committed operation is at least one round trip time between the primary and the replica [8].

2.2 Data Replication in Distributed Databases

Distributed Databases differ from centralized databases in that there are now multiple sites where a given operation can happen. Replicating data is useful for improving both the availability and the performance of the system. Replication techniques can be broadly classified based on the nature of the protocol (synchronous or asynchronous), the cluster architecture (primary copy or update-everywhere), and commit protocols (voting or non-voting) [9]. Among these strategies, it is possible for distributed databases to utilize a single primary copy replication scheme by (horizontally) partitioning data between the machines in the cluster [10, 11]. By partitioning the data, a system can provide the semantics of a single master database, similar to a centralized database, except that the primary copy of each partition is only responsible for its subset of the data. Such a scheme still faces similar challenges due to fundamental impossibilities that are inherent in every distributed system. The CAP theorem [12] provides a view of what is possible in a distributed system when there is a partition in the network. It states that when there is a partition, a database can either be consistent (CP) or available (AP) but not both at the same time. It is important to note that the CAP theorem does not “allow” CA systems; while a system can be both consistent and available during normal operating conditions, the theorem encourages system designers to reason about the behavior of the system in the event of a network partition. That is, the choice between Consistency and Availability has to be made when there is a network partition or a fault/failure of a machine in the cluster. Data replication is useful in CP systems to recover after failures, and there will be a window during recovery where the system will be unavailable [13]. In AP systems, data replication will generally allow for updates to go to multiple places so that the overall throughput of the system is higher and so that a failure cannot halt progress of the entire system [14]. AP systems operate assuming that data inconsistency is inevitable, so they instead provide mechanisms to provide a reasonably consistent picture of the data in the cluster. Fundamentally, this means that it is possible to have replicas with different information that may be reconciled at a later time, but that will be inconsistent for unbounded amounts of time as long as there are nodes that are unreachable due to a partition in the network or a failure of the underlying hardware or software running on a node.

Chapter 3

AsterixDB Background

Apache AsterixDB [15] is a Big Data Management System (BDMS) that is well suited for large scale data ingestion, social media data storage and analysis, data warehousing, and other Big Data applications. It is built on top of a parallel runtime layer called Hyracks [16] which is different from the traditional MapReduce-based Big Data platforms. AsterixDB supports semi-structured data storage with a rich set of data types, and it supports querying through the Asterix Query Language (AQL) and SQL++ [17]. A detailed overview of AsterixDB can be found in [18]. In this section, we will discuss the architecture of AsterixDB and its storage and transactional model and examine its existing support for fault-tolerance.

3.1 Architecture and Data Model

Figure 3.1 shows the high-level architecture of an AsterixDB cluster. A Client communicates with the system through the Cluster Controller (CC) that will forward the request to the appropriate Node Controllers (NCs). The cluster controller manages the state and lifecycle of the cluster and also exposes RESTful APIs to ingest and query data. One of the Node Controllers in the cluster is designated as the Metadata Node and hosts the system catalogs.

When a user submits a query, the query is compiled into a Hyracks job that is expressed as a Directed Acyclic Graph of operators and connectors. The job's DAG is executed as a pipeline with support for the parallel execution of operators on the NCs. The underlying records are encapsulated in frames that are pushed between the operators during job execution.

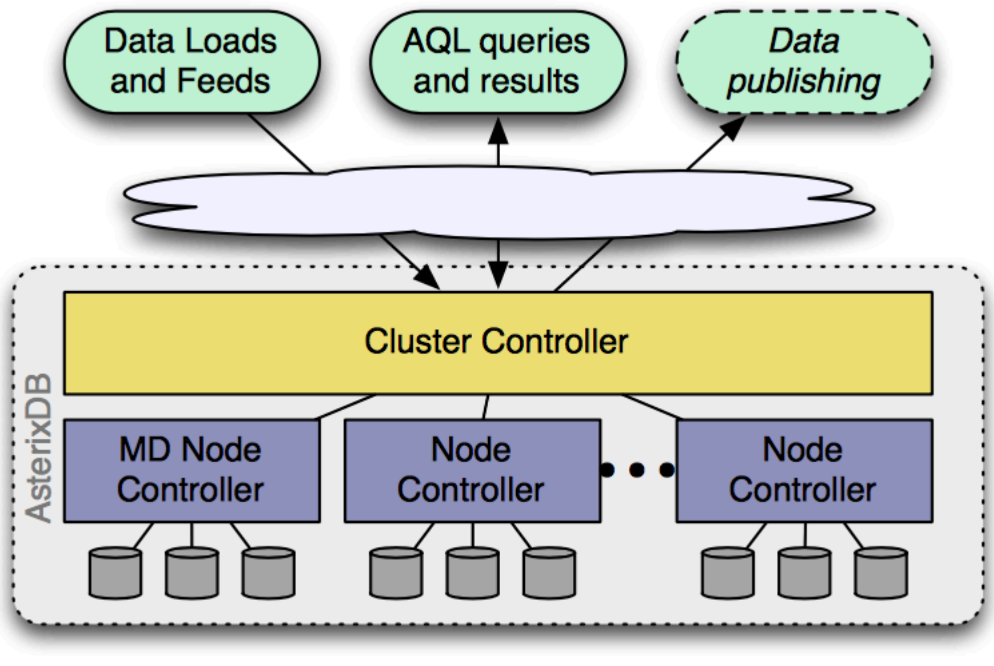

Figure 3.1: AsterixDB Architecture

AsterixDB uses a data model, called the Asterix Data Model (ADM), which is a super set of JSON [19]. The fields are optionally typed with support for user-defined types. The top level storage container is called a Dataverse, similar to a database in an RDBMS. Dataverses are populated with records that go into Datasets (akin to tables in an RDBMS) that are defined with a Datatype. Datatypes in AsterixDB are semi-structured, and the user can either have a fully-typed definition of each field in the Datatype or it can be defined as an open type that can accept any valid ADM object. All records residing in a dataset must conform to the structure defined by the given Datatype. The NC that hosts the Metadata has all the schema-related information about Dataverses, Datasets, and Dataypes in the cluster. As an example, Figure 3.2 shows an AQL DDL statement that creates a dataverse called SocialNetwork, a UserType with an ID, User name, the time when the user registered, and a list of Friend IDs. The UserType is then used to create a dataset called Users with the user ID as the primary key. AsterixDB can optionally generate a UUID and use that as the primary key if no user-provided primary key is defined in the dataset.

```
create dataverse SocialNetwork;
use dataverse SocialNetwork;

create type UserType as {
      userid: int64,
      username: string,
      since: date,
      friends: {{ int64 }}
};

create dataset Users(UserType) primary key userid;
```

Figure 3.2: Creating a dataverse, a data type, and a dataset.

3.2 Storage and Transactional Model

AsterixDB uses Log-Structured Merge (LSM) Trees [5] for all storage and indexing. LSM based indexes have an in-memory component that receives all writes until the indexes' memory utilization reaches a certain configurable threshold. The entries in the in-memory component are flushed into an immutable disk component as a single sequential write operation, similar to bulk loading a new index, when the in-memory threshold is reached.

The index components on the disk are periodically merged according to a merge policy [20]. This index lifecycle of writing to an in-memory component and sequentially writing to disk enables high volume data ingestion.

AsterixDB datasets are hash-partitioned on the primary key (or generated UUID), with all secondary indexes placed in the partition that contains the associated primary key for that record. A partition is local to a single Node Controller, and there is no shared state between partitions within a node or across Node Controllers in the cluster. Partitions are single master shards of a dataset's primary and secondary index records in AsterixDB. Figure 3.3 shows a three-node AsterixDB cluster with each node hosting two partitions of a dataset named *A*.

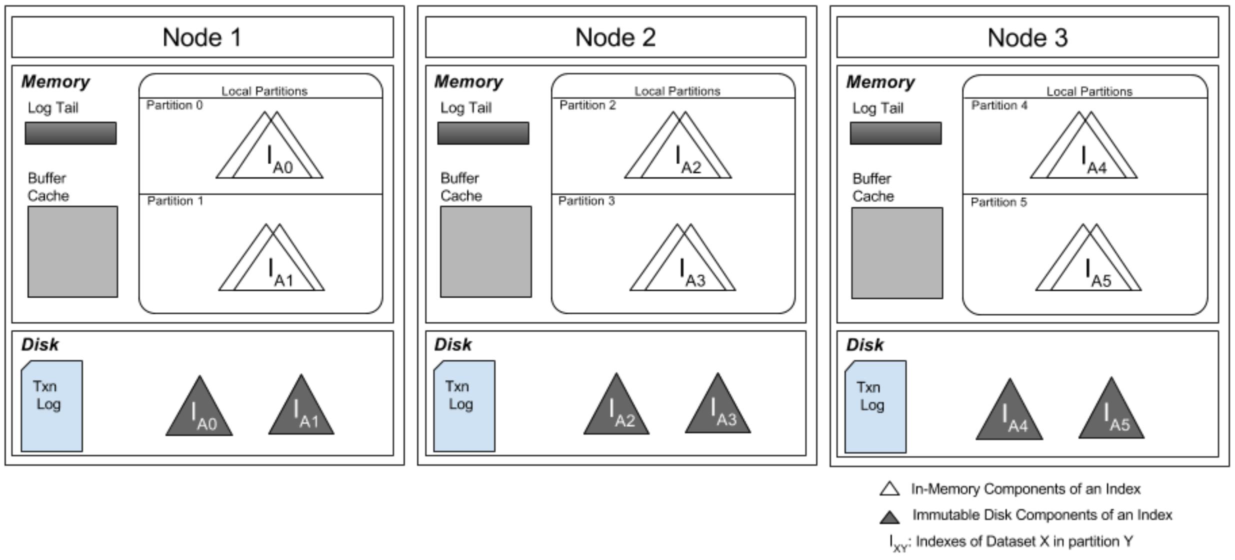

Figure 3.3: AsterixDB Node Controllers

Each node in the figure has two partitions, and the partitions are assigned globally unique identifiers. The Cluster Controller keeps track of the assignment of partitions to node controllers. The index created for dataset *A* has its representation in each partition of every node in the cluster. The index will have memory components in each partition (I_{A0} in partition 0, I_{A1} in partition 1 of Node 1), and when a memory component reaches its threshold, it is flushed and becomes an immutable disk component. The disk components of a dataset are periodically merged into larger components in a fashion defined by a merge policy.

