Replicated Training in Self-Driving Database Management Systems

Gustavo E. Angulo Mezerhane

CMU-CS-19-129

December 2019

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Andrew Pavlo, Chair
David G. Andersen

Submitted in partial fulfillment of the requirements for the degree of Master of Science.

Keywords: Database Systems, Replication, Machine Learning

Para mis padres Gustavo y Claret

Abstract

Self-driving database management systems (DBMSs) are a new family of DBMSs that can optimize themselves for better performance without human intervention. Self-driving DBMSs use machine learning (ML) models that predict system behaviors and make planning decisions based on the workload the system sees. These ML models are trained using metrics produced by different components running inside the system. Self-driving DBMSs are a challenging environment for these models, as they require a significant amount of training data that must be representative of the specific database the model is running on. To obtain such data, self-driving DBMSs must generate this training data themselves in an online setting. This data generation, however, imposes a performance overhead during query execution.

To deal with this performance overhead, we propose a novel technique named Replicated Training that leverages the existing distributed master-replica architecture of a self-driving DBMS to generate training data for models. As opposed to generating training data solely in the master node, Replicated Training load balances this resource-intensive task across the distributed replica nodes. Under Replicated Training, each replica dynamically controls training data collection if it needs more resources to keep up with the master node. To show the effectiveness of our technique, we implement it in NoisePage, a self-driving DBMS, and evaluate it in a distributed environment. Our experiments show that training data collection in a DBMS incurs a noticeable 11% performance overhead in the master node, and using Replicated Training eliminates this overhead in the master node while still ensuring that replicas keep up with the master with low delay. Finally, we show that Replicated Training produces ML models that have accuracies comparable to those trained solely on the master node.

Acknowledgments

I want to thank my advisor Andy Pavlo for his mentorship not just through this entire project, but throughout the majority of my undergraduate college experience. Having fun is a significant motivator for my work ethic, and Andy showed me the joy in doing sick computer science work. His lessons and sensational stories are something I will carry dearly into my next chapter of life. I also want to shoutout and thank the fantastic peers I've worked with throughout my tenure with the CMU Database Group, including but not limited to: Matt Butrovich, Tianyu Li, Lin Ma, John Rollinson, Wan Shen Lim, Amadou Ngom, and Terrier. I wish them all the best and I am hyped for all the amazing work they will continue to do. I want to thank Hadley Killen for her unwavering support and excitement about me pursuing my passions. Finalmente, quiero darle mis gracias a mi familia por su apoyo y amor. Todos mis logros son por y gracias a ellos.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Database management systems (DBMSs) are notoriously hard and expensive to manage and tune [44]. DBMSs can have hundreds of tuning knobs and configuration settings that have a significant impact on the performance of the system. To best make use of these options, a database administrator (DBA) is in charge of hardware provisioning, monitoring, and tuning for a DBMS. DBAs, however, are expensive to hire and do routine work such as tuning individual knobs and monitoring their effects or scaling the DBMS hardware to meet with demands. It plays a critical role in the deployment of a production DBMS, but one that modern computational techniques could improve.

Previous research has explored using machine learning (ML) as a solution to this problem. ML is used to train models that predict the runtime behavior of DBMSs and automate tuning system configurations. Some systems have used ML as a central coordinating component in the system, creating a new class of so-called self-driving DBMSs [34, 36]. These systems use ML for high-level decision making, such as workload prediction [20], or system-wide tasks such as transaction scheduling [17]. Analogous to this, other systems use ML to solve component-scoped challenges by creating so-called learned components. Examples include learned index structures [16], cardinality estimation [15], and join ordering in the PostgreSQL query optimizer [22].

Other research has resulted in tools external to the DBMS that are powered by ML models. The tools output system configurations, such as knob settings [10, 44, 48], or make recommendations, such as adding or dropping indexes [9]. The DBA then tunes the system by deciding which recommendations to apply.

All these approaches face the same problem: the ML models that power these techniques require a lot of training samples. These models are trained using metrics collected

by the DBMS. These metrics include query arrival rates [20], latch contention in the transaction manager, and disk write performance in the log manager. The more metrics our system generates, the more models we can train to automate its run time configurations.

Further, it is crucial to generate training data in different environments to prevent the models from overfitting to one specific configuration. In cloud environments, even identical instance types can have significant variations in performance [8]. Accounting for such differences during training data generation will make the models robust and resilient to dynamic environments.

One possibility is to generate training data for models in an offline setting (i.e., not during production execution). Some systems [33] capture production workloads to allow DBAs to try out different system configurations on a snapshot of the database. Other tuning systems [10, 44, 48] use training data from a simulated or prior execution to train ML models that recommend configurations in new deployments. The problem with these offline approaches is that the models are trained on past workloads and environments. They do not test the effects of system configurations on the most recent workload. As workloads evolve, the efficacy of these tool's recommendations decreases if they are not retrained [7].

In contrast, an online approach collects training data that the production system generates during live execution. Systems such as IBM's LEO [23] and Automatic Indexing [9] use metrics generated from live query execution as training data. These systems can also tune their models on the fly, i.e., while the system is running. This technique has limitations; however, as it will only collect data for the specific environment and configuration the production system is currently running. As a result, the models could overfit the production setting. On the other hand, the exploration and testing of configurations in a production environment can have serious performance risks, so customers expect these features to result in zero regressions on their performance [9]. These regressions include both the overhead incurred from running these systems and the potential degradations of harmful recommendations or configurations.

1.1 Motivation

The need for self-driving DBMS has increased with the growing size of data sets and the desire to run complex analytics over that data [36]. As databases grow in size, so does their complexity, and therefore the difficulty to tune the DBMS. DBAs spend approximately a quarter of their time on tuning tasks and account for nearly 50% of a DBMS's operating cost [6]. Self-driving DBMSs could alleviate this problem by using ML models to optimize

themselves without human intervention, allowing DBAs to spend their valuable time on other essential tasks.

In a self-driving DBMS, the training data, or input, to the ML models is metrics collected during the execution of the system. For example, the logging component may collect metrics on how long it takes to write data to disk. This metrics collection, however, comes at an overhead to system performance. To measure the effects of metrics collection in a DBMS, we execute an online transactional processing (OLTP) workload (TPC-C [5]) on PostgreSQL with and without metrics collection, taking the average throughput performance over five runs. In our benchmark, the number of client threads equal to the number of warehouses. We run this benchmark on an AWS m5d.4xlarge instance running with an Intel Xeon Platinum 8175 with eight threads (16 hyper-threads). Our experiments indicate that when scaling up the number of threads, the overhead of metrics collection decreases the throughput of PostgreSQL by 11% on average.

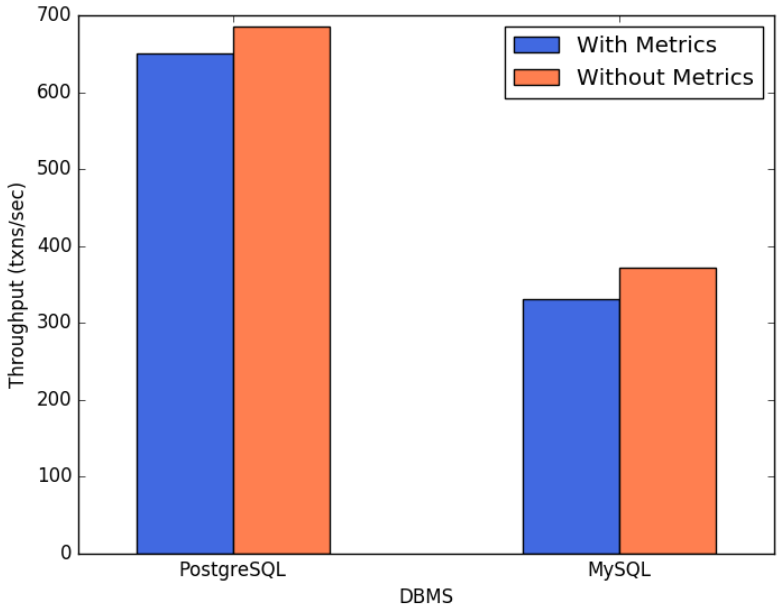


Figure 1.1: Metrics Overhead on Throughput – Effects of metrics collection on throughput of TPC-C with 4 warehouses for various systems

Further, we execute the TPC-C benchmark with four warehouses on an AWS m5ad.xlarge instance, and show the effects of metrics collection on throughput for PostgreSQL and MySQL-8 in Figure 1.1. The measurements show that metrics collection results in a no-

ticeable throughput degradation of 5.4% and 12.5% for PostgreSQL and MySQL-8, respectively.

We propose a technique called Replicated Training for users that may not want to pay the performance overhead of metrics collection on the master node, but still want to take advantage of the autonomous capabilities of a self-driving DBMS. Many DBMSs operate in a distributed master-replica architecture, where the master node sends new changes to replica nodes that hold up-to-date copies of the database. Replicated Training makes use of these replicas by using their metrics collected to train the ML models of a self-driving DBMS. Doing so balances the overhead of training data generation across the entire distributed system.

Replicated Training allows offloading metrics collection to replica nodes; however, the system must still make sure that the replica can keep its copy of the database in sync with the master. Many database users have strict service-level agreements (SLAs) with regards to replication delay to limit the amount of data loss in the event of a node failure in an asynchronous replication environment. Allowing metrics collection to run unchecked in the replica can result in degradations in transaction replaying performance, resulting in higher replication delays.

1.2 Contribution

We present our Replicated Training technique, discuss the system architecture considerations that make it possible and implement it in NoisePage. We further propose additional variants to the method to highlight its promising potential. Finally, we evaluate the effectiveness of Replicated Training and show its ability to produce accurate models without needing to pay the penalty of metrics collection on the master node.

The remainder of this thesis is structured as follows. We first discuss some background for DBMSs and ML in Chapter 2. In Chapter 3, we describe the system architecture of NoisePage, primarily focusing on the components that make distributed replication possible. We then present Replicated Training in Chapter 4, and discuss various considerations and variants in the technique. Next, we evaluate Replicated Training in Chapter 5. Finally, we discuss related work in Chapter 6 and conclude in Chapter 7.

