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

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.

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. We propose a novel technique named Replicated Training that utilizes existing database replicas to generate training data for models used in self-driving DBMSs. This approach load balances the expensive task of data collection across the distributed architecture, as opposed to being done entirely in the master node. It also provides the advantage of more diverse training data in the case where replicas are running in different hardware environments. Under Replicated Training, each replica can dynamically control 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 performance overhead in the master node, and using Replicated Training reduces this overhead while still ensuring that replicas keep up with the master. Finally, we show that Replicated Training produces ML models that have accuracies comparable to those trained solely on the master node.

Acknowledgments

I love Jello. This stuff has fed me from the day I proposed until the day I defended. Jello is my friend, my sustenance, my very being.

Oh yeah. My advisor is cool too.

In addition, **Catherine Copetas** and **Sharon Burks** should be mentioned in a **large** font in everyone's Acknowledgements section, since they said so. (It is a required part of the thesis formatting guidelines.)

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Database management systems (DBMSs) are notoriously hard and expensive to manage and tune. Configuration and monitoring of physical database design, knobs, and hardware provisioning can have significant effects on the performance of the system. Currently, a database administrator (DBA) has the role of this configuration and monitoring. DBAs, however, are expensive to hire and do routine work.

Previous research has explored using machine learning (ML) to automate the runtime behavior of DBMSs. Some systems have put ML at the core, creating a new class of so-called self-driving DBMSs [26, 28]. These systems use ML for high-level decision makings, such as workload prediction [13], or system-wide tasks such as transaction scheduling [11]. Analogous to this, other systems use ML to solve component-scoped challenges by creating so-called learned components. Examples include learned index structures [10], cardinality estimation [9], and join ordering in the query optimizer [15].

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 [6, 33, 36], or make recommendations, such as adding or dropping indexes [5]. These can then be used by the DBA to tune their system.

All these approaches face the same problem: the ML models that power these techniques require a lot of training samples. These models commonly use metrics across the DBMS as training data. These metrics include query arrival rates [13], latch contention in the transaction manager citation for new paper, or disk write performance in the log manager citation for new paper. The more diverse the training data the system generates, the more models we can train. 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 [4]. Accounting for such differences during training data generation will make the models robust and resilient to dynamic settings.

One possibility is to generate training data for models in an offline setting (i.e., not during production execution). Some systems [25] capture production workloads to allow DBAs to try out different system configurations on a snapshot of the database. Other tuning systems [6, 33, 36] 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.

In contrast, an online approach collects training data that the production system generates during live execution. Systems such as IBM's LEO [16] and Automatic Indexing [5] use metrics generated from live query execution as training data. These systems can also tune their models on the fly. This technique is straightforward to instrument, but 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 have serious performance risks. Many customers expect these features to result in zero regressions on their performance [5]. This includes both the overhead incurred from running these systems and the potential degradations of harmful recommendations or configurations.

1.1 Motivation

The demand for self-driving DBMS has grown with the growing size of data sets and the increased desire to run complex analytics over that data [28]. 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 [3]. Self-driving DBMSs 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 NoisePage, the training data, or input, to the ML models is metrics outputted during the execution of the system. For example, the logging component may output as a metric how long it took to write some amount of data to disk. As we discussed prior, this metrics collection comes at an overhead. To measure the effects of metrics collection in a DBMS, we execute TPC-C on PostgreSQL with and without metrics collection, taking the average over five runs. In our benchmark, the number of worker 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 results show that, on average, this metrics collection overhead was around 11%. Users that may not want to pay this overhead on the master node, but still want to take advantage of the self-driving component of the DBMS can use the Replicated Training technique to generate the data needed for the ML models.