Transactions in AsterixDB are only at the record level, that is, per-record updates to the primary and the secondary indexes are atomic and a record is visible for reading only after the primary and all associated secondary indexes for that dataset are updated. A write job can span multiple records but the set of operations over all the records is not atomic. For example, if we have a job that spans 1000 records, AsterixDB does not guarantee that the operations on all 1000 records will either occur or not. However, it does ensure that there will be 1000 individual atomic record operations regardless of the number of secondary indexes that are present on the dataset. In other words, a job might fail after 500 atomic record updates, and such a job failure will not roll back the changes related to those 500 record updates. However, if the 501st record being inserted has only updated the primary index when the failure occurs, that change will be rolled back unless all of the associated secondary indexes for that record's dataset have also already been updated. This Record-level isolation and partitioning over primary keys allows for any given transaction to be local to a single partition: Since transactional semantics are only applicable at the record level, and the primary and all associated secondary indexes for the primary are within a partition, no atomic operations happen across partitions. Additionally, since the system does not support transactions spanning multiple records, it avoids any coordination requirements between partitions and hence between node controllers for reading or writing a particular record.

For durability, AsterixDB uses a no-steal/no-force Write Ahead Log (WAL) that logs the changes to a record along with the logical operation performed on the record to a transaction log. This transaction log is the database and the source of truth for crash recovery. Since AsterixDB is based on LSM indexes that reside in memory, with updates going into the in-memory components, the only durable resource in the database is the transaction log until the LSM components are flushed to disk. In order to bound the time for recovery of the in-memory components, AsterixDB periodically checkpoints by requesting the LSM components to be flushed to the disk and by capturing the Log Sequence Number (LSN) corresponding to when this operation successfully completes. This allows the recovery algorithm to start reading the transactional logs from the last checkpoint to reconstruct the state before the failure. After a crash, the system does not allow any new jobs to make progress during local recovery processing from the transaction log; this is easy to control because we have a single cluster controller that manages the current state of the cluster.

The transactional log buffer on each NC is flushed to the log disk periodically (a la group commit) or whenever the log disk is idle. The latter ensures that there is no minimum latency for updates due to delayed periodic flushing of the log buffers. The record locks acquired by transactions are released when the associated log buffer has been successfully flushed, and the user gets a response only after the logs have been written to disk. Log buffers are recycled from a list to prevent other transactions from having to wait while one log buffer is being written to disk.

3.3 Passive Replication in AsterixDB

As described in the previous section, AsterixDB's partitions are each single masters – they are resident on only one node controller which will have all the state required for crash recovery, which happens locally. However, if a node controller fails, we completely lose the data within that partition until it comes back online, at which point the node controller can run local recovery by replaying the operations in the transaction log from the earliest flush operation or checkpoint in that node controller. Without replication, local partition storage is a single point of failure in the cluster and the cluster cannot service any read or write requests when one node controller is down because it could potentially serve incomplete information for read queries and it will not have the failed node controller to accept the writes for its partition(s).

AsterixDB has recently added support for high-availability and fault-tolerance by replicating the transaction log records in multiple node controllers. We call this approach Passive Replication [21] because, during normal operation, it does not keep an active (queryable) replica of the in-memory components of dataset partitions – only the log contents are replicated in order to prevent data loss. As a result, during failover of a Node Controller, the system first has to redo the operations of the failed partition by reading and replaying the remote log records that were received from the failed Node Controller before the replica can be used to process queries. Passive replication has two main operations:

1. For ongoing transactional operations, the local log records that are generated are sent over to the replicas and persisted on the replicas.
2. When an in-memory component gets flushed to disk, the flushed disk component is transferred to the designated replica(s).

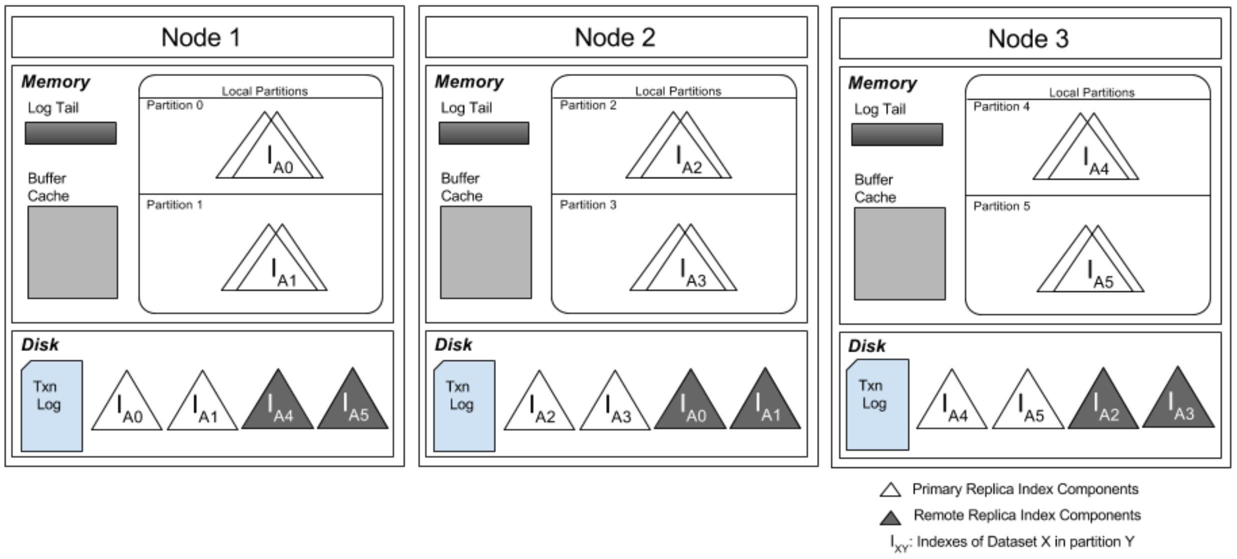

Figure 3.4: Node Controllers with Passive Replication

Partition	Primary Replica Node	Remote Replica Node
0	Node 1	Node 2
1	Node 1	Node 2
2	Node 2	Node 3
3	Node 2	Node 3
4	Node 3	Node 1
5	Node 3	Node 1

Table 3.1: Replica Placement in a three-node cluster with Replication Factor 2

Figure 3.4 depicts a cluster with a passive replication factor of two and Table 3.1 shows the assignment of partitions to node controllers. Node 1's replica is Node 2, Node 2 replicates to Node 3, and similarly, Node 3 replicates to Node 1. A Node is called a primary replica for a partition for which it hosts the in-memory components of a dataset's indexes whose key values hash to that partition. The partitions that have in-memory components of indexes are called the local partitions of that Node Controller. In the diagram, Node 1 is the primary replica for partitions 0 and 1 and Node 2 is a remote replica for partitions 0 and 1. Partitions 0 and 1 are called local partitions on Node 1 and replica partitions on Node 2. Remote replicas store transaction log records generated from operations on the primary replica partitions in addition to the log records that are generated by operations on indexes on its local partitions. This enables a remote replica to recover the state of the in-memory components of indexes on its primary replica partition when the primary replica fails by using the transaction log records to redo operations. Additionally, each remote replica contains immutable disk components that are generated by flushing the in-memory components of indexes in its primary replica. The disk components I_{A4} and I_{A5} in Node 1 are generated when Node 3 flushes those indexes to disk and they are copied over the network to Node 1.

Replicas are placed on nodes based on Chained-Declustering [22] and the replica placement information is shared with the node controllers during runtime (using a static configuration during cluster bootstrap).

3.4 Fault-Tolerance

When a node fails in the cluster, no jobs can make immediate progress without compromising the consistency of the system. AsterixDB is a CP system, so it gives up availability when there is a node failure, but because of passive replication we can recover the state of the failed node locally in a replica without needing any additional information from the failed node. This happens in three steps –

1. The Cluster controller considers all the nodes that are currently active and picks a node to take over the partitions of the failed node. The cluster controller may optionally pick multiple nodes if it has to recover many partitions, which will help with the distribution of the load after failover.
2. A failover node taking over a partition identifies the minimum of the maximum LSNs of all the persistent disk components of the remote replica's partitions.
3. The failover node performs a redo-operation to recover the state of the remote primary's in-memory LSM components.
4. When all failover nodes complete their redo operations, the Cluster Controller updates all necessary state to ensure that jobs in the future will make use of the new partition placement information for their read/write queries and then it sets the state of the cluster to ACTIVE to resume jobs.

When the failed node comes back online and is ready to re-join the cluster, the cluster performs the following operations to get the failed node up to speed on the current state –

1. The Cluster Controller requests a flush operation for all indexes on the node that is currently performing the operations on behalf of the failed node and also on the node controllers that have replicas for the partitions that the failback node was a replica for before its failure.
2. The flushed disk components are copied over to the failback node.
3. The Log Manager enables logging from an LSN greater than all disk component LSNs in the failback node.
4. The Cluster Controller updates the state of partitions and makes the cluster ACTIVE again.

In both cases, the replica connections are re-established since there has been a change in the primary replicas for the partitions of the failed node. The Cluster Controller may decide to reduce the replication factor if there are fewer nodes in the cluster than the number of nodes required to support the originally configured replication factor.

Passive replication has a relatively low overhead in terms of its utilization of CPU cycles and memory [21]. The network is used for communicating log records and the flushed disk components. Components are sent over the network consuming at least as much network bandwidth as the size of the disk components for every flush operation. Disk utilization is slightly higher since the system will have to write the flushed disk components of the remote primary partition to disk in each replica. Passive Replication's modest runtime overhead comes at a cost of increased recovery time, at which point the system first has to replay operations in the transaction log to resurrect an in-memory component. While this time is controllable by switching to a smaller in-memory component size threshold, doing so will potentially give up other benefits of having larger in-memory components that allow for deferring of I/O to larger sequential disk writes at a later time.

Chapter 4

Active Replication in AsterixDB

Active Replication is a new AsterixDB replication protocol that maintains hot-standby partitions on the replicas; that is, it replays log operations on the replica when it receives them instead of doing so later during recovery. This enables the system to rapidly recover from a failure since the stand-by node will already have the state of the in-memory components of its replica and that will minimize the downtime of the cluster when a Node Controller fails. In this chapter, we discuss the design and implementation of Active Replication in AsterixDB.