Chapter 2

Background

2.1 Self-Driving Database Management Systems

Self-driving DBMSs are a family of DBMSs that utilize ML models to optimize their runtime performance and behaviors [36]. Self-driving DBMS consist of the same infrastructure in a traditional DBMS, along with a central self-driving component to manage the autonomous operation. At a high level, the self-driving infrastructure consists of a modeling and prediction component. The modeling component receives as input metrics collected throughout the system. It then builds ML models representative of the DBMS's runtime behavior. In the prediction component, these models are used to forecast future workloads and recommend actions to optimize the DBMS's configurations to handle these workloads [20].

The self-driving component optimizes the DBMS by recommending and applying *actions*. Actions range from adjusting configuration knobs on the memory available to the system, constructing index structures based on predicted future workloads, to even changing the storage layout of data in memory [36]. The self-driving component monitors the consequences of applying these actions through the metrics collected by the system. If it finds that an action has an unpredicted negative impact, the system can rollback the action. The system also maintains a history of past actions and their effects.

2.2 Replication

Practically every production DBMS will support some form of replication. Most systems will employ a hot-standby approach to replication, where a different database instance is running on a separate machine. This instance, known as a replica node, will receive changes from the master and replay them to create a consistent snapshot of the database. In the event of the master failing, the replica can become the new master. The number of replicas is provisioned by the user, and is usually at least two to ensure a greater degree of fault tolerence.

Table 2.1 shows the replication protocol of various log shipping DBMSs. Production systems handle how changes are communicated between the master and its replicas in different ways. Most systems [14, 26, 30, 32, 35, 37] will do a form of log shipping, where the primary node ships changes to the replica. The replica then replays the changes in the order they were made on the master. Other systems [1, 12, 42] abstract away how replication is done by letting an underlying cloud object storage handle the data replication. To the system programmers, this appears as if all replicas are reading from the same logical copy of the data.

If systems employ replication using log shipping, they will have to make the critical decision of what type of logging to use: physical or logical logging. Physical logging involves recording the physical changes that are made to the storage layer of the DBMS. Logical logging, on the other hand, logically describes the high-level modifications made to the data [47]. Most commercial systems that employ logical logging will implement command logging, where the transactions themselves (or commands) are logged [21]. Each logging type has its tradeoffs. Physical logging carries more overhead during execution because it produces more logs than logical logging, but is faster to replay and more parallelizable during recovery. Logical logging, on the other hand, has less overhead during execution (typically a single log per query) but is more expensive to replay during recovery. Some systems [47] implement hybrid logging, which is a mix of logical and physical logging. Hybrid logging dynamically determines which logging type to use that best fits the replaying behavior of the workload.

Additionally, there are two ways changes are committed in a replicated setting. In synchronous replication, the DBMS does not notify the user that their transaction has been persisted in the database until at least one replica replays the changes. In asynchronous replication, the user may be told the DBMS has persisted their transaction before a replica has completely replayed their changes. Synchronous replication minimizes data loss in the event of a master node failure, as all committed changes are guaranteed to be on some replica. Synchronous replication, however, adds more latency to requests as at least one

remote replica must receive and replay the changes.

System	Logging Type	Replicated Commit Type
MemSQL [24]	Logical	Asynchronous
MongoDB [29, 30]	Logical	Asynchronous
MySQL [31, 32]	Both	Both
PostgreSQL [37]	Physical	Both
SQL Server [25, 26]	Logical	Both

Table 2.1: Replication Protocol for Various DBMSs – Systems that do log shipping can still have differences in their replication protocol.

2.2.1 Replication Delay

A common problem that all DBMSs that support replication will suffer from is replication delay: the difference in time between when a change is applied on the master, and when a replica replays the change. Many DBMS users have strict service-level agreements (SLAs) that they must adhere to. A delimited replication delay is a frequent SLA users expect their DBMS to support to ensure a bounded degree of consistency between master and replica nodes. Low replication delays also reduce the amount of data loss in the event of a crash. Synchronous replication also increases this delay, as an extra network trip is incurred before the transaction is allowed to commit.

2.3 Training Data Collection

Effective training data generation is a hard and expensive task for production ML models. Companies have entire teams of costly data analysts and spend thousands in compute resources to generate enough training data for their models to produce reasonable ouputs [39]. Training data is a vital piece of any ML operation, and there are many challenges associated with generating quality training data. Further, many ML techniques involve supervised learning, where the training data must be labeled with a desired output value.

There are two crucial facets of training data generation that we will concern ourselves with: performing actions to produce this training data and choosing what data to generate. To illustrate the challenges in these two areas, we will discuss them in the context of self-driving cars. Self-driving cars serve as an excellent comparison to self-driving

DBMSs as they are both expensive to deploy, and must be able to handle completely new environments or workloads with little to no human interaction.

Performing actions to generate training data can be broken down into two parts: sampling and labeling. Self-driving cars sample new data by driving through streets and recording the environment of the car through the use of high-tech cameras. Apart from the difficulty of building all the technology to capture the car's conditions, this process is costly. In the case of supervised learning models, the sampled data must then be labeled. Often times this requires large numbers of humans manually labeling objects in the car's film. For other domains, such as ML for medicine, labeling can often be incredibly expensive as few people are qualified enough to label data accurately [39].

The other aspect we consider is choosing what data to sample. As we have discussed, the process of sampling and labeling can be extremely resource-intensive. On the one hand, the training data should be diverse for the ML models to adequately generalize [11]. Capturing every single environment and data point, however, is often not feasible. For example, our self-driving car can prioritize learning how to drive down streets we've never seen before. It's not realistic, however, to drive down every possible road during every different environment, such as daytime, nighttime, rain, rush hour, etc.

One approach is to build an environment, or gym, where the workload can be simulated to generate training data and see how the self-driving infrastructure behaves [4]. For example, one could construct a test track where a self-driving car could drive around and learn. There are some obvious limitations to this approach. The first is cost: it may be impossible to build such an environment depending on the resources required. Many production DBMSs are too large and expensive to duplicate in a simulated environment. Secondly, these simulated environments only approximate the real conditions. A car test track would not perfectly replicate the chaotic and variable reality of driving in the real world. Similarly, a simulated environment would not be able to perfectly represent the variabilities of a mission-critical production system, such as workload spikes or machine failures.

2.4 Active Learning

In some ML environments, training data annotation is often an expensive task, as we discussed in Section 2.3. Suppose the DMBS collects metrics for query execution by annotating queries with their runtime cost. The system needs to execute the query to compute this runtime cost. Depending on the query, this could cost a lot of resources such as CPU utilization or memory. Some of the variations to Replicated Training we

propose make use of an ML technique called active learning to overcome high labeling cost environments. The idea behind active learning is that the model is allowed to choose which data to label and train on [2]. In the example of query execution, the model may determine that it does a poor job at predicting runtimes for queries that sort their output. The model will, therefore, choose to only execute and generate metrics on queries that contain a sort operator. Using active learning often leads to better model accuracy with fewer training samples [41].

Chapter 3

System Architecture

NoisePage is an in-memory DBMS that supports ACID transactions and SNAPSHOT-ISOLATION. The system is built in C++, and supports the PostgreSQL wire protcol for communicating with the database server. In this chapter, we will discuss the architectural components of NoisePage relevant to supporting Replicated Training.

3.1 Transactions

Transactions are the atomic unit of work in a DBMS. The way transactions in NoisePage make updates to the database is conducive to a streamlined logging, recovery, and physical replication scheme. NoisePage's transactional engine is a multi-versioned delta store [40] that supports SNAPSHOT-ISOLATION [3]. The transactional engine is coordinated by the Transaction Manager (TM) component. In NoisePage's transaction engine, readers do not block writers and vice versa, however, write-write conflicts on a per tuple basis are not allowed. When logging (Section 3.3) is disabled, transactions are considered active between when they begin, and when they commit or abort. When logging is enabled, transactions are active between when they begin, and when the log manager (LM) serializes their changes. Enabling synchronized commit further extends this by guaranteeing that a transaction is active until its changes are persisted in the disk.

Tuples in NoisePage are uniquely identified by a *TupleSlot* that contains offset information about the tuple. When a tuple is first inserted into a table, the *TupleSlot* denotes the in-memory location in the table. Transactions update tuples in the database by creating delta records that are not an updated copy of the tuple, but rather the after-image of the changes. There are four types of delta records: Redo, Delete, Commit, and Abort. Redo

records represent both inserts and updates. Aside from the changes or transactional actions, the delta records hold additional information such as the transaction *start* timestamp and what database and tables were modified. These pieces of information are necessary for ensuring correct replaying of the record during recovery or replication. Delta records, however, do not hold all the information needed to accomplish this. It is the role of the LM, discussed in Section 3.3, to serialize any additional information required along with the record.

Each transaction has its own redo buffer to store these records. Rather than using an extendible buffer, redo buffers are fixed size (4096 bytes). Upon creation, transactions acquire a buffer from a pre-allocated, centralized buffer pool. To record a change, a transaction reserves space in its redo buffer and writes in the new change. In the case that there is not enough space left in the buffer, the transaction will relinquish the buffer to the log manager, and receive a new one from the buffer pool. This allows downstream consumers, such as logging and replication, to process these changes before a transaction has completed.

To commit, the transaction will write a commit record to the redo buffer and relinquish it to the log manager. During commit time, the oldest active transaction timestamp is also queried from the TSM, and included in the commit record. Aborts, on the other hand, require additional logic to ensure correct behavior during recovery and replication. We discussed before that a transaction relinquishes its redo buffer to the log manager once it is full. Due to this, an aborting transaction can have already persisted records. For correct behavior during recovery and replication, an aborting transaction that has previously relinquished a redo buffer must also write and relinquish an abort record. Without this abort record, a downstream consumer would be unable to differentiate between an aborted transaction and an uncommitted, active transaction. From an observation of an OLTP workload [5], however, it is rare that an aborting transaction will ever relinquish a buffer, as the amount of data generated by an OLTP transaction rarely fills up an entire redo buffer.

3.2 Timestamp Manager

The Timestamp Manager (TSM) is a central component of NoisePage that is in charge of providing atomic timestamps to running transactions. Timestamps are globally unique 64-bit unsigned integers. A transaction is given a *start* timestamp when it begins, and a *commit* timestamp if it commits.

The TSM also maintains an *active transaction set* that contains the *start* timestamp of

every active transaction running in the system. Components can query the TSM for the oldest active transaction timestamp (smallest *start* timestamp) using this set. If there is no oldest active transaction, it is indicated using a special value. The importance of this information is discussed in Section 3.4. If logging is enabled, then once the LM (discussed in section Section 3.3) serializes the delta records of a transaction, the transaction is removed from the *active transaction set*. This prevents the background garbage collecttor (GC) from cleaning up the transaction before its changes are persisted to disk.