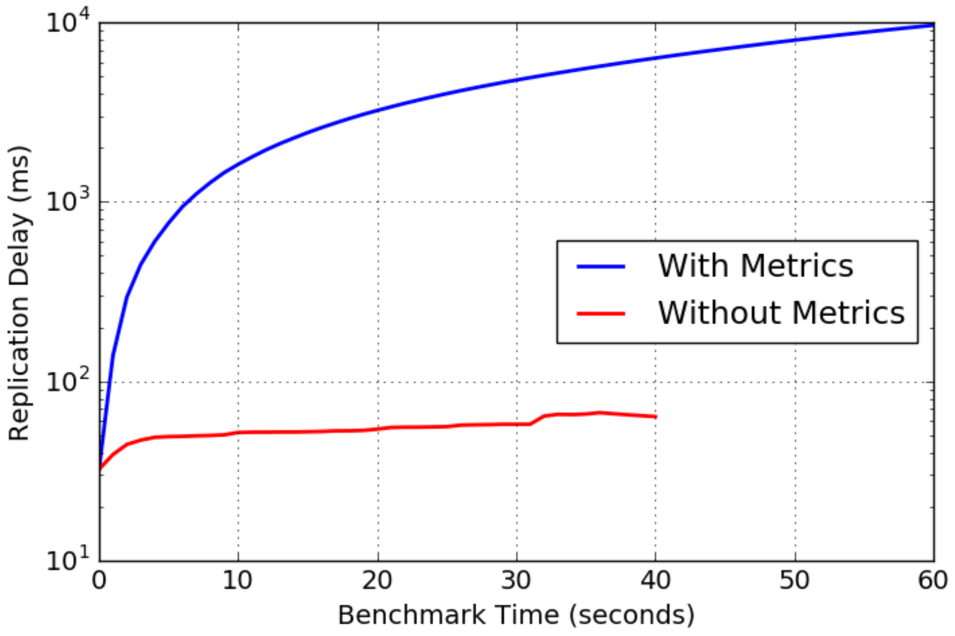


Figure 1.1: Metrics overhead on replication delay in NoisePage - Effects of replication delay when metrics collection is enabled over TPC-C with 4 warehouses and a 10k txns/sec arrival rate

Replicated Training allows offloading metrics collection to replica nodes; however, we must 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. To observe the impact of these degradations on replication delay, we run the TPC-C benchmark on NoisePage with metrics collection enabled and disabled. The database is replicated asynchronously across the network on two machines with Intel(R) Xeon(R) CPU E5-2420 CPUs with six threads (12 hyper-threads). We look at the running mean of the replication delay over the benchmark entire benchmark run. The results in figure fig. 1.1 show that metrics collection results in a significant increase in replication delay. We note that the benchmark run when metrics collection is enabled lasts longer because it takes longer for the replica to become fully in-sync with the master. In our first data point, the replication delay with metrics collection enabled and disabled is 1.48ms and 1.33ms, respectively. This difference is an 11.6% overhead, supporting the behavior we saw in PostgreSQL. For our last data point for each benchmark run, the replication delay with metrics enabled is $152\times$ worse compared to when replication is disabled. When metrics collection is enabled without controls, the replica falls behind to the master. It is never able to catch up, resulting in this significant difference in the replication delay at the end of each benchmark run.

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 that its ability to produce accurate models without needing to pay the penalty of metrics collection on the master node.

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

Chapter 2

Background

2.1 Self-Driving Database Management Systems

Talk about the components of the self driving infrastructure (modeling, prediction)

Talk about what actions are, give some examples

2.2 Replication

Practically every production system 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 data. 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.

There are two replicated commit types we should be aware of. In synchronous replication, a user is not told that their change has been persisted in the database until it has been replayed by at least one replica. In asynchronous replication, the user may be told their change has been persisted before a replica has replayed the change. Synchronous replication minimizes data loss in the event of a master node failure, as all commited changes are guaranteed to be on some replica. Synchronous replication, however, adds more latency to requests as there is an extra network trip incurred by the replica confirming it has replayed the request.