4.1 Basic Architecture

In contrast to Passive Replication, eagerly replaying the log records will result in keeping the LSM indexes' in-memory components in the primary replica and all its remote replicas. Table 4.1 and Figure 4.1 illustrate the state of the memory and persistent disk components in Active Replication mode with each of the nodes hosting two data partitions with a replication factor of two. Active Partitions are those partitions for which the Node Controller is a primary replica. Similarly, an Inactive Partition is a partition that hosts the in-memory components that shadow the operations on its corresponding remote primary partition and that will be eventually consistent with the remote primary's Active Partition content. In a sense, the inactive partition is eventually consistent with respect to its active partition on the remote primary replica because the remote primary replica partition will have operations that are not synchronously committed to the inactive partition in the replica. The operations represented by remote log records are queued for asynchronous replay, and the committed operation represented by the remote log record is allowed to be visible to the user at the primary copy before it has been replayed on the replicas. (We will discuss potential inconsistencies due to this design further in Section 4.7.)

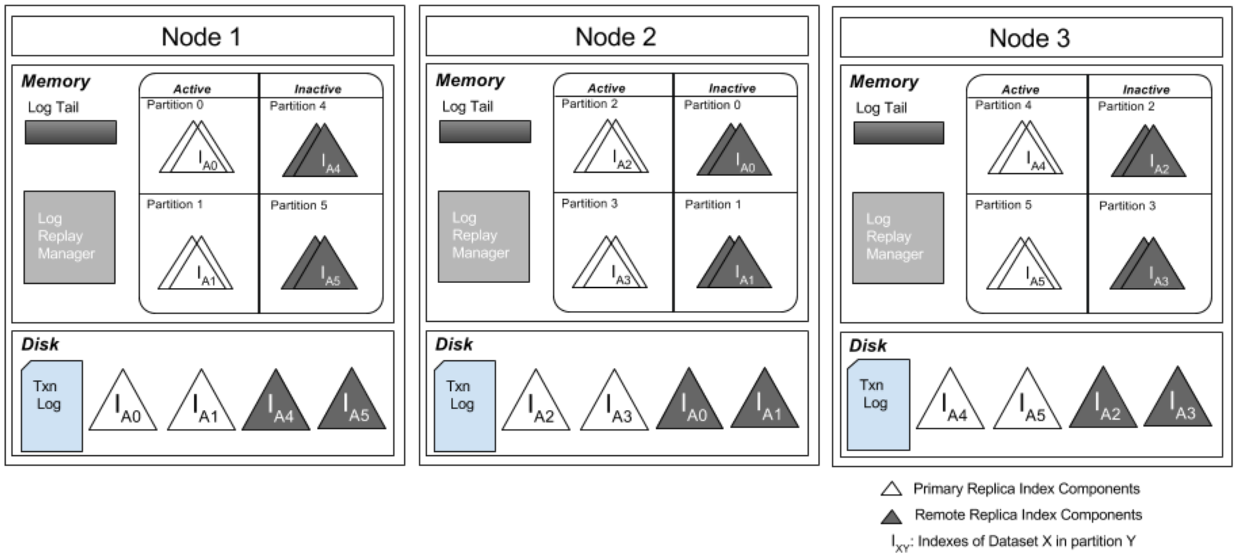

Figure 4.1: Node Controllers running with Active Replication Factor of 2

In the figure, Node 1 hosts partitions 0 and 1 as Active Partitions, and all reads and writes for a record that should belong to partition 0 or 1 will be directed to Node 1. Similarly, partitions 2 and 3 are Active Partitions on Node 2, i.e., Node 2 hosts the primary replicas for partitions 2 and 3. Partitions 0 and 1 are inactive partitions on Node 2 and they are remote replica partitions on Node 1. When a user submits a write (or read) operation of any kind, the operation is forwarded to the active partitions where it should be executed. The corresponding transaction log record generated by the operation on an active partition index is forwarded to all the replicas that hosts the same partition as an inactive partition and they will queue the generated transaction log record for asynchronous replay. The inactive partition contents are flushed by the replica to disk when a flush operation is triggered in the active partition on its primary replica.

Partition	Active Partition Node	Inactive Partition Node
0	Node 1	Node 2
1	Node 1	Node 2
2	Node 2	Node 3
3	Node 2	Node 3
4	Node 3	Node 1
5	Node 3	Node 1

Table 4.1: Partitions and their status in Node Controllers

Each node controller in an AsterixDB cluster is aware of the number of nodes that are in the cluster and the total number of partitions, including their default placement, through a static cluster configuration made available when the cluster is created. After bootstrapping and performing any crash-recovery required, each node controller informs the cluster controller that it is active and ready to accept new jobs. The CC then broadcasts a message to all the nodes currently active in the cluster with information about the node that is now active including the active partitions that it is hosting. The node controller also establishes a TCP connection with the nodes that are replicas for its active partitions. This connection is used to communicate the transaction logs and other replication events and functions when required.

4.2 Log Replication Protocol

Figure 4.2 illustrates the replication protocol lifecycle when a job is submitted and how the replicas receive the transaction logs, and Table 4.2 describes the transaction log entry structure in detail.

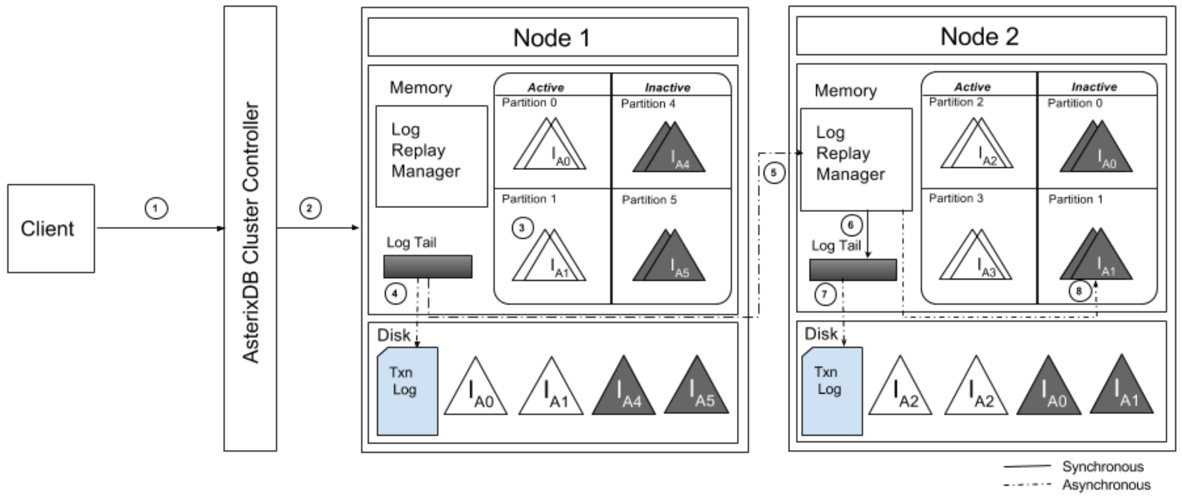

Figure 4.2: Job Execution with Active Replication

A client submits a job to insert a record to the Cluster Controller (1), where it is compiled into a Hyracks job for execution on a node controller (2). The job performs an insertion in the primary replica partition, which involves writing to all the in-memory components of the associated indexes associated with the record's dataset (3). Each insertion into an LSM index generates an associated UPDATE type log record with the new value in that index which is written to the log tail and eventually written out to disk before the commit operation is made visible to the user (4). The primary index is always the first index that gets updated so that subsequent insertions into all secondary indexes can safely refer to the primary key value in the primary key index. Once all the secondary indexes have also been updated and their associated log records have been written to the transaction log buffer, the job creates an ENTITY_COMMIT log record for that record. (The job itself may affect multiple records, but since isolation is only at the record level, the ENTITY_COMMIT log record is the commit operation for the given record.) When the last record is written, the job also issues an overall JOB_COMMIT log record containing the identifiers for the job. The transaction log buffer is never forced to disk and AsterixDB does group commit for all the operations. The user gets a response reporting the status of a job only after the log buffer containing its JOB_COMMIT log record has been safely persisted to disk. Additionally, the transaction log buffer is replicated in batches to each of the partition's replicas by sending the log records over the network to the replicas (5).

Field	Description	Size (Bytes)
Source	Local/Remote: Local logs are generated for active partitions and Remote log records are received from a replica	1
Operation	UPDATE / ENTITY_COMMIT / JOB_COMMIT / JOB_ABORT / FLUSH: The type of operation that generated the log record	1
Job ID	Globally unique monotonically increasing identifier for jobs	4
Dataset ID	The dataset on which the operation was performed	4
Partition ID	The partition of the index that generated UPDATE/ENTITY_COMMIT	4
Primary Key Hash	The Hash of the Primary Key value that decides the partition where this key will reside	4
PK Size	Size of the primary key (PK_SIZE)	4
PK Value	Value of the primary key	PK_SIZE
Resource ID	A globally unique identifier for each in-memory/disk component in a partition	4
New Value Size	Size of the new value in the record, if applicable (NV_SIZE)	4
New Value	Serialized record content	NV_SIZE

Table 4.2: Transaction Log Record structure

When the replica receives a batch of log records, the Log Replay Manager first writes the log records to the log tail (6). While these operations are ongoing, the primary replica will not wait for the remote replica to replay the log records in the replica's inactive partition, as this would otherwise result in a synchronous commit for every record, increasing the latency to include the acknowledgement from the replica. However, when there is a job commit, the primary replica waits for an acknowledgement from each of the remote replicas; an acknowledgement will be sent after the remote replica has persisted the JOB_COMMIT log record in its local transaction log (7). This synchronous job level replication commit operation ensures that all the log records that were sent to the replica have been persisted to disk on the replica without increasing the latency for the individual updates to records. UPDATE and ENTITY_COMMIT log records are eventually written out to disk; that is, the primary replica does not wait for these log records to be persisted in the replicas. The Log Replay Manager eventually performs the operation represented by remote log records on the inactive partition index (8), the details of which are explained in the next section.

4.3 Log Replay

Figure 4.3 shows the details of how the log is replayed at a replica during normal operation. Log records are processed in buffers that arrive through a TCP connection from the remote primary. The incoming log records are first written to the local transaction log with their log source field set to REMOTE (1). Once a remote log record is in the WAL, it is recoverable in case of failure. Each such log record is now sent to a demultiplexer that forwards the log record to a thread responsible for the inactive partition on which it has to be replayed (2). The globally unique partition identifier is a part of every log record, and that information is used to forward the log record to the correct replay manager. Each partition's log replay manager has two buffers – a write buffer that is used to copy the incoming log records (3), and a read buffer that is used to read the payload and then to perform an insert/delete operation on the index with the new value present in the transaction log. The write buffer is consumed from a list of empty buffers that is created during node controller bootstrap. When the write buffer is full, it is added to a backlog queue (4). The replay thread takes buffers from the backlog queue (5) and performs the operation that generated the log record in the inactive partition (6).