Fast performance of polling for the oldest active transaction timestamp is crucial because it is done during the critical section of every committing transaction. When logging is enabled, there is no bound on the number of active transactions, since transactions are active until they are at least serialized. Due to this, polling for the oldest active transaction, which requires scanning the entire active transaction set, can be a costly operation. Instead, the TSM keeps a cached oldest active transaction timestamp. During commit, transactions atomically read this value instead of scanning the active transaction set. Background GC periodically (default to every 5 ms) refreshes the cached value. Although the cached value might be a stale view of the system for a committing transaction, it still maintains correctness, as the transaction associated with the cached value is guaranteed to have been active at some point and older relative to any transaction which reads the cached value.

3.3 Logging

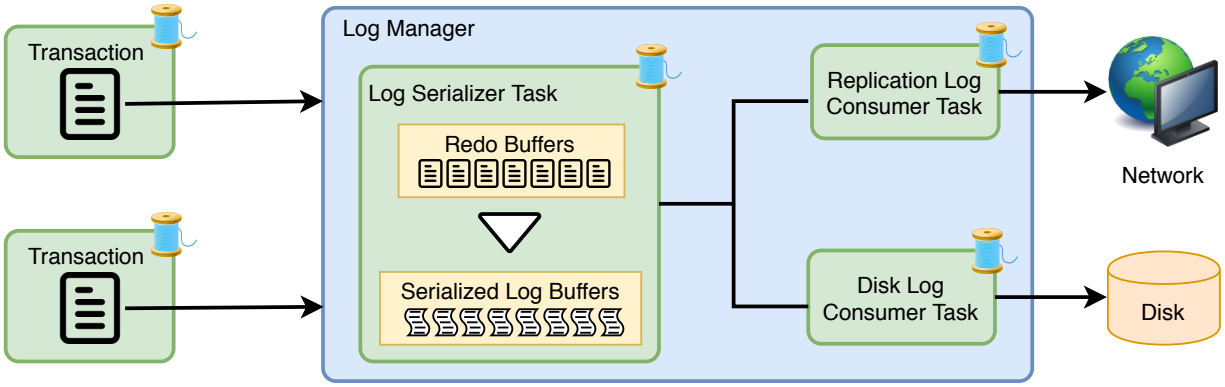


Figure 3.1: Log Manager Architecture – The log manager receives as input log record buffers from transactions, serializes these records, and sends them to different destinations (e.g. disk).

Changes to a database in NoisePage are persisted on disk using write-ahead logging

[28] through a dedicated component called the Log Manager (LM). The LM serializes delta records such that they are entirely replayable on their own without any additional in-memory metadata (e.g., attribute sizes), such as in the case of recovery after a system crash or replication.

The LM coordinates multiple parallel tasks structured in a Producer-Consumer architecture shown in Figure 3.1. The producer, a log serializer task, feeds serialized log records to multiple log consumer tasks (e.g., replication log consumer task).

3.3.1 Log Serializer Task

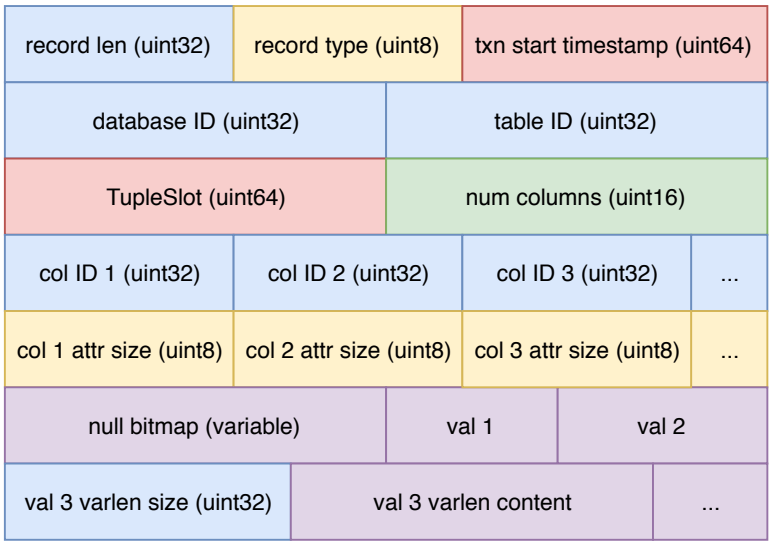
The log serializer tasks receive redo buffers from transactions and write their raw memory contents (ignoring any padding used for memory alignment) into fixed-sized buffers (4096 bytes) to be processed by the log consumers. The serializer task ensures that all the information needed to replay the delta record is included alongside it. For example, large variable-length strings (varlens) in NoisePage are not inlined in the delta record but instead stored in a separate memory location, with a pointer to the varlen entry stored in the delta record. The serializer task must thus fetch the varlen entry and serialize it inline.

Figure 3.2 shows the serialized format of the log records. The delete, commit, and abort records have fixed sizes of 29, 29, and 13 bytes, respectively. Due to the variability in the updated data, the size of serialized redo records varies depending on the contents. Over a run of the TPC-C benchmark [5] with four warehouses, NoisePage will generate approximately 1 GB of total log data, with an average redo record size of 100 bytes.

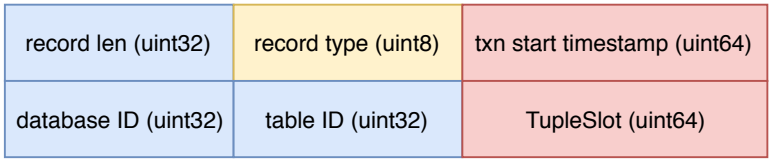
The serialized ordering of the records is essential to ensure correct replayability. Recall from Section 3.1 that transactions relinquish redo buffers as soon as they fill them. This means that records of different transactions can be interleaved in the log. Additionally, transactions can appear in the log in non-serial order relative to their *start* timestamp. The only guarantee that is made is that records for individual transactions appear in the order that they were created, with a commit or abort record always being the last record to appear. As we will discuss in Section 3.4, this guarantee, along with the oldest active transaction timestamp discussed in Section 3.1, is enough to achieve a consistent snapshot of the database after replaying the log.

3.3.2 Log Consumer Tasks

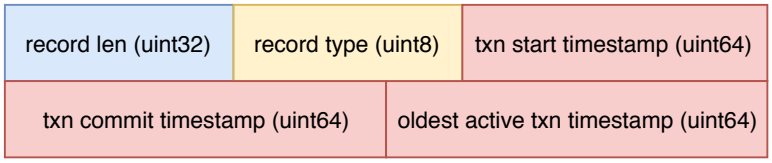
The buffers generated by the log serializer are distributed to potentially multiple log consumers. Each consumer is given a copy of the serialized buffer so they can each work



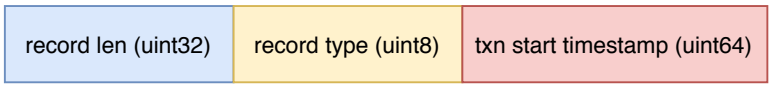
(a) Redo Record



(b) Delete Record



(c) Commit Record



(d) Abort Record

Figure 3.2: Log Record Serialization Formats – Along with the log record, additional information must be serialized to ensure replayability.

independently of each other, preventing a slow consumer from slowing down the others. Currently, NoisePage supports two consumers: (1) a disk consumer that writes logs to a file on disk, and (2) a replication consumer that sends logs over the network to replicas.

The disk consumer task waits until it receives buffers from the serializer task, and writes them to a log file. For proper performance in persisting the log file to the disk, we take advantage of group commit, which is configurable by the user with a combination of time and data size settings. For example, under the default settings, the disk consumer task will persist the log file every 10 ms or if more than one megabyte of data has been written since the last persist.

The replication consumer task also receives buffers from the serializer task and sends them over the network to any replicas listening to the master. There is no group commit done in order to minimize the replication delay. Instead, the system sends serialized logs as soon as they are handed off to the replication consumer task. We describe the network protocol used for sending logs between nodes in Section 3.5.

3.4 Recovery

The Recovery Manager (RM) component manages recovery in NoisePage. The RM receives serialized log records from an arbitrary source and replays them to produce a consistent view of the database.

A log provider will deserialize log data into delta records, and hand them off to the RM. The log provider provides an abstraction to the recovery manager as to what the source of the records is. This way, log records can come from any source, such as a log file or over the network, without any changes needed to the log replaying logic of the RM.

Abstracting the source of log records gives NoisePage the advantage that the replaying component of replication can be implemented for "free." By simply having the source of records be a stream of logs over the network, a standby replica can use the RM to replay the log records arriving from the master node. Other systems, such as PostgreSQL [37], also take this approach for replication. This method of replication, however, requires the processing model of the RM to be a streaming model. We can not take advantage of cases when all the log data is available apriori, as is the case during single node crash recovery. Other recovery algorithms, such as ARIES [28], take advantage of having access to all the data from the start and trim out unnecessary processing.

3.4.1 Log Record Replay

The API for updating tables in the system allows for easy replaying of records. Recall from Section 3.1 that transactions must write their changes as delta records in their private buffers. The system takes advantage of this by passing tables a pointer directly to these records in the buffer. This prevents having to make an additional copy of the data. Since recovery deserializes delta records, there is no need to transform the data; the table API accepts these delta records directly.

The *TupleSlot* contained in the replayed record is no longer valid during recovery, as it represented a unique memory location before recovery. Instead, it is used during recovery to create an internal mapping from the original `TupleSlot` to the new one when an insert is replayed. Using the mapping, the RM can correctly identify what `TupleSlot` to apply updates to after recovery.

Processing records that modify the catalog require additional logic. The metadata stored in the catalog is kept in tables. These tables are the same structure used for user tables in the system. This makes recovering the catalog metadata the same process as updating a user table, as they appear as updates to the catalog tables. Despite this, the RM needs additional logic to re-instantiate particular in-memory objects in the system, such as indexes, views, or user tables.

While recovery could replay changes as it sees them in the log and roll them back in the case of aborts, we will see in Section 3.4.2 that the RM must defer all updates until it sees a commit or abort record anyway. Buffering also gives the added advantage that it prevents the unnecessary work of replaying records for aborted transactions. When the RM sees an abort record, it cleans up and discards any records buffered for that transaction. When the RM sees a commit record, it processes the transactions as described in Section 3.4.2. The transactions apply changes in the order they were made before recovery.