System	Architecture	Logging Type	Replicated Commit Type
MemSQL	Master-Slave	Logical	Asynchronous
MongoDB	Master-Slave	Logical [21]	Asynchronous
MySQL	Master-Slave [24]	Both [23]	Both [24]
PostgreSQL	Master-Slave	Physical [29]	Both [29]
SQL Server	Publisher-Subscriber	Logical [19]	Both [18]

Table 2.1: Replication architecture for various DBMSs - Systems that do log shipping can still have differences in their replication architecture.

Production systems handle how changes are communicated between the master and its replicas in different ways. Most systems [8, 19, 22, 24, 27, 29] 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. Table 2.1 shows the replication architecture of various log shipping DBMSs.

Other systems [1, 7, 31] abstract away how replication is done by letting an underlying cloud object storage handle the replication. To the system programmers, this appears as if all replicas are reading from the same logical copy of the data.

Regardless of how replication is done, all DBMSs that support replication will suffer from replication delay: the difference in time between when a change is applied on the master, and when the change is replayed on a replica. Many DBMS users have strict service-level agreements (SLAs) that they must adhere to. A delimited replication delay is a common SLA users expect their DBMS to support in order to ensure a bounded degree of consistency between master and replica nodes. Low replication delays also reduces 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.

2.2.1 Logging Types

If systems employ replication using log shipping, they will have to make the important decision of what type of logging to use: physical or logical logging. Physical logging involves the 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 changes made to the data [35]. Most commercial systems that employ logical logging will implement command logging, where the transactions themselves (or commands) are logged [14]. Each logging type has its own tradeoffs. Physical logging carries more overhead during execution because it produces more logs than logical logging, but is faster and more parallalizable during recovery. Logical logging on the other hand has less overhead during execution (typically a single log per transaction), but is more expensive to replay during recovery. Some systems [35] implement hybrid logging, which is a mix of logical and physical logging, to gain the benefits of each type.

2.3 Training Data Collection

Effective training data generation is a hard and expensive task for production ML models. Companies have entire teams of Data Analysts and spend thousands in compute resources to generate enough data for their models to produce reasonable ouputs. write some more, this paragraph doesn't feel right/finished

There are two important facets of traning data generation that we will concern ourselves with: performing actions to generate 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 a good comparison to self-driving DBMSs as they are both expensive to deploy, and must be able to handle completely new environments or workloads.

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 its environment through use of high-tech cameras. Apart from the difficulty of building all the technology to capture the car's environment, this process is extremely expensive. In the case of supervised learning models (should i explain supervised vs unsupervised), the sampled data must then be labeled. Often times this requires a large amount of humans manually labeling objects in the car's film. For other domains, such as ML for medicine, labeling can be incredibly expensive as often times few people are qualified enough to accuretly produce labels.

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

A different approach is to build an environment where the workload can be simulated to generate training data. 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 database systems are simply too large and expensive to duplicate in a simulated environment. Secondly, these simualated environments approximate a real environment. A car test track would not perfectly replicate the chaoctic and variable reality of driving. 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. Should I discuss gyms? or is that another thing that isn't relevant here

Chapter 3

Related Work

Orthogonal to our method, iTuned [6] uses unutilized resources on hot-standby replicas to run experiments using the primary's workload. These experiments are solely for the purpose of tuning configuration knobs. While iTuned uses policies to terminate experiments when the hot-standby requires more resources, it does not bound the cost to recovery time, although they suggest it is a small cost. Further, iTuned still relies on human interaction, the DBA must approve or reject 's 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 [17, 22, 24, 29]. Some systems will also allow replicas to receive writes, with a change propogation component combined with a consensus protocol to send the changes to other replicas [7, 19].