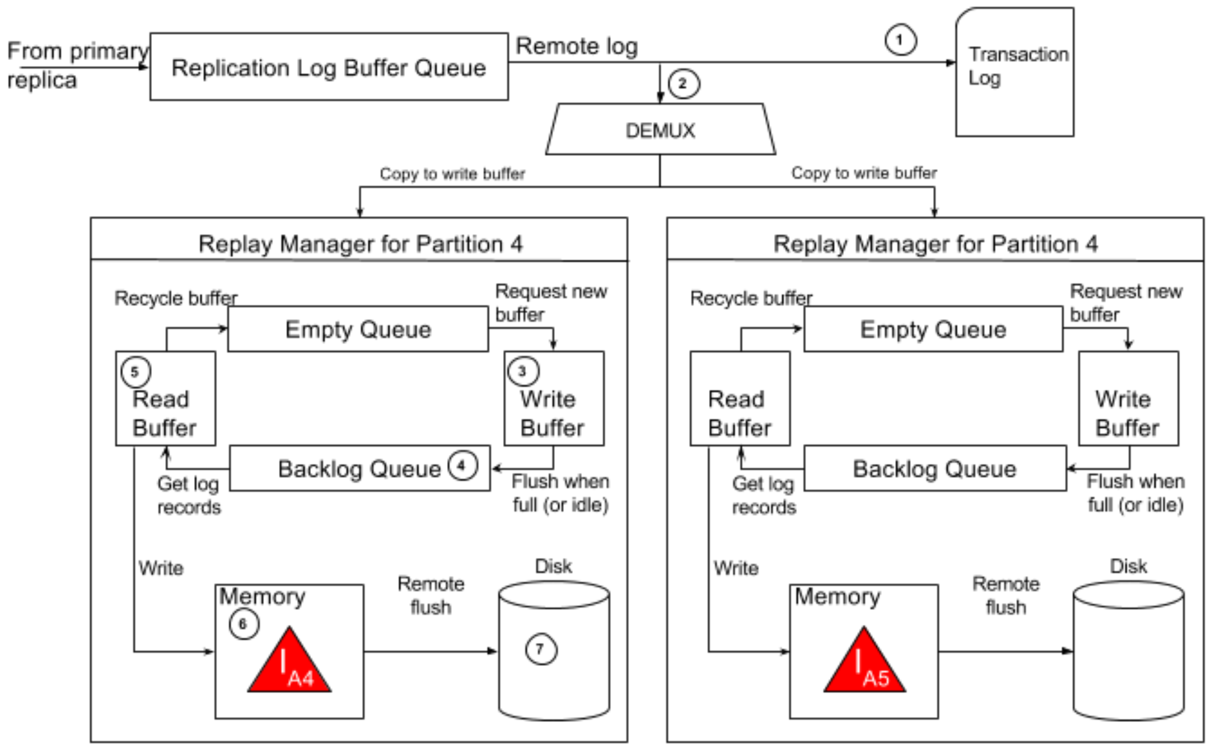

Figure 4.3: Log Replay Management

When all the log records in a read buffer have been replayed, it is recycled through the empty buffer queue. The Replay Manager does not wait until an incoming log buffer is full before sending it to the backlog queue because it would then not be replayed until new operations on the remote primary replica fill up the write buffer and it is flushed to the backlog. To avoid waiting indefinitely for subsequent operations on the index to fill the write buffer, the replay thread pulls the current write buffer into the backlog whenever it goes idle. (This is similar to how the non-replicated AsterixDB transaction log is managed.) Stealing the incoming buffer when the backlog is empty ensures that there are no stale log records waiting in the write buffers to be replayed at a later time when the write buffer becomes full. Each buffer is as large as a log page, and the number of buffers is equal to the number of log pages in the node controller. The backlog cannot exceed the length of all the buffers available in the empty queue at bootstrap. When the queue is full, incoming write buffers will make the remote replica wait before proceeding with the next set of buffers for replication. This is important in order to ensure that the remote primary replica will slow down its own ingestion if the replica is falling behind by the maximum allowed limit.

Since the operations on a replica are generated from the remote primary partition's index, and because AsterixDB does not support reads from the inactive partitions, the replicas do not acquire any locks or run their replay operations in a regular transactional context. For consistency, a transaction running on a primary partition of a dataset in AsterixDB will lock the primary key on the dataset and then perform the operation. Record-level isolation and the serial backlog make the transaction manager redundant for the inactive partitions. Not locking on inactive partitions has the advantage of making the write path for the index for replay operations faster than the write path for regular operations on the index, as active partition operations can be concurrent and therefore require locking. A serial schedule is already ensured by the locking done at the active partition, and the log buffer will contain operations corresponding to that serial schedule. This serial history is available in the replication log buffer queue and the protocol can safely demultiplex it by partition. Note that replay operations on Partition 5 can be interleaved differently with operations from Partition 4, but the ordering of operations within a given partition will be consistent with its primary replica. (A lock manager would become necessary if the protocol introduced any additional level of concurrency while writing within an inactive partition. It currently has only a single writer per inactive partition, and there cannot be any conflicting operations within a partition since they are already serialized.)

4.4 Index Management

When a new dataset is created, its active partitions create the necessary metadata information along with a set of persistent files that describe the index. When such an operation is performed, the node controller also creates a Replication Job that sends to each replica the LSM index properties and other information required to create the index structures in the replicas. Any persistent files that are created as a part of the create index operation are copied over the network to each of the replicas.

Active partitions' indexes are managed by each node controller by monitoring their on-going operations and keeping a watch on the memory utilization of the index partitions so that the NC as a whole doesn't exceed a global limit allocated to the indexes of each dataset. When the per-NC footprint of a given dataset exceeds the threshold allotted for in-memory components, the in-memory components of each index in the dataset are asked to flush themselves to disk once any on-going operations are complete. The flush operation waits for any/all on-going record modifications to commit, because LSM disk components are immutable and there should never be any record in an immutable component for which there is no associated ENTITY_COMMIT log record in the transaction log (i.e., a NO STEAL policy is enforced).

Figure 4.4 illustrates how a flush operation is handled in the replicas. Node 1 is a primary replica for partitions 0 and 1, and Node 2, being a replica of Node 1, hosts these partitions as inactive partitions. When the dataset exceeds its in-memory component threshold, the Active Index Manager triggers a flush operation by first writing a FLUSH log record to the log tail (1). When the log is persisted to disk, the Index Manager asks Indexes I_{A0} and I_{A1} to flush their associated in-memory components (2). Since each in-memory component is double-buffered, the request switches the buffer for new operations to write to the second buffer, and the first buffer is asynchronously written to disk. The log tail in Node 1, which contains the FLUSH log record, is sent to its replica, Node 2 (3). Node 2 writes the remote FLUSH log record to its local transaction log and copies the FLUSH log record to each inactive partition's write buffer (4). The write buffer gets added to the backlog and is eventually dequeued from the backlog into a read buffer (5). The read buffer is then read sequentially and the corresponding operations are performed on the associated index replica. When a flush log record is read from the read buffer, the replay manager asks the inactive partition index associated with the read buffer's partition to flush (6). The flush operation for the index happens in a manner similar to the active index flush operation by switching to a second in-memory component for new incoming writes and asynchronously writing the previous in-memory component of that index to disk (7). Although the indexes I_{A0} and I_{A1} will begin and complete their flush operations at different times, the contents of the immutable replica will ultimately be the same as those of the immutable primary component on the primary replica's node.

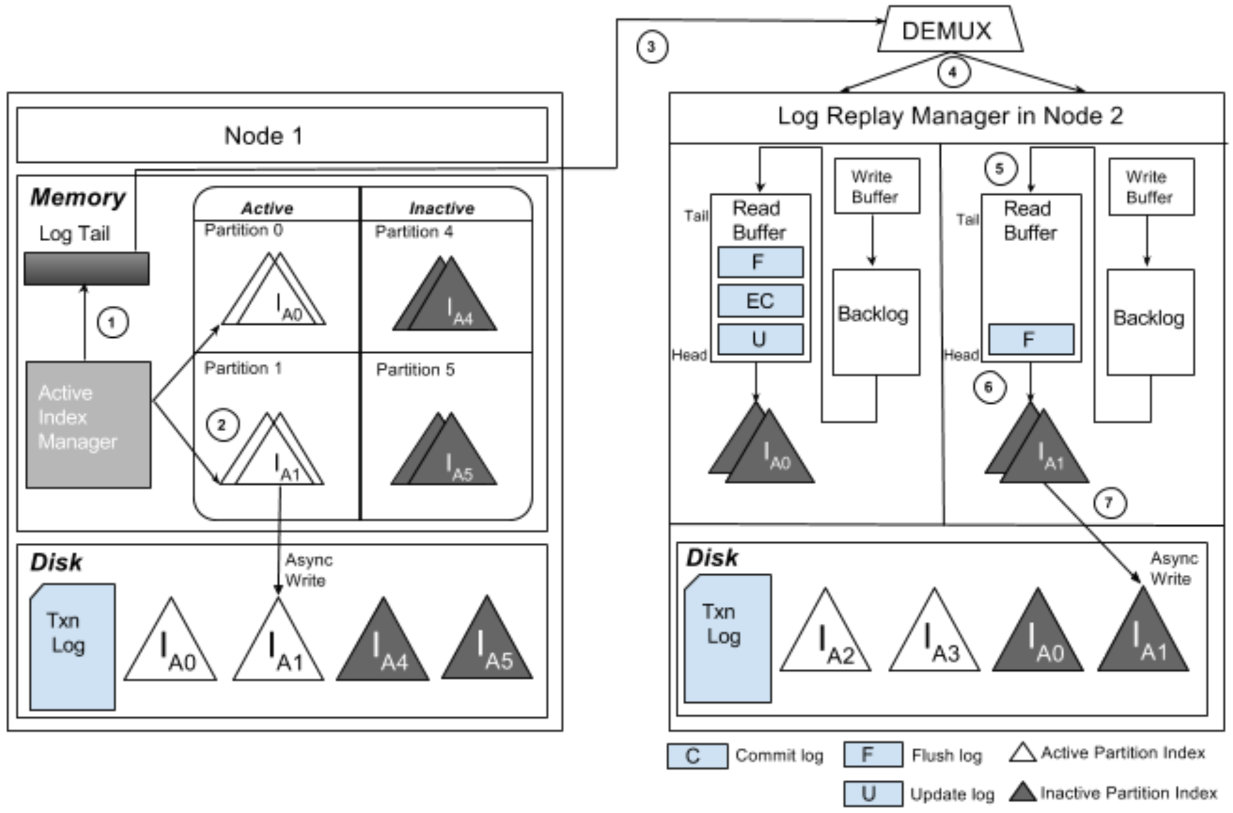

Figure 4.4: Handling Flush operations for inactive partition indexes

Notice that the flush operations for the replicas are triggered independently, per partition, instead of flushing all inactive partition indexes that belong to the dataset. This is due to the fact that the backlog of operations can be different for each of the partitions in the replica. In the diagram, the backlog of operations for partition 0 and 1 in Node 2 are different and, when the FLUSH log record is read from the read buffer in the Log Replay Manager, the state of the in-memory component will be the same as that of the active partition's in-memory component on the remote primary replica. Adding the flush request to each partition's queue ensures that each partition will replay the same operations as the active partition did at the point when the flush request is dequeued. In other words, when the flush request is read from the replay-read buffer, there will not be any pending dirty writes on the inactive replica partition due to the fact that that invariant is already checked by the primary replica. (We will discuss some implications of this design in more detail in Section 4.7.)