3.4.2 Transaction Replaying

The RM must make special considerations when replaying transactions because of the streaming processing model of recovery and the design of our transactional engine. Recall from Section 3.3 that the only guarantee we have about the ordering of logs is that changes for an individual transaction appear in the log in the order they occurred. There are no guarantees about how transactions are ordered relative to each other in the log, so it is the responsibility of the RM to execute them in an ordering that results in a consistent view of the database.

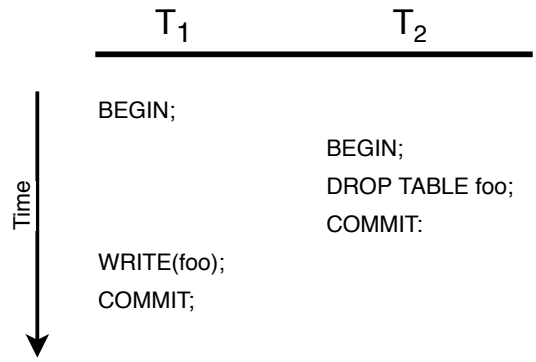


Figure 3.3: Schedule With DDL Changes – This schedule is allowed under SNAPSHOT-ISOLATION, but can create problematic races involving the DDL command.

Recall from Section 3.1 that NoisePage supports SNAPSHOT-ISOLATION, which means that each transaction operates on a "snapshot" of the database taken when the transaction begins. Additionally, any committed transactions which executed concurrently are guaranteed to not have any write-write conflicts with each other on a per tuple basis. The TM guarantees that there are no dependencies between transactions that executed concurrently and all committed transactions that the RM replays will successfully commit. Further, because we use physical logging, all the values written during log replaying are predetermined (i.e., no writes are based on randomization). Based on these two guarantees, the RM can replay transactions sequentially (i.e., a transaction commits before the next one is allowed to begin).

We have determined that the RM can execute transactions sequentially, but the order in that it executes them is also important. Consider the schedule in Figure 3.3 that is allowed under SNAPSHOT-ISOLATION. Although there is no write-write conflict in this schedule, there is an implicit conflict due to the DDL change (`DROP TABLE`). During replaying, transaction T_1 must be replayed before transaction T_2 in order to ensure that T_1's write to table foo occurs before T_2 deletes foo. There are no guarantees about the ordering of logs between transactions, so T_2's changes can appear before T_1's changes in the log. Even worse, if T_1 is a long-running transaction, its changes may not appear until much further along in the log. This raises the issue of when is it safe to execute a transaction.

Executing transactions in the order in that they appear in the log could violate SNAPSHOT-ISOLATION, since executing a newer transaction first would create a different snapshot than what an older transaction saw when it was executed before recovery. Thus, the RM must execute transactions in the order that they were created (i.e., ordered by their *start*

timestamp).

One approach for accomplishing this would be as follows: A transaction T_i with *start* timestamp i is safe to replay after the RM has replayed transaction T_{i-1}. The TM, however, does not guarantee that consecutive transactions have consecutive *start* timestamps. Additional processes, such as GC or assigning commit timestamps, also receive timestamps from the TSM.

The solution to this problem is to use the oldest active transaction timestamp (described in Section 3.1) as an indicator for when to replay a transaction. When a transaction commits, the LM includes the oldest active transaction timestamp at the time of commit in the commit record (shown in Figure 3.2(c)). When a transaction is entirely deserialized, rather than executing it right away, the RM defers its execution. Using the oldest active transaction timestamp i stored in the commit record, the RM then executes, in sorted order oldest-to-newest, all deferred transactions with *start* timestamps j where $j \leq i$. If i is the special value reserved for indicating there are no active transactions, then the RM executes all deferred transactions.

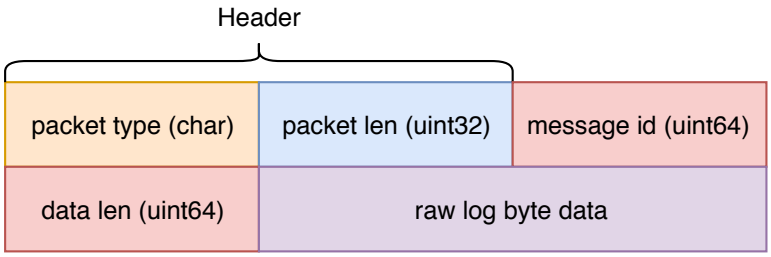
Once again, consider the schedule in Figure 3.3. Suppose the *start* timestamps of T_1 and T_2 are 1 and 2 respectively. The commit record of T_2 will indicate that the oldest active transaction timestamp at commit time was 1. If T_2 is serialized **before** T_1, it will be deferred because $1 < 2$. Eventually the RM deserializes and executes T_1, followed immediately by T_2, because there were no older transactions at the time T_1 committed.

3.5 Internal Replication Protocol

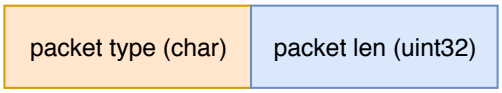
The Internal Replication Protocol (IRP) is the network protocol used to communicate between nodes of NoisePage. IRP uses TCP/IP sockets to send packets of data between the master and replica node(s). NoisePage uses TCP instead of UDP for its delivery guarantee since no log data can be lost over the network. NoisePage has a network layer that sits at the top of the system to handle client and other NoisePage node connections. The network layer is designed to support multiple protocols on separate ports. Currently it supports IRP and the PostgreSQL Wire Protocol (described in [19]).

Packets in IRP consist of a header and payload. The header is used by NoisePage's network layer to parse the packet. It consists of single `char` to identify the packet type, and a `uint32_t` value for the size of the packet. This portion of the protocol resembles the PostgreSQL wire protocol.

The current packet types for IRP are shown in Figure 3.4. The Replication Data packet



(a) Replication Data Packet



(b) End Replication and Replica Synced Packets

Figure 3.4: IRP Packet Types – Packets are minimal in size to reduce network congestion and speed up packet processing

(Figure 3.4(a)) holds variable-length portions of the serialized log data from the master node's LM to the replica. The End Replication packet is sent by the master and tells the replica to end replication. The Replica Synced packet is sent from the replica and notifies the master that the data in both nodes are in sync. Both these packets (figure Figure 3.4(b)) require no payload as the type in the header entirely describes the purpose of the packet.

3.6 Replication

Figure 3.5 shows an overview of the replication architecture in NoisePage. Replication can be done by two NoisePage instances, a master and a replica, running on different machines connected to the internet. Replication is established by a connection between the master's LM and the replica's network layer. The LM serializes the logs on the master and places them in a Replication Data packet that is shipped over the network to a replica. The packets reach the replica's network layer and are handed off to the RM running in the system. A log provider parses the log data from these arriving packets into log records. For replaying these logs, we discussed in Section 3.4 that we can accomplish replication using the same RM logic used for crash recovery. If the replica is ever in sync with the master (i.e., there are no more logs left to replay), it will also send a Replica Synced packet.

The RM running on the replica will sit in a loop, continually processing log records

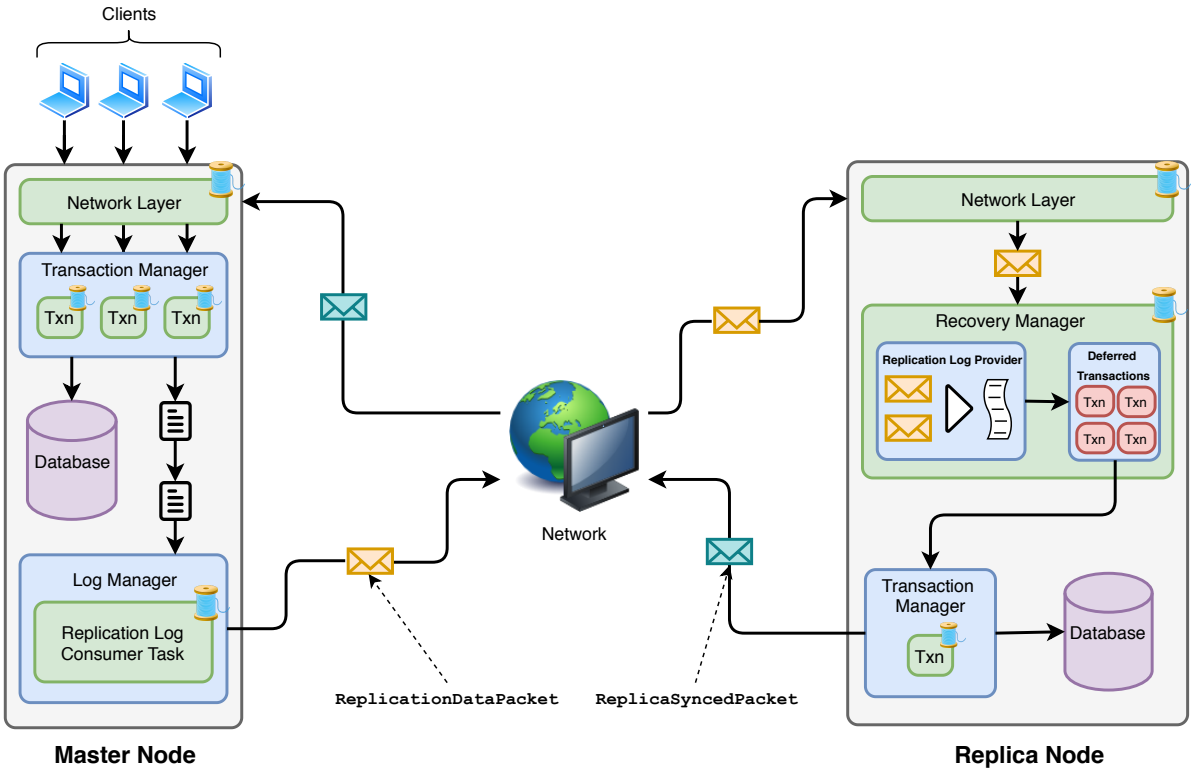


Figure 3.5: Replication Architecture – Although there are other components in NoisePage, this diagram highlights the processes involved in replicating data between a master and replica

as they arrive over the network. This is unlike in crash recovery, where the RM will terminate when it reaches the end of the log. Instead, the master can terminate replication with a replica by sending it an End Replication packet.