Some systems use a replica as a voter during elections. Google's Cloud Spanner has ”witness replicas” that participate in voting during write commits [7]. MongoDB has ”arbitrer replicas” that are members of the replica set to have an uneven number of voters during primary replica elections [22]. Overall, these voter replicas allow for easier achievement of quorums without needing the resources of a read or write replica. Contrary to traditiona replicas, however, they do not carry a copy of the data.

iBTune [32] uses database replicas to seemlesly change buffer pool sizes in nodes to reduce impact to 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 off where the promoted replica left off, maintaining full availability throughout the process.

Mixing of logging types has been studied in the past. Adaptive logging [35] is distributed recovery technique that mixes logical and physical 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.

QueryFresh [34] combats the problem of replica divergence by using advanced hardware (NVM, InfiniBand) to speed up log shipping during replication.

Chapter 4

System Architecture

in-memory database management system (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. NoisePage was originally built to be a single node system, and we will discuss in section 4.6 the addition of hot-standby replication. Transactions can be executed synchronously or asynchronous, and crash recovery is possible. We now discuss the architectural components of NoisePage relevant to supporting replicated training.

4.1 Transactions

Transactions are the core atomic unit of work in a DBMS. The transactional approach taken by NoisePage is extremely conducive to a streamlined logging, recovery, and physical replication scheme. NoisePage's transactional engine is a multi-versioned delta store [30] that supports `SNAPSHOT-ISOLATION` [2]. The transactional engine is coordinated by the `TransactionManager` component. In our 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 4.3) is enabled, transactions are considered active between when they begin, and when they commit or abort. When logging is disabled, transactions are active between when they begin, and when their changes have been serialized by the log manager (described in section 4.3). This can be further extended by enabling synchronized commit, which guarantees that a transaction is active until its changes have been persisted in disk.

Tuples are uniquely identified by a *TupleSlot* object that stores offset information about the tuple. The `TupleSlot` is created when a tuple is first inserted into a table. Each table has an additional, invisible column reserved for storing the head of this version chain.

Transactions update the database by adding their changes to a version chain for a specific tuple. To read a tuple, a transaction traverses the version chain until it sees the first change visible to that transaction. Changes are stored newest-to-oldest in the version chain to facilitate fast reads.

These changes, or delta records, are not an updated copy of the tuple, but rather the after-image changes made by the transaction on the tuple. There are four types of records: Redo, Delete, Commit, and Abort. Aside from the changes or transactional action, the delta records hold additional information such as a transaction *start* timestamp and what database and table were modified. These pieces of information are necessary for ensuring correct replaying of the record during recovery or replication. It is important to note, however, that delta records do not hold all the information needed to accomplish this. It is the role of the log manager, discussed in section section 4.3, to serialize any additional information needed 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 are given 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 hand off 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.

Comitting a transaction is straightforward: the transaction will write in a commit record to the redo buffer, and hand it off to the log manager. During commit time, the oldest active transaction timestamp is also polled from the timestamp manager, 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 hands off its buffer to the log manager once it is full. Due to this, it is possible for an aborting transaction to have already persisted records. In order for correct behavior during recovrey and replication, an aborting transaction that has previously flushed records must also flush an abort record. Without this abort record, a downstream consumer would be unable to differentiate such changes from a long running transaction. From observation, however, it is rare that an aborting transaction will ever flush a buffer, as the amount of data generated by an OLTP transaction rarely fills up an entire redo buffer.

4.2 Timestamp Manager

The timestamp manager is a central component 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 when it succesfully commits.

The timestamp manager also maintains an *active transaction set* that contains the *start* timestamp of every active transaction currently running in the system. Using this, the timestamp manager can be polled for the oldest active transaction timestamp (smallest *start* timestamp in the set). If there is no oldest active transaction, we indicate so using a special value. The importance of this information is dicusssed in section 4.4. A transaction is not removed from this set until its contents have been serialized by the log manager (discussed in section section 4.3). This prevents background garbage collection (GC) from cleaning up the transaction before its changes have been persisted.