4.5 Fault-Tolerance: Node Failover

AsterixDB detects failures when there is a network partition between Node Controllers. When there is a partition in the network, log records cannot be sent to the replicas, and this will block incoming write requests from being serviced. When a replica detects that the connection to its primary is broken, it informs the Cluster Controller about the link failure and starts a timeout to wait for the failed node to rejoin and re-establish a connection. When the timeout elapses, the Cluster Controller updates the state of the cluster to INACTIVE and suspends all jobs until a candidate has been identified to take over the active partitions that were hosted on the failed node. Candidate selection may consider any previous failures and the number of partitions hosted on each candidate node; it selects a node controller to take over the newly unavailable partitions. The chosen node controller will be one of the hot-standby replicas. When the failover node receives a partition takeover request from the cluster controller, it flushes all of its log replay buffers for the relevant inactive partitions and waits for the replay threads to replay any remaining log records that are in the backlog. Once the backlog is empty, all of the partitions in the failover node will be in the same state as they were on the Node Controller that hosted the active partitions. The node controller the marks its copy of the partitions as active and asks the cluster controller to refresh its state again. The cluster controller then updates its internal state about active partitions, and broadcasts this information to all of the active node controllers in the cluster. The replica connections are reestablished, and the cluster can resume processing of incoming jobs.

With Active Replication, the recovery process after the timeout does not involve any replay operations except for waiting for anything remaining in the backlog to be completed. Furthermore, when waiting for the timeout for a failed node to rejoin the cluster, each node controller can continue performing operations from its replay backlog, and each node controller gets at least the time needed to detect the failure to catch up with their primary replicas. The total time to recover in the worst case is bounded by the timeout to mark a node as failed plus the time needed to replay all the log records in the backlog. Since the length of the backlog is fixed, the total time to recover will not be more than the time to insert the number of records in the backlog. In our experiments, to be discussed in Chapter 5, we have observed that the backlog is almost always empty by the time that the recovery of a node begins, which means that the total downtime for the cluster is determined primarily by the timeout to detect a failure plus the time needed to reassign a new node controller to take over.

In the event that there are now fewer nodes than the requested replication factor, the Cluster Controller could decide to reduce the replication factor to the highest number lower than the requested number that can be supported with the number of remaining active nodes in the cluster.

4.6 Fault Tolerance: Node Failback

The failback process for a rejoining NC in Active Replication involves deleting its entire persistent state and fully recovering that state from the replica that is currently active for its partitions by transferring the disk components of all indexes in partitions that were previously owned by the failing back NC before its failure. When a node tries to rejoin the cluster, the cluster controller first identifies the node which currently hosts all the active and inactive partitions that were previously assigned to the failing back node. In general, the recovery plan constructed by the Cluster Controller may contain different nodes from which to remotely recover the state.

Partition	Partition State	Initial Location	Failover from Node 1	Failback to Node 1
0	Active	Node 1	Node 2	Node 1
	Inactive	Node 2, Node 3	Node 3	Node 2, Node 3
1	Active	Node 2	Node 2	Node 2
	Inactive	Node 3, Node 1	Node 3	Node 3, Node 1
2	Active	Node 3	Node 3	Node 3
	Inactive	Node 1, Node 2	Node 2	Node 1, Node 2

Table 4.3: Partition-Node assignments initially, after failure of Node 1, and after failback of Node 1 with initial Replication Factor 3

Table 4.3 shows an example of the state of the cluster before failure, after failover, and finally after failback completion. (The failover process was discussed in the previous section.) When Node 1 informs the cluster that it wants to rejoin the cluster, the node has to obtain recovery information from both Nodes 2 and 3. Node 1 has to recover its assigned active partition 0 from Node 2, which is currently maintaining partition 0 in the active state. Additionally, Node 1 has to recover the states of partitions 2 and 3 to host them as inactive partitions after failback. Node 1 receives the persistent components of partition 2 from Node 2 and those of partition 3 from Node 3. The Cluster Controller then sends a request to the Node Controllers to flush their in-memory components for the partitions that the failback node must receive. Once these components have been flushed, the persistent disk components are transferred according to the recovery plan constructed by the Cluster Controller. Once the disk components have been transferred, the log manager starts logging from the maximum LSN from among all of the disk components. The nodes will also reestablish all of their replication connections since the placement of active and inactive partitions is now different. The Cluster Controller will know the state of the partition assignments post-failure so that incoming jobs will use the latest information for replication and regular job execution.

4.7 Design Discussion

In this section, we discuss the Active Replication protocol's implications on the consistency of the data and other potential issues.

4.7.1 Data Consistency

As described in the previous chapter, AsterixDB's isolation is at the record level. A record is considered committed once a corresponding ENTITY_COMMIT transaction log record is safely on disk on the active partition's node controller. AsterixDB's transaction model does not wait for a JOB_COMMIT before allowing another job to read the inserted record. Once the ENTITY_COMMIT log record is written to disk, the locks acquired are released and the record will become visible for reading in the primary replica. However, if the node fails after a reader sees the new record but before the corresponding transaction log record is sent to the replica, the replica will not replay this operation and the new value will not be visible to users after failover, despite the fact that it was previously (albeit briefly) visible before the failure. However, if the record is a part of a larger job that completes its execution, the job level JOB_COMMIT is synchronously replicated and will ensure that all previous log records are persisted to disk both on the primary and all its replicas. Note that since there is no limit on the number of records that a given job can insert/read, it is possible that a failure during execution before a JOB_COMMIT can result in such a read inconsistency in the replica. This can be mitigated by either having fewer records per job or by periodically "checkpointing" the job's progress with respect to its replicas by synchronously replicating ENTITY_COMMITs at regular intervals for long running jobs.

In a cluster with a replication factor greater than two, it is possible for a node controller to fail after replicating its transaction log record to a subset of replicas but not all replicas. Depending on the replica that is chosen as the failover node, the replicas will either have additional operations that were not sent to every other replica or they will have some operations missing that are visible on other replicas. This can be mitigated by letting the cluster controller pick the node with the highest known LSN in the replication buffers among all candidates. Alternatively, if a different replica is chosen, a replica that contains additional log records can replicate those records to the chosen primary replica to replay during failover.

4.7.2 Index Memory Management

The lifecycles of the in-memory index components of an inactive partition are driven by the node controller that hosts the corresponding active partition for that index. All active and inactive partition index resources are managed at the dataset level. When an index crosses its threshold for in-memory components due to some relatively larger inactive partition memory component for an index, the node controller will only flush the indexes in its active partition memory component. When such a flush occurs, the active partition's memory component may actually be utilizing less memory than the inactive partition, but the active partition indexes are the only indexes that can be flushed safely (by checking for on-going operations on that index on the same node controller). This can lead to disk components that are irregularly sized and that are suboptimal for reading, and this can lead in turn to smaller than expected sequential writes on the disk because of their less-than-ideal size. This could be mitigated by managing the index resources independently for active and inactive partitions or by locally tracking the status of operations on the inactive partition indexes so that flushes can be triggered locally (yet safely) without waiting for a message from the remote primary replica for that partition.

4.7.3 Flow Control for Replay Operations

The replay operations on an inactive partition are bounded by the size of the buffers in the Log Replay Manager's empty queue when the node controller is started. If a remote primary replica has additional resources that cause its throughput to be higher than any of its replicas, the remote primary will have to wait for the replicas to free up buffers for replay operations, which will reduce the primary replica's throughput, thus allowing the replicas to catch up. Additionally, such a reduction in the throughput on the primary replica ensures that a replica node controller can control the memory utilization of the replay operation backlog in the remote replicas. This controlled replay backlog size is important to ensure that the recovery time for a node controller does not involve replaying an unbounded number of operations from the backlog, which could take longer than the time required to detect a failure.

Chapter 5

Initial Performance Evaluation

In this chapter, we present experimental results to quantify the performance impact of maintaining Active Replicas in AsterixDB. Section 5.1 outlines the experimental setup and then describes the dataset used for all the experiments. The rest of the chapter explores the performance impact of Active Replication under different scenarios and compares its performance with both an AsterixDB cluster running without replication and with Passive Replication.

5.1 Experimental Setup

The experiments reported in this section were performed on a cluster consisting of 6 Node Controllers and 1 Cluster Controller. Each of the nodes has two Opteron 2212HE processors, 8 GB of DDR2 RAM, and two 1 TB I/O devices running at 7200 RPM. The Node Controllers each use one disk for the transaction log and the other disk to host a single active data partition. The disk buffer cache is 2.5GB, and the indexes have an in-memory component limit of 2GB, which is the limit for both the active and the shadow buffer of the in-memory component per dataset. The node controller has 5 log tail buffers of 6MB each plus 5 reusable log tail replication buffers that are used to send the locally-generated transaction log records to the replicas. The replication log buffer block size is 2KB when sending data to the replicas. The Replay Manager starts with 5 empty pages in its empty queue, with each page being equal to 6MB, the size of a log page per partition. These empty queues are used by the backlog to queue operations in the remote log records for replay.

```
create dataverse Gleambook;
use dataverse Gleambook;

create type EmploymentType as {
      organization: string,
      start_date: date,
      end_date: date?
}

create type GleambookUserType as {
      id: int64,
      alias: string,
      name: string,
      user_since: datetime,
      friend_ids: {{ int64 }},
      employment: [EmploymentType]
};

create dataset GleambookUsers(GleambookUserType)
primary key id;
```

Figure 5.1: DDL used to create the dataset for experiments

Number of Records (Millions)	Size
1	384 MB
5	1.9 GB
10	3.8 GB
15	5.8 GB
20	7.8 GB

Table 5.1: Dataset Size

Figure 5.1 shows the DDL used to create the dataset. The datasets used for the experiments were generated using SocialGen [23]. SocialGen, as used here, distributes the data in ADM format into 6 partitions, one for each node controller. The file containing the data to be ingested in each node controller is placed on the same I/O device as the data partitions.