Chapter 4

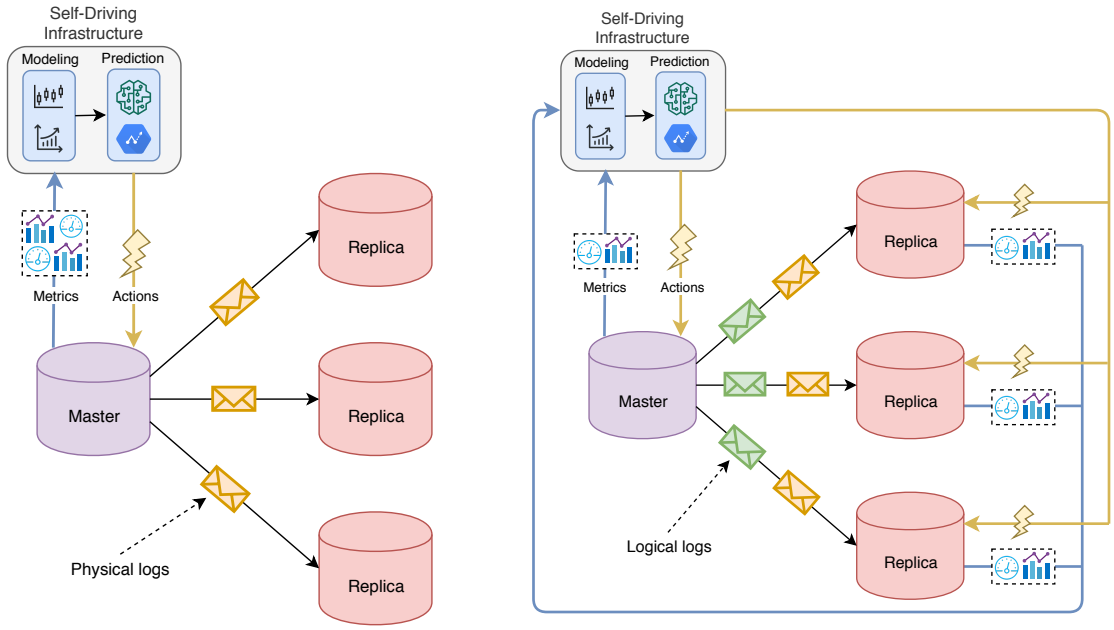
Replicated Training

We discussed in Section 1.1 how self-driving capabilities are beneficial to a system, but collecting the necessary training data is detrimental to performance. Replicated Training leverages the existing architecture of a distributed DBMS to support self-driving infrastructure without paying the penalty of supplying training data solely from the master.

We first present the Replicated Training architecture in a self-driving DBMS. We then discuss how Replicated Training can adhere to performance requirements set by the user through Dynamic Metrics Collection (DMC) while still generating useful models for the system. We propose different policies and techniques to control and improve the effectiveness of DMC. Lastly, we present future extensions to Replicated Training to help with action exploration in a self-driving DBMS.

4.1 Replicated Training Architecture

The crux of Replicated Training is to use resources available on existing replicas to generate additional training data for models in a self-driving DBMSs. In a self-driving DBMS architecture (shown in Figure 4.1(a)), the master node generates metrics by executing queries. The DBMS aggregates and sends the metrics to the self-driving infrastructure, which then uses them as input to train ML models. For instance, the DBMS may keep metrics on which tables and columns queries access over time. The master node sends these aggregated metrics to the self-driving infrastructure that trains a model to forecast future data accesses [20]. The system can then use this model to predict future workloads and recommend actions that improve performance, such as building an index on a column the self-driving infrastructure expects will be frequently scanned. The critical issue is that



(a) Without Replicated Training

(b) With Replicated Training

Figure 4.1: Self-driving DBMS Architectures – Replicated Training enhances a self-driving DBMS by leveraging database replicas for training data generation

metrics collection penalizes performance on the master node, which may exceed some user's requirements.

With Replicated Training, the system balances the metrics collection overhead by distributing the task of training data generation across the entire distributed topology. As shown in Figure 4.1(b), replicas send their aggregated metrics to the self-driving infrastructure. The process of training models and recommending actions remains the same as in the self-driving architecture shown in Figure 4.1(a). The user can choose whether the master should output metrics, or rely entirely on replicas for training data generation. As we will discuss in Section 4.3, the self-driving infrastructure can also choose to test actions on a replica before applying them on the master node.

4.2 Dynamic Metrics Collection

Customers often have strict SLAs for their DBMS deployments. For instance, some customers may require that the replication delay (discussed in Section 2.2) between their master and replica nodes is at most 100 milliseconds. We showed in Section 1.1, however, that metrics collection causes performance degradations in a DBMS. To avoid this issue, we propose a method that allows the DBMS to dynamically control when to collect metrics under Replicated Training. Instead of metrics collection being permanently enabled, the system enables metrics when replication complies with some policy. We also propose tuning the granularity of metrics collection by selectively enabling or disabling metrics for specific components. Finally, we propose dynamically performing hybrid logging to exercise more layers of the system in the replica node.

4.2.1 Metrics Collection Policies

To control when metrics collection is enabled, Replicated Training can impose a policy that determines when to enable or disable metrics collection. For this section, we will assume that metrics are either enabled or disabled across all components, but as we will see in Section 4.2.2, this is not always the case.

The goal of Dynamic Metrics Collection is to ensure that the replica can remain in sync with the master when using Replicated Training. Since replication delay gives a numerical estimate of how in sync the replica is with the master, we propose a policy around this value as follows: if the replication delay during the previous time window (default: 1 second) exceeds a fixed threshold (e.g., 100 milliseconds), then the system disables metrics collection until the delay drops below the threshold, after which the system can enable metrics once again. Assuming a reasonable threshold relative to the expected replication delay, even if Replicated Training causes the to replica fall behind and exceed the delay threshold, it should be able to catch up once metrics collection is disabled. Users can configure this threshold based on their system requirements. Further, as we will see in Section 5.3, metrics collection is generally enabled for more time with a higher delay threshold, resulting in more training data generated.

It is worth noting that the strictness of a metrics collection policy can affect the quality of the training data. Consider an environment with high resource contention where the replica can only keep up with the master when resources are abundant. With too restrictive of a policy, metrics collection will only be enabled when there is low resource contention, resulting in training data that is highly skewed towards such an environment. By using a more relaxed policy, the system can still generate training data during periods of high

resource contention, improving the quality of the training data.

4.2.2 Partial Metrics Collection

Some system's [13, 27] metrics collection allows for more fine-grained metrics control than "all-or-nothing." The system can toggle metrics for individual components during runtime, which we call Partial Metrics Collection. Enabling or disabling metrics for parts of the system can reduce the overall Replicated Training overhead while still generating some amount of training data.

The ability to choose which metrics to enable or disable opens up unique possibilities for dynamic metrics collection. One option would be to disable metrics on the log replaying critical path when the replica falls behind. This option can minimize the overhead while the replica is catching up, while still generating metrics for other tasks in the system, such as garbage collection. Another approach would be to use active learning [2] to prioritize what metrics to keep enabled, and which ones can be disabled. For example, if a replica is violating the collection policy, it can use active learning to determine what components the system's ML models no longer need training data from, and disable metrics for those components. Inversely, if the replica is complying with the collection policy, it may choose to enable metrics for components whose collection is currently disabled if our models require their specific training data. Using active learning reduces the overall overhead of metrics collection while ensuring that models receive the training data they need the most.

4.2.3 Dynamic Hybrid Logging

We discussed in Section 2.2 how some systems use physical logging to replicate data across machines. Physical logs have the advantage that the replica can replay them without additional processing. In the case of training data generation, however, this can be a disadvantage, as the upper layers of the system are not exercised and, therefore, not generating metrics. Logical logging requires processing through the entire DBMS stack (e.g., parsing, optimizing, execution) to replay. This additional processing generates more metrics than processing physical logs. To take advantage of this property of logical logging, we propose a modified hybrid logging scheme that always uses physical logging for replication, but sends a subset of the read workload as logical logs for generating training data (shown in Figure 4.1(b)). These logical logs are then replayed for metrics collection.

Similar to the policies mentioned in Section 4.2.1, we propose two policies to control

which read queries the master will send as logical logs to the replica nodes. The first is to use random sampling: whenever a new read query arrives in the system, it samples a binomial distribution [45] to decide if it should send this query to the replica. By adjusting the success probability of the binomial distribution, the master can control the number of queries sent to the replica, thereby limiting the number of resources used on the replica to replay logical logs. The second approach is similar to the active learning approach proposed in Section 4.2.2. The self-driving infrastructure can use active learning to determine what components of the system it needs to exercise and choose queries based on that. For example, if the self-driving infrastructure has a model on query execution that determines it needs more training data on aggregations, the system can sample queries containing `GROUP BY` clauses. This technique would work well with the active learning-based partial metrics collection approach proposed in Section 4.2.2, as the system can decide which components it needs metrics for and which queries the system can sample to target them.

4.3 Action Exploration

A self-driving DBMSs applies actions to the system to optimize for some objective function. The user can set an objective function (e.g., maximize throughput, minimize latency) depending on their system requirements. Actions applied to the master, however, are not always guaranteed to impact the objective function positively. The self-driving infrastructure might incorrectly predict an action to improve the system, but it does the opposite.

Similar to how Replicated Training leverages replicas for training data generation, we propose using replicas to explore the effects of actions. When the self-driving infrastructure identifies a candidate action, it applies the action on one or more candidate replicas. The self-driving infrastructure then monitors the metrics received from Replicated Training on the candidate replica(s) to evaluate the performance impact of the action. If the self-driving infrastructure determines the action has a positive effect on the objective function, it applies the action on the master; otherwise, it undoes the action on the replica(s). Using Replicated Training prevents the performance monitoring of the action from having severe performance degradations by using the dynamic metrics collection described in Section 4.2.

Chapter 5

Evaluation

We now evaluate our replication architecture and Replicated Training technique. We build all the infrastructure within NoisePage. We use the following two types of machines for our experimental evaluation:

- Type 1: Dual-socket 10-core Intel Xeon E5-2630v4 CPU, 128 GB of DRAM, and a 500 GB Samsung 970 EVO Plus SSD. This machine is used for single-node microbenchmarking.
- Type 2: Single-socket 6-core Intel Xeon CPU E5-2420 CPU and 32GB of DRAM. These machines are used for experiments involving replication between two nodes. Each machine has a 1GB NIC and are connected to each other on the same switch through 1GB Ethernet.