Fast perfomance of polling for the oldest active transaction timestamp is a crucial because it must be done during the critical section of every committing transaction. When logging is disabled, the number of active transactions is bounded by the number of worker threads availible to the system. When logging is disabled, however, 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. We instead keep a cached oldest active transaction timestamp in the timestamp manager. During commit, transactions atomically read this value instead of scanning the active transaction set. The cached value is periodically refreshed by background Garbage Collection (default to every 5 ms). While the cached value may 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.

4.3 Logging

Changes to NoisePage are persisted on disk using write-ahead logging [20] through a dedicated component called the log manager. The log manager is in charge of serializing delta records such that they are entirely replayable on their own without any additional in-memory metadata, such as in the case of recovery after system crash or replication. To do so, the log manager coordinates multiple parallel tasks structured in a Producer-Consumer

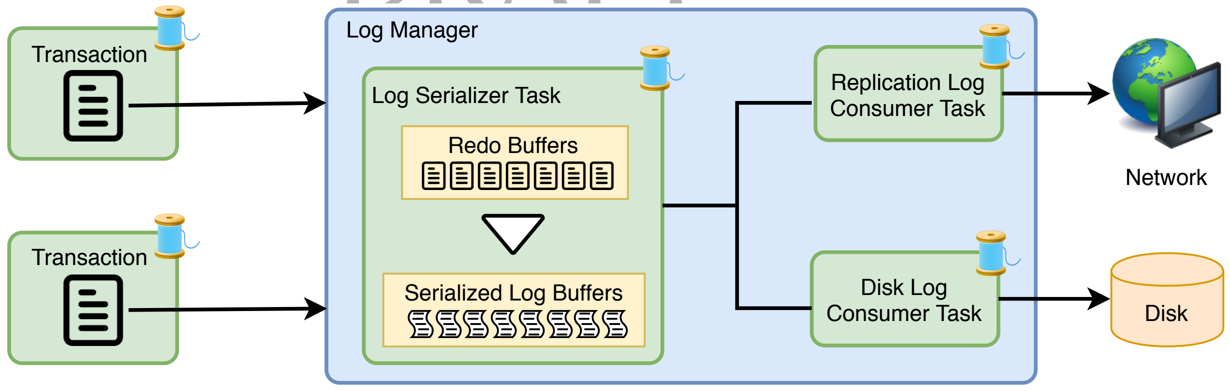


Figure 4.1: Log Manager Architecture - The log manager receives as input log record buffers from transactions, serializes these records, and sends them to different destinations (e.g. disk).