The experiments are executed on different dataset sizes. Table 5.1 shows the dataset sizes (as measured on disk) and the corresponding number of records present in each dataset.

5.2 Results and Analysis

5.2.1 Ingestion Time

AsterixDB supports high speed data ingestion into a dataset with feeds by having a long-running continuous insertion job that waits for data to arrive from the feed's source [24]. In this experiment, we have created a file-feed job that consumes data in ADM format in a file on disk. The dataset described in the previous section is partitioned, and one partition of the dataset is available in each Node Controller. We evaluate the time taken to complete ingesting the data from the contents in the partitions of all the Node Controllers.

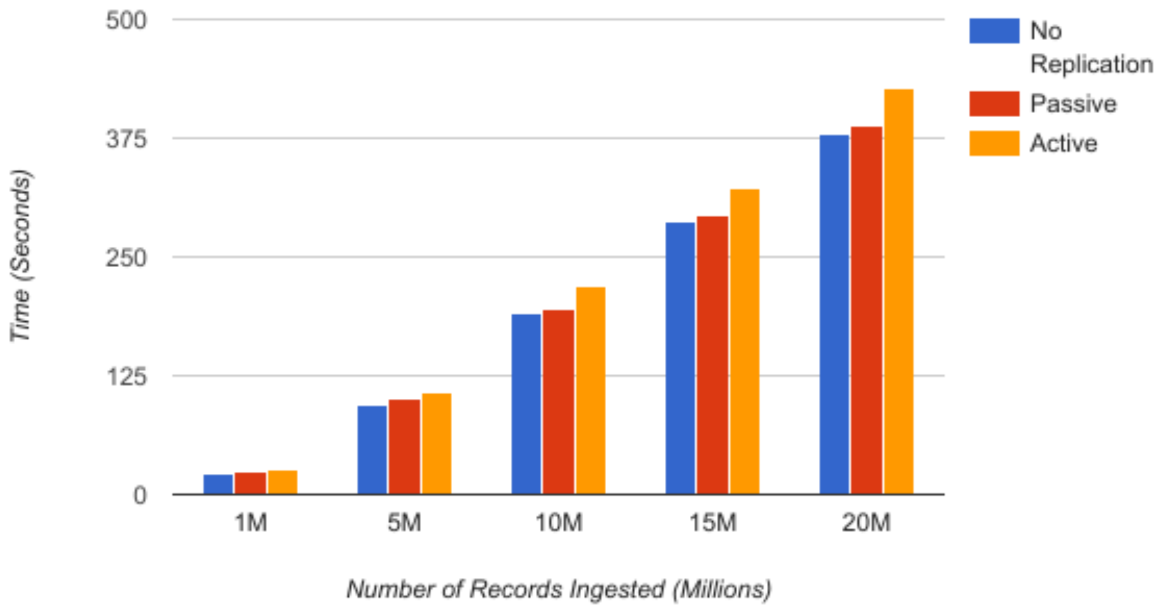

Figure 5.2: Data Ingestion Time with Replication Factor 2

Figure 5.2 shows the results from ingesting datasets with no replication, passive replication, and active replication with a replication factor of two. The average data ingestion time increase due to Active Replication compared to no replication is between 10 and 15%. Passive Replication, on the other hand, has a much lower overhead, with less than a 5% increase in the ingestion time when compared to the system's ingestion performance without replication. This is because with Passive Replication, the system only writes additional log records received from replicas into the transaction log file. In contrast, with Active Replication, in addition to writing transaction log records from every active replica, the cluster also has to perform twice the number of (in-memory) insertion operations on the node. Additionally, since we have compared the performance while not changing the memory allocation per dataset for all three cases, Passive Replication and a cluster with no replication will have twice the amount of memory per index because they only work with the active partition. This results in larger sequential writes to disk compared to Active Replication, where there will be smaller sequential writes to the disk, and this happens more frequently than with passive replication. However, the amount of data written to disk will be the same with both active and passive replication; the source of the data to write for passive is the network, while for active, it is memory.

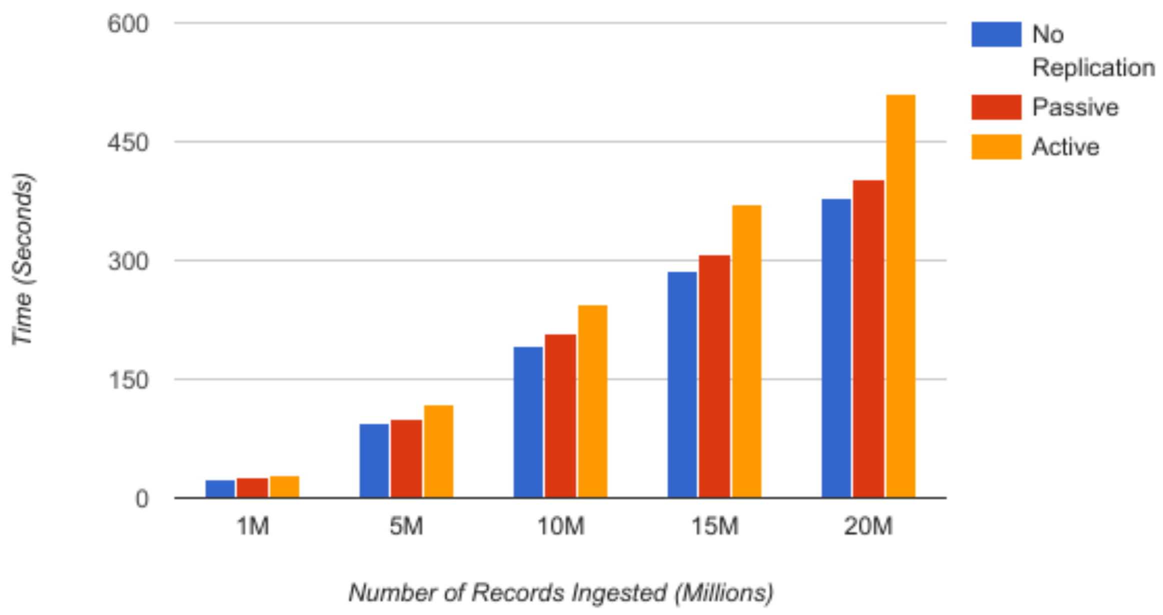

Figure 5.3: Data Ingestion Time with Replication Factor 3

Figure 5.3 shows the results of the same experiment with a replication factor of 3. With a Replication Factor of 3, a cluster using Active Replication will have one third of the memory available per dataset because each of the other two thirds of the memory budget for a dataset will be used to host inactive partitions on which to replay operations from other node controllers in the cluster. This translates to more writes to disk, with smaller memory components being written, since the index is managed at the dataset level and includes both the active and inactive partitions for that dataset.

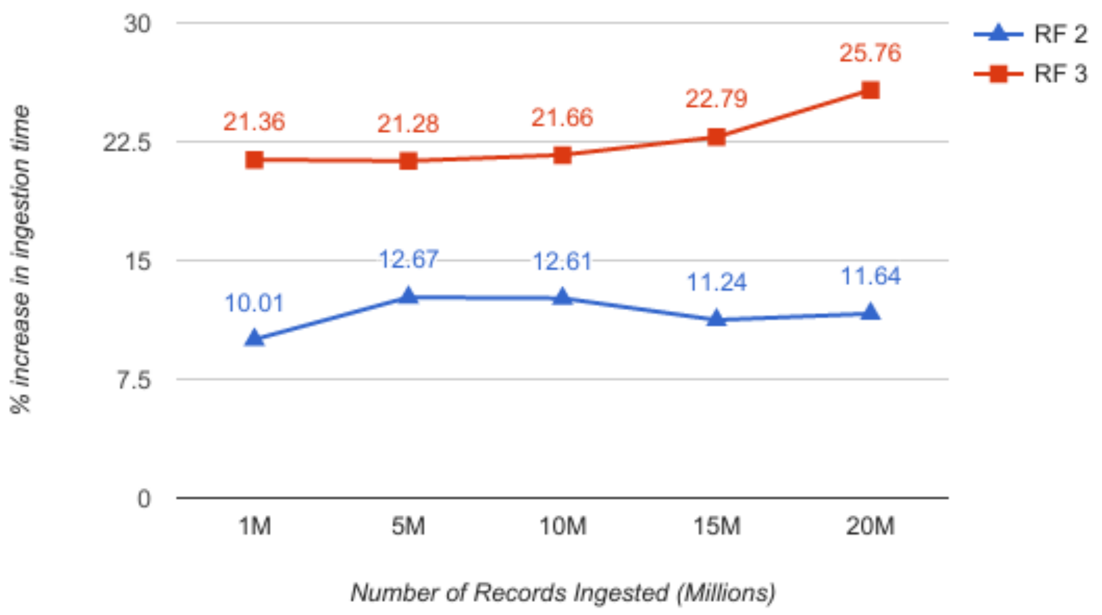

Figure 5.4: Percent increase in ingestion time with Active Replication factor 2 and 3

Figure 5.4 shows the relative slowdown during ingestion incurred due to Active Replication compared to a cluster without replication. With a replication factor of two, we have observed an average 11% slowdown and 22% with a replication factor of three. The low overhead of in the ingestion time can be explained by the fact that for the replica indexes, we do not maintain the same immediately readable transactional semantics required for insertion into active partitions, even though we are inserting three times the number of records per node with a replication factor of three. This lack of real-time transactional semantics means that there is no need to check for concurrent operations on the same record and that there will be no contention due to the lock manager being in the write path for inactive partitions.