We first give an analysis of the overhead of metrics collection in NoisePage. We next evaluate the OLTP performance of our replication architecture described in Chapter 3. We then observe the behavior of dynamically controlling metrics exporting. Finally, we analyze the effectiveness of our Replicated Training technique to build accurate ML models.

5.1 Replication Architecture

To evaluate our system architecture, we use a write-heavy OLTP workload. Even though reads are not replicated, a write-only workload is not representative of a real OLTP workload. We use the TPC-C benchmark [5] as our OLTP workload, which simulates a ware-

house order processing system. The number of warehouses is used as a scale factor for the TPC-C database, and is also equal to the number of client threads in our experiments.

To simulate a distributed environment, we execute the TPC-C benchmark between two NoisePage instances running on Type 2 machines. We use a master-replica architecture where one instance is the master and serves requests, and the other machine is a hot-standby replica node. We replicate data asynchronously across the two machines. We use the Network Time Protocol (NTP) to synchronize clocks in our machines to get measurements for replication delay. We do not require high clock precision because we are not making any decisions based on timestamp orderings, we are only using the timestamps to estimate delay.

We now evaluate our replication architecture using microbenchmarks, and then define important test configurations and baselines.

5.1.1 Arrival Rate

One important consideration to make with any benchmark is the arrival rate. The arrival rate is defined as the frequency clients query the database. For example, if there are four clients, each executing 2,500 transactions per second (txns/sec), then the arrival rate is 10,000 txns/sec.

It is important to pick a good arrival rate for measuring the replication delay. If the arrival rate exceeds the rate at which replication is able to replay transactions, then the replication delay will grow unboundedly because the replica is not able to keep up with the master node. Figure 5.1 illustrates this effect in NoisePage. We can see how the replica remains in sync wth the master with a sub-second delay until the arrival rate reaches 14,000 txns/sec. After that, the replica is not able to keep up with the arrival rate of transactions, and accordingly the delay sharply increases. This is a natural limitation in any DBMS, although systems may vary in the arrival rate they are able to handle. We assume an arrival rate of 10,000 txns/sec for future experiments to get stable delay measurements.

5.1.2 Replication Delay Over Time

As discussed in Section 2.2, many DBMS users have replication delay SLAs that they expect the DBMS to support. To get an idea of replication delay in NoisePage, we execute TPC-C using asynchronous replication to measure the replication delay over the span of the benchmark. In Figure 5.2, the spikes in delay are a result of few transactions of TPC-C that take longer to replay relative to the other transactions. Despite the spikes, we

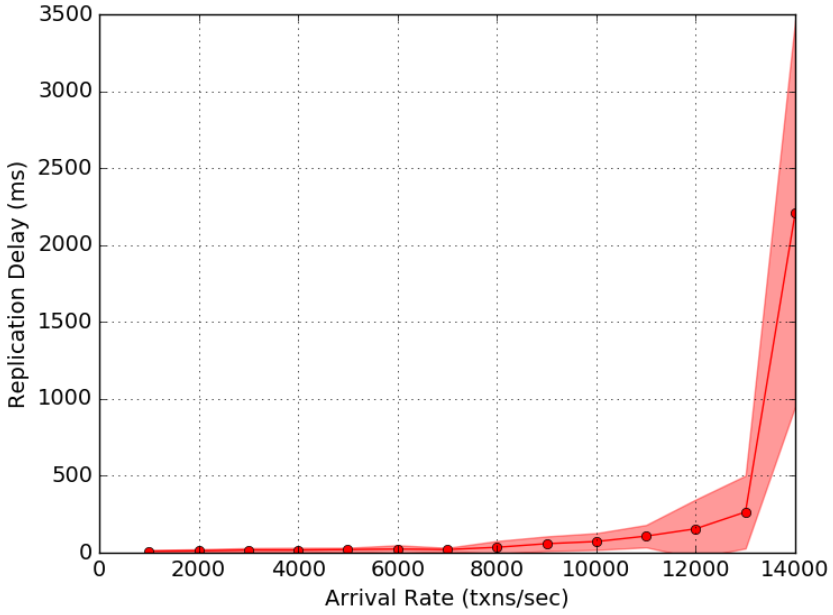


Figure 5.1: Sensitivity of Replication Delay – Measuring the average replication delay with varying arrival rates in NoisePage over TPC-C with 4 warehouses on Type 2 machines. When the arrival rate exceeds the transaction replaying rate, delay sharply increases. The shaded region denotes one standard deviation from the mean for each data point.

see from the running mean (red line) that the replication delay remains stable throughout the benchmark execution. Over the entire benchmark, the average replication delay is approximately 30 ms.

5.2 Metrics Overhead

In Section 1.1, we motivated our decision to use database replicas by discussing the metrics collection overhead, and observed it is on average 11% in PostgreSQL across various number of threads. For this analysis, we used PostgreSQL instead of NoisePage because NoisePage does not have as advanced metrics as PostgreSQL. In particular, NoisePage still has an immature metrics collection infrastructure and does not yet produce as many metrics as a system of its size should.

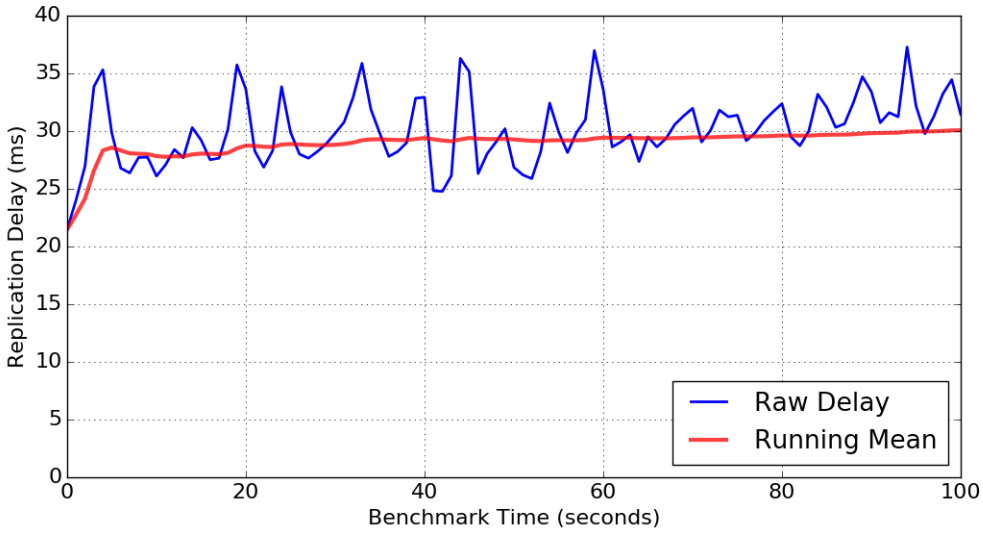


Figure 5.2: Replication Delay in NoisePage – Measuring the average replication delay in NoisePage over TPC-C with 4 warehouses on Type 2 machines. We average the delay for each second, and plot the average over 10 benchmark runs. The red line shows the running mean of the replication delay.

NoisePage currently has two (out of 10) high-level components, the TM and LM that produce metrics. Collectively, these two components export 11 unique metrics (e.g., disk write speed, transaction latch wait time). For reference, we estimate PostgreSQL-9 to export approximately 300 unique metrics. Due to this difference, we simulate NoisePage having similar metrics overheads to PostgreSQL.

The approach we take to simulate a realistic DBMS metrics overhead in NoisePage is to scale up the amount of metrics data exported by each component (i.e, when a metric is generated, the system exports it multiple times). To show the effect of this approach, we execute the TPC-C benchmark with six warehouses on machine Type 1. We compare the transaction throughput while scaling up the number of metrics exported using the approach described. Figure 5.3 shows the effects of this technique on transactional throughput. With the current metrics in the system (red line), NoisePage executes approximately $67,000$ txns/sec. If we scale number of metrics to the number of metrics we estimate PostgreSQL exports (green line), we see a throughput of approximately $65,000$ txns/sec, only a 3% overhead. This is not equivalent to the 11% we expect to see because throughout the lifecycle of a tranaction, the system exports different metrics at different frequencies. Therefore, scaling NoisePage's metrics to the same number of metrics that PostgreSQL

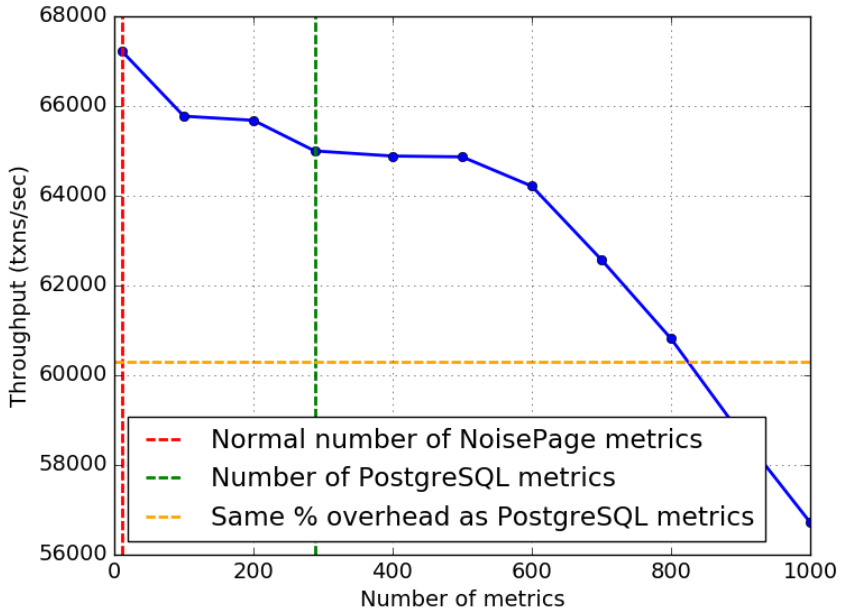


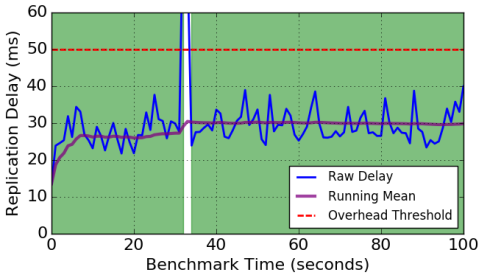
Figure 5.3: Metrics Overhead in NoisePage – Overhead of metrics collection over TPC-C with 6 warehouses as we scale up the number of metrics exported.

exports is an insufficient comparison, as NoisePage exports at different frequencies than PostgreSQL. Instead, we scale up the number of metrics until we see the 11% overhead (yellow line), which occurs at approximately 800 metrics. We use this scale of metrics collection for future experiments.