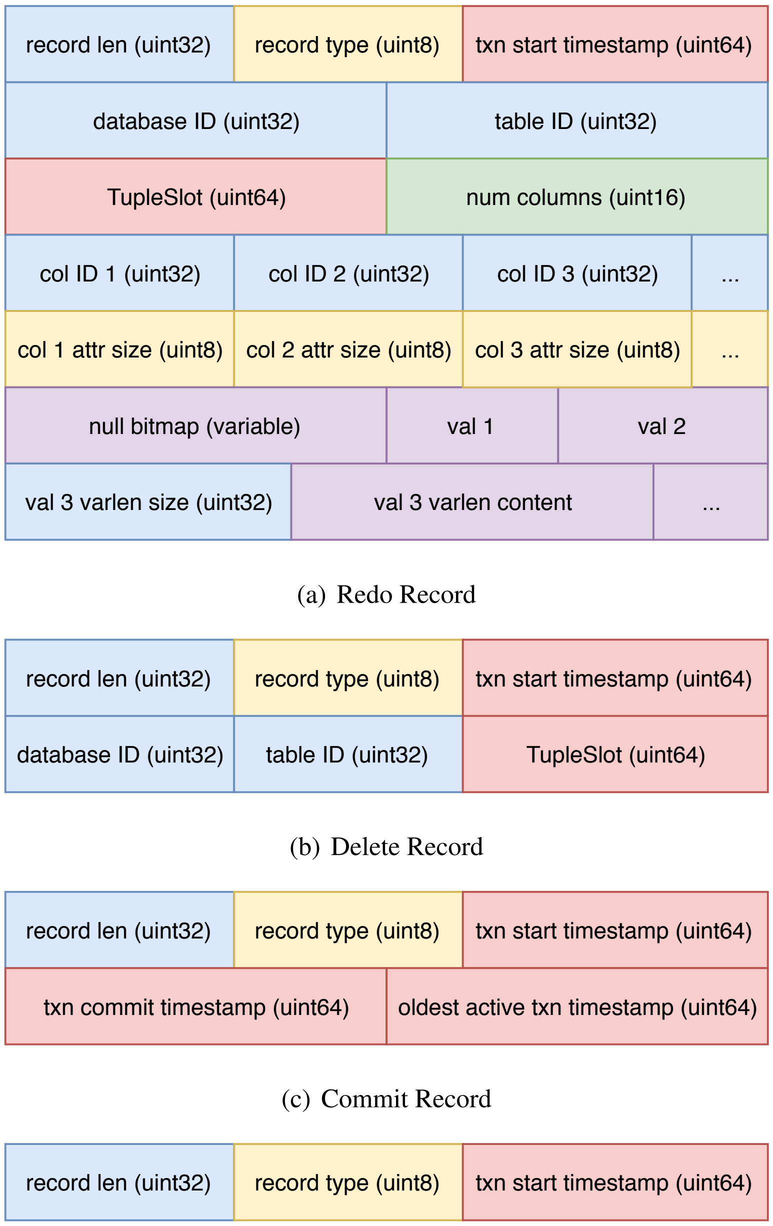
architecture shown in fig. 4.1. The producer, a log serializer task, feeds serialized logs to multiple log consumer tasks (e.g. disk log consumer task).

4.3.1 Log Serializer Task

The log serializer tasks receives redo buffers from transactions, and serializes their contents 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 varlens in NoisePage are not inlined in the delta record, but rather stored in a separate memory location, with just a pointer to the varlen entry stored in the delta record. The serializer task must thus fetch the varlen entry and serialize it inline. Serialization also removes any padding used in the in-memory log record.

The serialized format of the log records is shown in fig. 4.2. The delete, commit, and abort records have fixed serialized sizes of 29, 29, and 13 bytes respectively. Due to the variability in data being updated, the size of serialized `RedoRecords` varies depending on the delta record's contents. Over a run of the TPC-C benchmark with 4 warehouses, the average `RedoRecord` size is 100 bytes. For reference, the same execution of TPC-C will generate approximatly 1 GB of total log data.

The serialized ordering of the records is important to ensure correct replayability. Recall from section 4.1 that transactions can hand off record 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 non-serial order. The only guarantee that is made is that



(a) Redo Record

(b) Delete Record

(c) Commit Record

(d) Abort Record

Figure 4.2: Log record serialization formats - Along with the log record, additional information must be serialized to ensure replayability.

records for an individual transactions appear in order that they were created, with a commit or abort record always being the last record to appear. As we will discuss in section 4.4, this guarantee, along with the oldest active transaction timestamp discussed in section 4.1, is enough to achieve a consistent snapshot of the database after replaying the log.

4.3.2 Log Consumer Tasks

The buffers generated by the log serializer are handed off to an arbitrary number of log consumers. Each consumer is given a copy of the serialized buffer so they can each work independently of eachother, preventing a slow consumer from slowing down the others. Currently, NoisePage suppors two consumers: (1) a disk consumer that writes logs to a file on disk, and (2) a replication consumer that sends logs over network to replicas.

The disk consumer task waits until it receives buffers from the serializer task, and writes them to a log file specified by the user. Writing to the log file is fast because we are always doing sequential writes. For good performance in persisting the log file to the disc, we take advantage of batch 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 milliseconds or if more than one megabyte of data has been written since the last persist. The configurability of this process will allow the self-driving infrastructure to find the optimal combination of settings for a given workload.