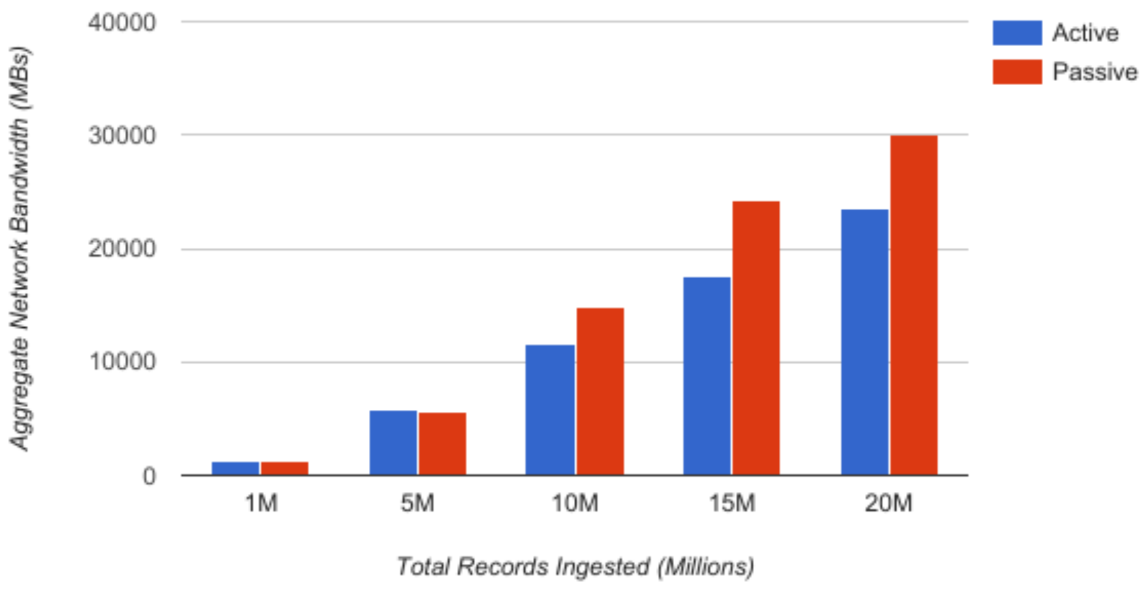

Figure 5.5: Aggregated Network Bandwidth with Replication Factor 2

Figures 5.5 and 5.6 show the total network utilization in the cluster with Passive and Active Replication factors of two and three (respectively) during ingestion. This was measured by initially checking the total number of received and transmitted bytes at each network interface of Node Controllers after creating the dataset and then again after the ingestion is complete. The graph represents the sum of transmitted and received bytes at the network interface during ingestion from all Node Controllers in the Cluster.

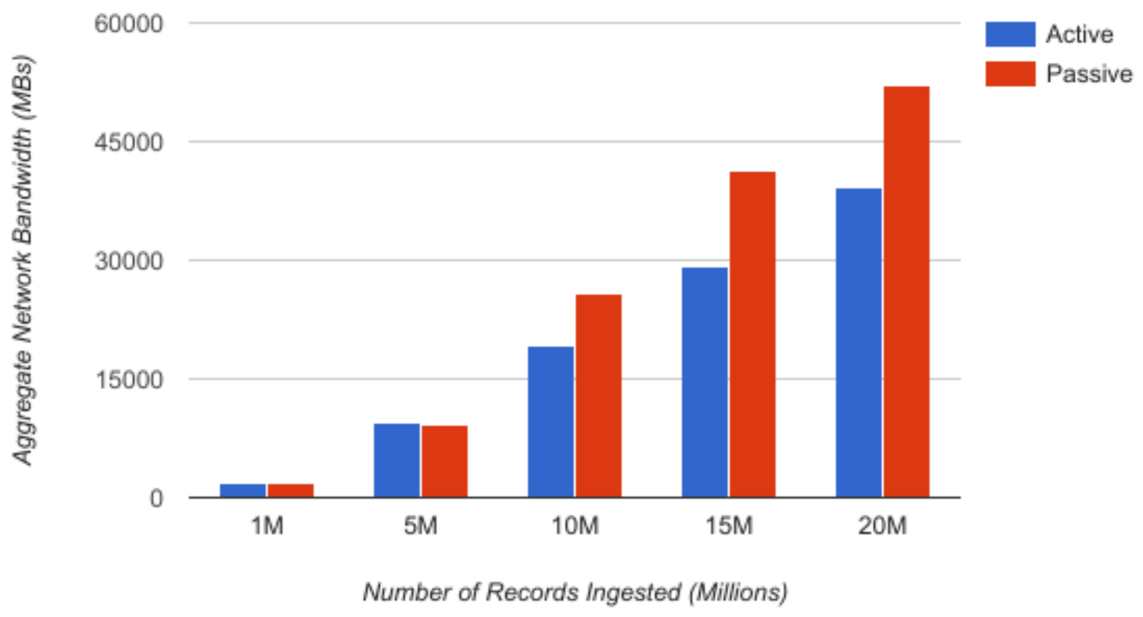

Figure 5.6: Aggregated Network Bandwidth with Replication Factor 3

During ingestion of 1 and 5 million records, there are no flush operations in the cluster with Passive Replication so the total network utilization is the same as with Active Replication. Whenever there is a flush operation in Passive Replication, the immutable disk components are transferred over the network to all replicas, thus consuming network bandwidth to copy the components over the network. However, in Active Replication, we free up the network by not having the flushed disk components sent over the network; however, we still have to do a sequential write operation locally on disk when there is a flush request from the remote primary replica for that node. Additionally, when there are merge operations, the merged component is also sent to the replicas. These merge operations become bigger during the lifecycle of the cluster if we continue the ingestion processes, and the same data (albeit with a different structure) will be sent to the replicas multiple times due to the merge operations.

5.2.2 Failover Time

Active Replication maintains its in-memory components in order to have a near-constant time to failover to a replica node to resume processing of queries. In this experiment, we first ingest our experimental dataset and then kill a single Node Controller. We record the failover time with Active Replication and compare it against that of a cluster running with Passive Replication. The timeout to allow a partitioned node to rejoin the cluster is set to 60 seconds. After this timeout, the Cluster Controller asks a replica node to takeover the partitions of the failed node. We record the time it takes for the cluster to change the status to ACTIVE again after the timeout. Figure 5.7 shows the results.

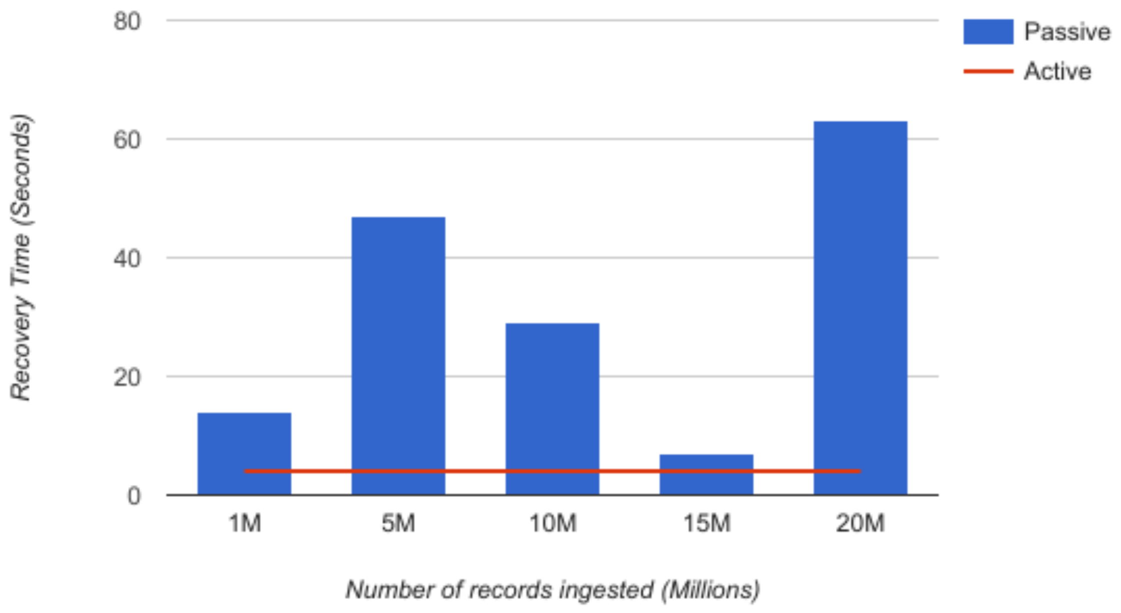

Figure 5.7: Recovery time for a cluster with Active/Passive Replication factor of 2

The failover process in Active Replication first empties the replay backlog by replaying any pending operations and then changes the failed-over partition state to Active. The constant 4-second failover time for Active Replication is because, during the timeout needed to detect a failure, each node controller can continue working towards emptying its backlog of operations and manages to catch up with the primary replica before the 60-second timeout; this allows the replacement node to take over instantly. The additional 4-second delay after the timeout involves communication with the Node Controllers about the state of the partitions after the failure of a Node Controller and reestablishing connections to continue replication from the new active partition.

Passive Replication has to replay the transaction log records received from the failed remote primary before the failure to first reconstruct in-memory state of the database. This involves sequentially reading through the transaction log records and replaying the operations into a new partition. Since the in-memory components could have different fill factors at the time of a failure, the amount of work that needs to be done during failover varies. If there is a failure after a checkpoint where the disk components of the active partitions have been copied over to the replicas, the failover time can be as low as that of Active Replication. However, if a node fails when its in-memory component size is almost full, the recovery time will be proportional to the size of the in-memory component and the time that it takes to (re)fill the configured in-memory component size.

5.2.3 Failback Time

In this experiment, we restart the node that was killed in the previous experiment and record the time that it takes to fully recover and rejoin the cluster. Figure 5.8 shows the time for failback for different dataset sizes. The time to recover for 20 million records is higher for Active Replication because the replica node will have to first flush its indexes and only then start to copy the generated disk components, whereas in Passive Replication, the flush operation would have already been completed during the failover process.