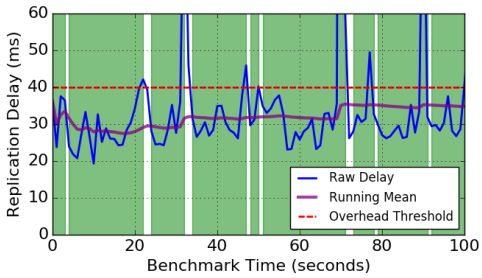
5.3 Dynamic Metrics Collection

In Section 4.2, we proposed using Dynamics Metrics Collection (DMC) for Replicated Training. The basis of DMC is to prevent the overhead of training data generation from increasing the replication delay on the replica. We proposed various policies for controlling DMC, including one based on the measuring average replication delay over some time window and limiting it based on a user-defined threshold.

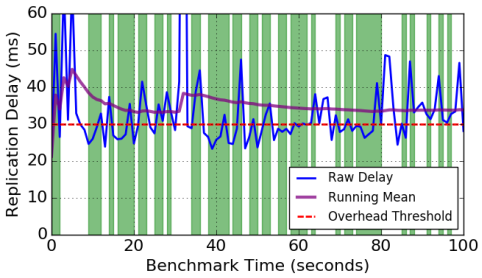
We implemented this policy in NoisePage and show the effect of different thresholds on the replication delay and DMC effectiveness in Figure 5.4. The green shaded regions indicate when metrics collection is enabled. One can see that as we increase the thresh-



(a) 50 ms threshold



(b) 40 ms threshold



(c) 30 ms threshold

Figure 5.4: Varying Delay Thresholds on Dynamic Metrics Collection (DMC) – We execute the TPC-C benchmark, replicating the data between two Type 2 machines. The green regions on the graphs indicate when metrics collection is enabled under DMC.

old, metrics collection is enabled for more time as the higher threshold allows for more overhead incurred by Replicated Training. For reference, DMC enables metrics collection 49%, 90%, and 98% of the time when the threshold is 30, 40, and 50 milliseconds respectively. Further, we can see that whenever the delay exceeds the threshold, the system correctly disabled metrics collection. Doing so lets the replica replay records without the overhead of metrics collection and catch up with the master, reflected by the delay eventually dropping below the threshold.

Special attention should be paid to Figure 5.4(c). When the threshold is close to the average delay without metrics collection (approximately 30 ms), we see a significant performance degradation, illustrated by the running mean being much higher in Figure 5.4(c) compared to Figure 5.4(a) and Figure 5.4(b). This degradation is because the system is rapidly enabling and disabling metrics as the replication delay oscillates around the threshold set by the policy. Toggling metrics can be an expensive operation if done too

frequently, as the system must reset some internal state each time. This additional work, however, allows the system to toggle metrics at runtime without needing to restart the system, unlike other DBMSs [38].

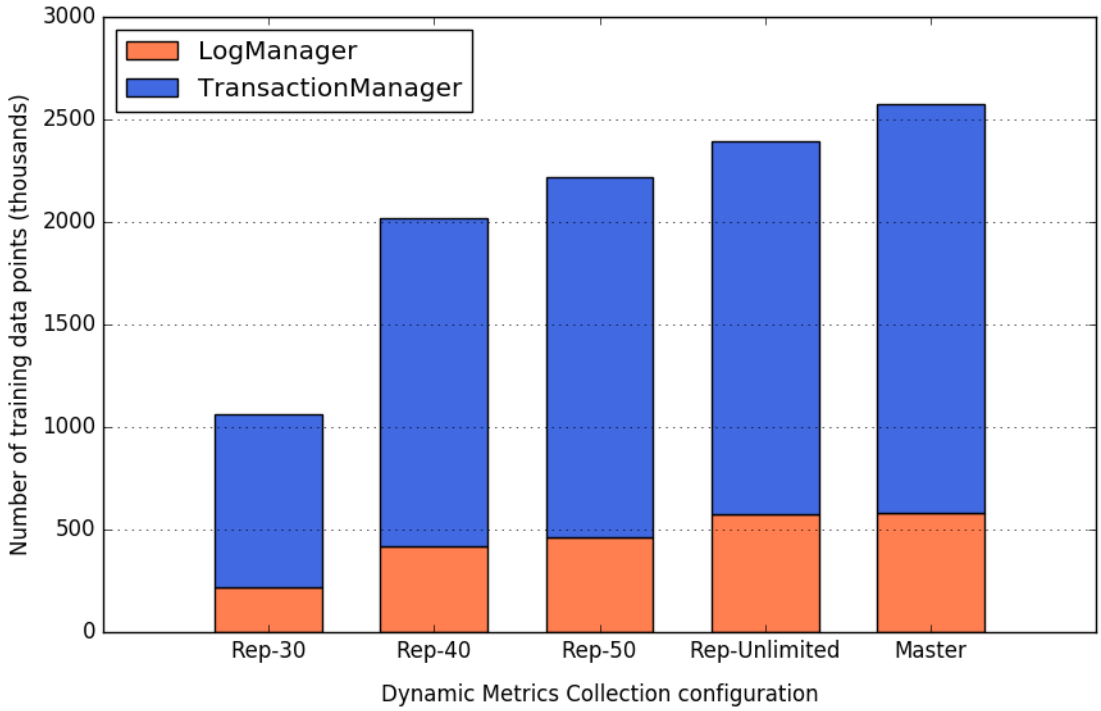


Figure 5.5: Quantity of Training Data Generation – As we increase the replication delay threshold on the replica (e.g., configuration "Rep-30" has a 30ms threshold), DMC generates more training data. "Rep-Unlimited" denotes no threshold. The data sets are separated by each component.

We also suggested that higher thresholds enable metrics collection for more time with DMC, therefore generating more training data. As we see in Figure 5.5, the system generates more training data with a higher delay threshold. We also include the total amount of metrics generated by the master during the same benchmark run for reference, denoted by "Master" in the chart. The difference between "Master" and "Replica-Unlimited" is because the TM generates less data on the replica since replication does not replay aborted transactions.

Processing Time (μs)	Percentage of Samples
0 - 8	16.9
8 - 10	22.0
10 - 12	28.0
12 - 16	17.7
16 - 20	9.0
$>$20	6.4

Table 5.1: Test Set Data Distribution – Processing time distribution in Log Serializer Task for test set (30,000 total samples).

5.4 Self-Driving Models

To show the efficiency of Replicated Training to build useful models, we train a model using the training data generated from the experiments in Section 5.3. We described in Section 3.3.1 how the log serializer task receives redo buffers from transactions and serializes them into log buffers. The model we built estimates the amount of time the log serializer task will take to process some amount of data. the self-driving planning components could use thos model to predict when data might take a long time to serialize and allocate more log serializer tasks to process the redo buffers in parallel.

From the log serializer metrics, we use the number of redo buffers processed and the amount of data (in bytes) in these buffers as our features for the model. Due to the limited feature space, we use a linear regression model. We build our model in Python using SciKit Learn's linear regression implementation [18]. The output of our model is the processing time in microseconds.

To test our model, we generate a test set consisting of metrics generated on the master during a run of TPC-C. We test using metrics from the master because we want to see the effectiveness of models trained using data from the replicas to predict behavior in the master node. Table 5.1 shows the distribution of processing times our model is trying to predict.

Figure 5.6 shows the mean average error (MAE) over the test set for the linear regression model trained on data generated from each DMC configuration. For a baseline accuracy, we also train a model using metrics from the master node, denoted by the "Master" bar. Although these error values are high relative to the processing times we are trying to predict, we believe that with more metrics instrumentation in the system, we will see much better accuracies.

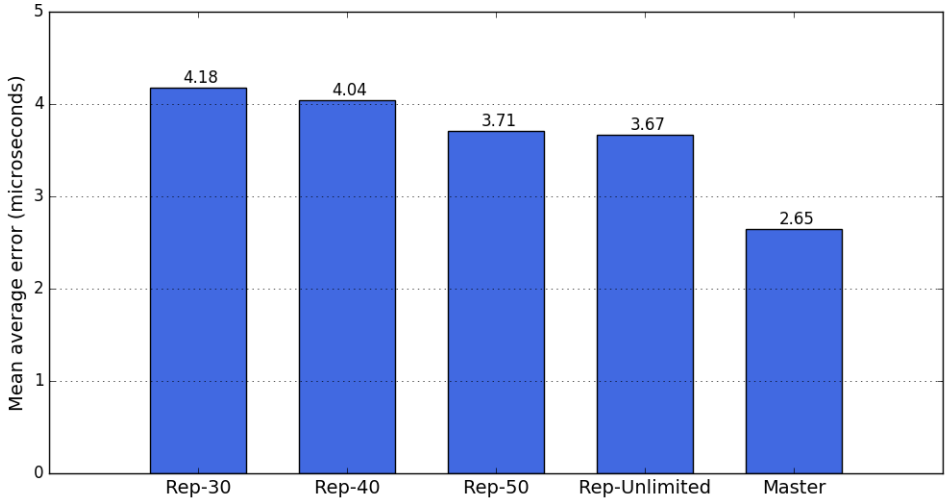


Figure 5.6: Mean Average Error (MAE) Across DMC Configurations – As we increase the replication delay threshold on the replica (e.g., configuration "Rep-30" has a 30 ms threshold), Replicated Training produces more accurate models. "Relica-Unlimited" denotes no threshold.

We see that as the replication delay threshold increases, our model can predict the processing time with lower error, shown by the improvement between the "Rep-30", "Rep-40", and "Rep-50" models. These results suggest that more training data results in better model accuracy. Users can thus reduce the strictness of their DMC policy if they want to improve the quality of the self-driving infrastructure. Further, using Replicated Training for additional training data generation with replicas can lead to better model accuracy for self-driving capabilities.