The replication consumer task also receives buffers from the serializer task and sends them over the network to any replicas listening to the master. In order to minimize the replication delay, there is no batch commit done. Instead, serialized logs are sent as soon as they are handed off to the replication consumer task. The network protocol used for sending logs between nodes is described in detail in section 4.5.

4.4 Recovery

Recovery in NoisePage is performed by a component called the recovery manager. The recovery manager receives serialized log records from an arbitrary source, and replays them to produce a consistent view of the database.

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

Abstracting away the source of log records gives us the great advantage that we can implement the replaying component of replication for ”free”. By simply having the source of records be a stream of logs over network, a standby replica can instantiate a recovery manager to replay the log records arriving from the master node. This approach for replication is also done by other systems like PostgreSQL. This, however, requires the processing model of the recovery manager to be a streaming model. We can not take advantage of cases when all the log data is availible apriori, as is the case during single node crash recovery. Other recovery algorithms, such as ARIES [20], take advantage of having access to all the data from the start and trim out unecessary processing.

4.4.1 Log Record Replaying

The API for updating tables in the system (`SqlTable`) allows for easy replaying of records. Recall from section 4.1 that transactions must write their changes as delta records in their private buffers. The system takes advantage of this by passing `SqlTable`s a pointer to these records in the buffer. This prevents having to make an additional copy of the data. Since recovery deserializes `RedoRecord`s, there is no need to transform the data, the `SqlTable` API simply accepts these `RedoRecord`s directly.

The `TupleSlot` contained in the replayed record is no longer valid during recovery, as it represented a unique memory location prior to recovery. Instead, we use it as an internal mapping from the original `TupleSlot` to the new one when the insert is replayed. Using the mapping, we can correctly identify what `TupleSlot` updates should be applied 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 largely easy, as they simply appear as updates to the catalog tables. Despite this, additional logic is needed to reinstantiate certain in-memory objects in the system, such as indexes, views, or user tables.

While its is possible that we could replay changes as we see them in the log and roll them back in the case of aborts, we will see in section 4.4.2 that we must defer all updates until we see a commit or abort record anyway. Buffering also gives us the added advantage that we prevent the unnecessary of replaying records for aborted transactions. When we see an abort record, we clean up and discard any records buferred for that transaction. When we see a commit record, we process the transactions as described in section 4.4.2.

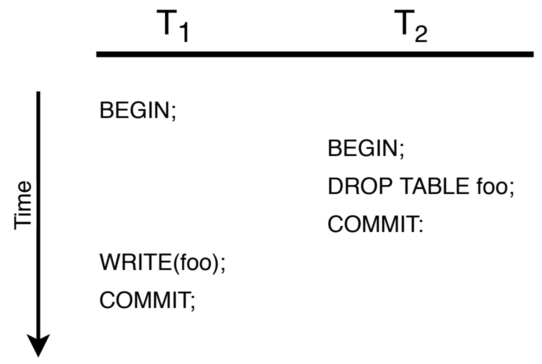


Figure 4.3: Schedule involing DDL changes - This schedule is allowed under `SNAPSHOT-ISOLATION`, but can create problematic races involving the `DROP TABLE` command.

The transaction's changes are alwas applied in the order they were made prior to recovery.

4.4.2 Transaction Replaying

Special considerations must be made when replaying transactions because of the streaming processing model of recovery and the design of our transactional engine. Recall from section 4.3 that the only guarantee we have about the ordering of logs is that changes for an individual transaction are in the order they occured. There are no guarantees about how transactions are ordered relative to eachother in the log, so it is the responsability of the recovery manager to execute them in an ordering that results in a consistent view of the database.