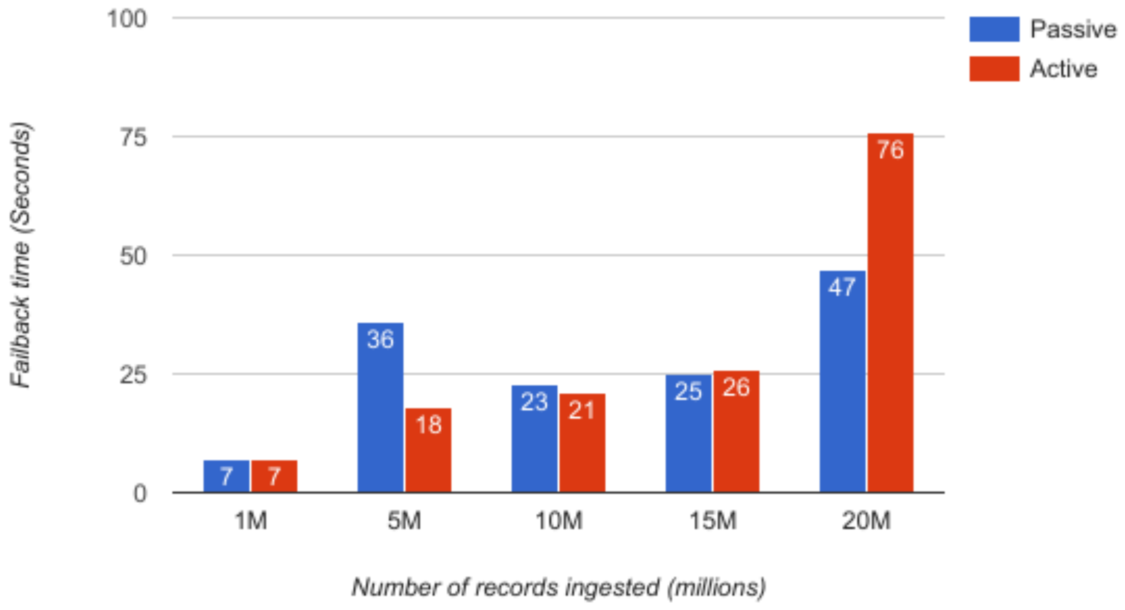

Figure 5.8: Failback Time for a cluster with Active/Passive Replication Factor of 2

5.2.4 Query Response Time

In this experiment, we first ingested the 20M-record dataset using file feeds in a cluster with Passive Replication and in one with Active Replication, both with a replication factor of two. We then start another insertion job that reads through the ingested dataset and copies all records into a new dataset to generate update traffic requiring replication activity. While this happens, we run the query shown in Figure 5.9 several times on the initially ingested dataset and report the average response time. Figure 5.10 shows the query response time results. In Passive Replication, the index components can occupy a larger fraction of memory, thus increasing the search fraction that happens in memory since the Node Controllers have to maintain only the active partitions. Active Replication will have both active and inactive partitions of the new dataset in memory, with a smaller fraction of the initially loaded dataset in memory when we run the query, thus increasing the time required to execute the search query. (AsterixDB can be tuned to perform better on reads by increasing the amount of buffer cache available, but in our experiments here, the configuration has been optimized for writes, with a comparatively bigger fraction of the memory being allocated to the indexes' in-memory components.)

```
use dataverse Gleambook;
let $count := count(for $record in dataset GleambookUsers
                     where $v.id < 100
                     and $v.id < 100000000
                     return $v)
return $count
```

Figure 5.9: Range Query used during ingestion

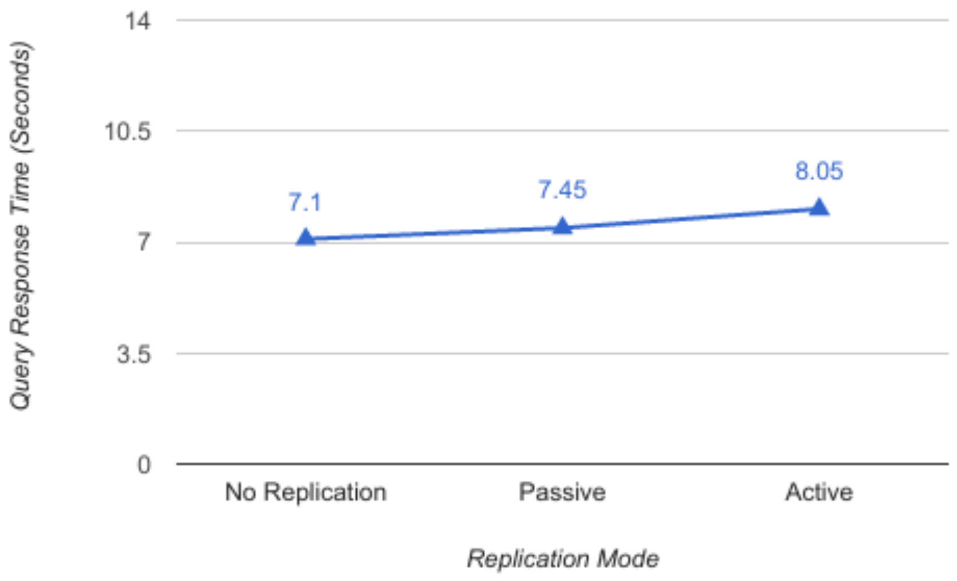

Figure 5.10: Query Response Time during Ingestion

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis, we have described a new replication protocol that minimizes the downtime of an AsterixDB cluster by eagerly replicating a Node Controller's state to one or more replicas. We described the new protocol and explained how it asynchronously replays the operations on replicas without compromising AsterixDB's existing record-level consistency model. We described the index management and other potential issues that may arise from the remote management of indexes. We followed that by describing how fault-tolerance with near-constant failover time can be achieved with Active Replication. Finally, we presented an initial performance evaluation aimed at quantifying the impact of Active Replication under different replication factors and we compared its performance to both that of a cluster with Passive Replication and that of one without replication.

6.2 Future Work

6.2.1 Weakly Consistent Querying

The Active Replication protocol does not allow reads to happen from inactive partitions of a Node Controller. This is because the inactive partitions shadow the operations of their corresponding primary replica partition asynchronously, and allowing them to be read could thus lead to inconsistent reads. It would be possible to redirect reads to shadow partitions by supporting a query extension where users can explicitly ask for any true value of a record and not necessarily the latest version of the record. Additionally, it could be possible to quantify the inconsistency in the response of such queries based on the size of the backlog of replay operations during execution of the query.

6.2.2 Delta Recovery

Recall that during failback, the failback Node Controller in Active Replication destroys its local state and recovers the persistent state from active NCs that are hosting the failback node's partitions after its failure. This can be expensive when there are frequent failures of a Node Controller, i.e., where it fails and rejoins shortly after failover, leading to the cluster spending a large amount of time in keeping the failing-back Node Controller in sync with the failover node. The protocol could potentially be modified to instead request transaction log records that were generated since the time it failed in order to perform recovery by only replaying those operations that were performed since its previous failure.

Bibliography

[1] B. Schroeder and G. Gibson, "Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you?," *FAST,* vol. VII, pp. 1-16, 2007.

[2] S. A. Moiz, P. Sailaja, G. Venkataswamy and S. N. Pal, "Database replication: A survey of open source and commercial tools," *Database,* vol. 13, no. 6, 2011.

[3] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM Computing Surveys (CSUR),* vol. 15, no. 4, pp. 287-317, 1983.

[4] S. B. Davidson, H. Garcia-Molina and D. Skeen, "Consistency in a partitioned network: A Survey," *ACM Computing Surveys (CSUR),* vol. 17, no. 3, pp. 341-370, 1985.

[5] P. O'Neil, E. Cheng, D. Gawlick and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica,* vol. 33, no. 4, pp. 351-385, 1996.

[6] N. Budhiraja, K. Marzullo, F. B. Schneider and S. Toueg, "The primary-backup approach," *Distributed systems,* vol. 2, pp. 199-216, 1993.

[7] J. Gray and A. Reuter, Transaction processing: concepts and techniques, Elsevier, 1992.

[8] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein and I. Stoica, "Highly available transactions: Virtues and limitations," in *Proceedings of the VLDB Endowment*, 2013.

[9] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme and G. Alonso, "Database replication techniques: A three parameter classification," in *Reliable Distributed Systems, 2000. SRDS-2000. Proceedings The 19th IEEE Symposium on*, 2000.

[10] D. DeWitt and J. Gray, "Parallel database systems: the future of high performance database systems," *Communications of the ACM,* vol. 35, no. 6, pp. 85-98, 1992.

[11] V. Borkar, M. J. Carey and C. Li, "Inside Big Data management: ogres, onions, or parfaits?," in *Proceedings of the 15th international conference on extending database technology*, 2012.

[12] E. A. Brewer, "Towards robust distributed systems," in *PODC*, 2000.

[13] P. Hunt, M. Konar, F. P. Junqueira and B. Reed, "ZooKeeper: Wait-free Coordination for Internet-scale Systems," in *USENIX annual technical conference*, 2010.

[14] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall and W. Vogels, "Dynamo: amazon's highly available key-value store," *ACM SIGOPS operating systems review,* vol. 41, no. 6, pp. 205-220, 2007.

[15] "Apache AsterixDB," [Online]. Available: http://asterixdb.apache.org.

[16] V. Borkar, M. Carey, R. Grover, N. Onose and R. Vernica, "Hyracks: A flexible and extensible foundation for data-intensive computing," in *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, 2011.

[17] K. W. Ong, Y. Papakonstantinou and R. Vernoux, "The SQL++ query language: Configurable, unifying and semi-structured," *arXiv preprint arXiv:1405.3631,* 2014.

[18] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. Borkar, Y. Bu, M. Carey, I. Cetindil, M. Cheelangi and K. Faraaz, "AsterixDB: A scalable, open source BDMS," *Proceedings of the VLDB Endowment,* vol. 7, no. 15, pp. 1905-1916, 2014.

[19] "JSON," [Online]. Available: http://www.json.org.

[20] S. Alsubaiee, A. Behm, V. Borkar, Z. Heilbron, Y.-S. Kim, M. J. Carey, M. Dreseler and C. Li, "Storage Management in AsterixDB," *Proceedings of the VLDB Endowment,* vol. 7, no. 10, pp. 841-852, 2014.

[21] M. Al Hubail, "Data Replication and Fault Tolerance in AsterixDB," 2016.

[22] H.-I. Hsiao and D. J. DeWitt, "Chained declustering: A new availability strategy for multiprocessor database machines," in *Data Engineering, 1990. Proceedings. Sixth International Conference on*, 1990.

[23] "SocialGen," [Online]. Available: https://github.com/pouriapirz/socialGen.

[24] R. Grover and M. J. Carey, "Data Ingestion in AsterixDB," in *EDBT*, 2015.