We note that without DMC (shown by the "Rep-Unlimited" bar), there is still an accuracy difference relative to the model trained on the master. We expect models trained on the master node itself to have the best accuracy, as the training data will reflect the exact hardware environment the master node is running on. This accuracy difference, however, is relatively small compared to the processing times we are trying to predict. Based on the distribution in Table 5.1, this difference accounts for less than a 10% error on more than 60% of the processing times. Further, we expect this difference to be even smaller with better metrics instrumentation.

Chapter 6

Related Work

Similar to our method, iTuned [10] uses unutilized resources on hot-standby replicas to run experiments using the master's workload. These experiments are solely for tuning configuration knobs. While iTuned uses policies to terminate experiments when the hot-standby requires more resources, it does not bound the cost to replication delay, although they suggest it is a small cost. Further, iTuned still relies on human interaction; the DBA must approve or reject tuning recommendations.

Every mission-control DBMS installation uses replicas to serve as a hot standby for the master node. Commonly, these hot-standby replicas are used to service read-only queries made by customers to the system [24, 30, 32, 37]. Some systems will also allow replicas to receive writes, with a change propagation component combined with a consensus protocol to send the changes to other replicas [12, 26].

While Replicated Training uses replicas for data generation, we explore how other systems use replicas for non-hot standby uses. Some systems use a replica as a voter during elections. Google's Cloud Spanner has *witness replicas* that participate in voting during write commits [12]. MongoDB has *arbiter replicas* that are members of the replica set to have an uneven number of voters during master elections [30]. Overall, these voter replicas allow for the easier achievement of quorums without needing the resources of a read or write replica. Contrary to traditional replicas, however, they do not carry a copy of the data. iBTune [43] uses database replicas to seamlessly change buffer pool sizes in nodes to reduce the impact on customers. By promoting a standby replica to master, the demoted master node can modify its buffer pool size, while the promoted replica handles client requests. The demoted master node can then promote itself and take over from where the promoted replica left off, maintaining full availability throughout the process.

Previous research has explored the effect of mixing logging types on replication. Adaptive logging [47] is a distributed recovery technique that combines logical and physical logging (hybrid logging) on a per-transaction basis. This technique identifies which transactions cause dependency bottlenecks and then logs them physically to allow for parallel recovery across nodes. Different from adaptive logging, Replicated Training instead utilizes hybrid logging for targeted training data generation.

QueryFresh [46] combats the problem of high replication delay by using advanced hardware (NVM, InfiniBand) to speed up log shipping during replication. This approach limits the replication delay by reducing the network costs, while Replicated Training limits the delay using controls during log replaying.

Chapter 7

Conclusions and Future Work

We presented a technique called Replicated Training for self-driving DBMS to leverage existing database replicas for training data generation. We dove into the in-memory architecture of NoisePage that supports Replicated Training. We discussed how Replicated Training works and proposed variations that showcase the vast potential of the technique. We further presented Dynamic Metrics Collection (DMC) for controlling the overhead incurred by Replicated Training. We evaluated Replicated Training along with DMC on an OLTP workload and showed its ability to build accurate ML models, with no metrics collection overhead to the master node.

We implemented a simple Replicated Training architecture in NoisePage. Further, we proposed additional variations involving hybrid logging and active learning for better quality training data. These approaches require a more sophisticated metrics instrumentation and an end-to-end DBMS stack that is currently in the works for NoisePage. We are continuing to explore these and more variations to Replicated Training, and plan to implement these techniques in future work.

Bibliography

[1] Amazon. Amazon aurora documentation: Replication with amazon aurora. 2.2

[2] Maria-Florina Balcan and Ruth Urner. Active learning – modern learning theory. Technical report, New York, NY, 2016. URL https://doi.org/10.1007/978-1-4939-2864-4_769. 2.4, 4.2.2

[3] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 1–10, New York, NY, USA, 1995. ACM. ISBN 0-89791-731-6. doi: 10.1145/223784.223785. URL http://doi.acm.org/10.1145/223784.223785. 3.1

[4] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016. 2.3

[5] Transaction Processing Performance Council. Tpc benchmark(tm) c standard specification. Technical report, February 2001. 1.1, 3.1, 3.3.1, 5.1

[6] B.K. Debnath, D.J. Lilja, and M.F. Mokbel. SARD: A statistical approach for ranking database tuning parameters. In *ICDEW*, pages 11–18, 2008. 1.1

[7] Tom Diethe, Tom Borchert, Eno Thereska, Borja Balle, and Neil Lawrence. Continual learning in practice. *ArXiv*, abs/1903.05202, 2019. 1

[8] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proc. VLDB Endow.*, 7(4):277–288, December 2013. ISSN 2150-8097. doi: 10.14778/2732240.2732246. URL http://dx.doi.org/10.14778/2732240.2732246. 1

[9] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R. Narasayya. Ai meets ai: Leveraging query executions to improve index recommendations. In *Proceedings of the 2019 International Conference on Management of*

Data, SIGMOD '19, pages 1241–1258, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-5643-5. doi: 10.1145/3299869.3324957. URL http://doi.acm.org/10.1145/3299869.3324957. 1

[10] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. Tuning database configuration parameters with ituned. *PVLDB*, 2:1246–1257, 2009. 1, 6

[11] Zhiqiang Gong, Ping Zhong, and Weidong Hu. Diversity in machine learning. *IEEE Access*, 7:6432364350, 2019. ISSN 2169-3536. doi: 10.1109/access.2019.2917620. URL http://dx.doi.org/10.1109/ACCESS.2019.2917620. 2.3

[12] Google. Cloud spanner documentation: Replication. 2.2, 6

[13] CMU Database Group. Terrier. 4.2.2

[14] IBM. Ibm db2: High availability through log shipping. 2.2

[15] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. Learned cardinalities: Estimating correlated joins with deep learning. *ArXiv*, abs/1809.00677, 2018. 1

[16] Tim Kraska, Alex Beutel, Ed Huai hsin Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *SIGMOD Conference*, 2017. 1

[17] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. Sagedb: A learned database system. 2019. 1

[18] SciKit Learn. Scikit learn docs: sklearn.linear_model.linearregression. 5.4

[19] Tianyu Li. Supporting Hybrid Workloads for In-Memory Database Management Systems via a Universal Columnar Storage Format. Master's thesis, May 2019. 3.5

[20] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J Gordon. Query-based workload forecasting for self-driving database management systems. In *Proceedings of the 2018 ACM International Conference on Management of Data*, SIGMOD '18, 2018. 1, 2.1, 4.1

[21] Nirmesh Malviya, Ariel Weisberg, Samuel Madden, and Michael Stonebraker. Rethinking main memory oltp recovery. *2014 IEEE 30th International Conference on Data Engineering*, pages 604–615, 2014. 2.2

[22] Ryan Marcus and Olga Papaemmanouil. Deep reinforcement learning for join order enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, aiDM'18, pages 3:1–3:4, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5851-4. doi: 10.1145/3211954.3211957. URL http://doi.acm.org/10.1145/3211954.3211957. 1

[23] V. Markl, G. M. Lohman, and V. Raman. Leo: An autonomic query optimizer for db2. *IBM Systems Journal*, 42(1):98–106, 2003. ISSN 0018-8670. doi: 10.1147/sj.421.0098. 1

[24] MemSQL. Memsql docs: Using replication. **??**, 6

[25] Microsoft. Sql server: Availability modes. . **??**

[26] Microsoft. Sql server: Transactional replication. . 2.2, **??**, 6

[27] Microsoft. Sql server: Statistics. . 4.2.2

[28] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, March 1992. ISSN 0362-5915. doi: 10.1145/128765.128770. URL http://doi.acm.org/10.1145/128765.128770. 3.3, 3.4

[29] MongoDB. Mongodb documentation: Replica set oplog. . **??**

[30] MongoDB. Mongodb documentation: Replication. . 2.2, **??**, 6

[31] MySQL. Mysql documentation: Binary log overview. . **??**

[32] MySQL. Mysql documentation: Replication. . 2.2, **??**, 6

[33] Oracle. Database real application testing user's guide. . 1

[34] Oracle. Database: Oracle autonomous database. . 1

[35] Oracle. Timesten in-memory database replication guide: Overview of timesten replication. . 2.2

[36] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. Self-driving database management systems. In *Conference on Innovative Data Systems Research*, 2017. 1, 1.1, 2.1

[37] PostgreSQL. High availability, load balancing, and replication. . 2.2, **??**, 3.4, 6

[38] PostgreSQL. Run-time statistics. . 5.3

[39] Alexander Ratner, Christopher De Sa, Sen Wu, Daniel Selsam, and Christopher R. Data programming: Creating large training sets, quickly. 2016. 2.3

[40] D. P. Reed. Naming and synchronization in a decentralized computer system. Technical report, Cambridge, MA, USA, 1978. 3.1

[41] Burr Settles. Active learning literature survey. Technical report, 2010. 2.4

[42] Snowflake. Snowflake documentation: Key concepts & architecture. 2.2

[43] Jian Tan, Tieying Zhang, Feifei Li, Jie Chen, Qixing Zheng, Ping Zhang, Honglin Qiao, Yue Shi, Wei Cao, and Rui Zhang. ibtune: Individualized buffer tuning for large-scale cloud databases. *Proc. VLDB Endow.*, 12(10):1221–1234, June 2019. ISSN 2150-8097. doi: 10.14778/3339490.3339503. URL https://doi.org/10.14778/3339490.3339503. 6

[44] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1009–1024, 2017. 1

[45] Jeffrey S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11 (1):37–57, March 1985. ISSN 0098-3500. doi: 10.1145/3147.3165. URL http://doi.acm.org/10.1145/3147.3165. 4.2.3

[46] Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. Query fresh: Log shipping on steroids. volume 11, pages 406–419. VLDB Endowment, dec 2017. doi: 10.1145/3186728.3164137. URL https://doi.org/10.1145/3186728.3164137. 6

[47] Chang Yao, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, and Sai Wu. Adaptive logging: Optimizing logging and recovery costs in distributed in-memory databases. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1119–1134, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3531-7. doi: 10.1145/2882903.2915208. URL http://doi.acm.org/10.1145/2882903.2915208. 2.2, 6

[48] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 415–432, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-5643-5. doi: 10.1145/3299869.3300085. URL http://doi.acm.org/10.1145/3299869.3300085. 1