Recall from section 4.1 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 eachother on a per tuple basis. Thus, we are guaranteed that there are no dependencies between transactions that executed concurrently, and all committed transactions that are replayed will succesfully commit. Further, because we use physical logging, all the values being written during log replaying are predetermined - i.e. no writes are based on reads or randomization. Based on these two guarantees, we are able to replay transactions sequentially (i.e. a transaction commits before the next one is allowed to begin).

We have determined that we can execute transactions sequentially, but the order in which we execute them is also important. Consider the schedule in fig. 4.3 which is allowed under `SNAPSHOT-ISOLATION`. While 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 foo occurs before T_2 deletes foo. There are no guarantees about ordering of logs between transactions, so its possible for T_2's changes to appear before T_1's changes in the log. Even worse, if T_1 is a long running transaction, it's 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, we must execute the transactions in the order that they were created i.e. ordered by their *start* timestamp.

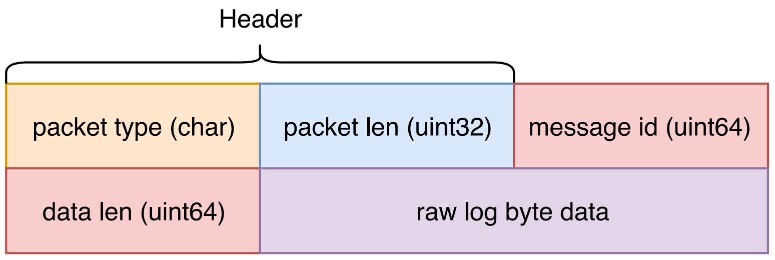
A simple aproach for accomplishing this would be as follows: A transaction T_i with *start* timestamp i is safe to replay after we have replayed transaction T_{i-1}. The transactional engine, however, does not guarantee that consecutive transactions have consecutive *start* timestamps. Additional processes, such as garbage collection or assigning commit timestamps, also receive timestamps from the timestamp manager.

The solution to this problem is by using the oldest active transaction timestamp described in section 4.1. When a transaction commits, the oldest active transaction timestamp at the time of commit is included in the commit record (shown in fig. 4.2(c)). When a transaction is entirely deserialized, rather than executing it right away, we defer its execution. Using the oldest active transaction timestamp i stored in the commit record, we then execute, 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 we execute all deferred transactions.

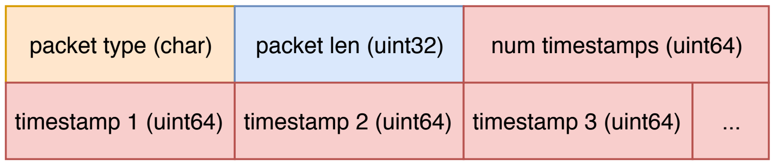
Once again, consider the schedule in fig. 4.3. Suppose the *start* timestamps of T_1 and T_2 are 1 and 2 respectively. The commit record of T_2 will indicate that the oldest active transaction timestamp at commit time is 1. If T_2 is serialized **before** T_1, it will be deferred because $1 < 2$. Eventually T_1 is deserialized and executed, followed immedietly by T_2, because there were no older transactions at the time T_1 committed.

4.5 Internal NoisePage Protocol

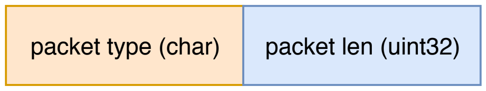
Internal NoisePage Protocol is the network protocol used to communicate between nodes of NoisePage. Internal NoisePage Protocol uses TCP/IP sockets to send data between the master and replica node(s). TCP is used over UDP for its message delivery guarantee; it's important that no log data is lost over network. NoisePage has a network layer which 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 Internal NoisePage Protocol and the PostgreSQL Wire Protocol (described in [12]).



(a) `ReplicationDataPacket`



(b) `CommittedTransactionsPacket`



(c) `EndReplicationPacket` and `ReplicaSyncedPacket`

Figure 4.4: Internal NoisePage Protocol packet types - Packets are minimal to reduce network congestion and speed up packet processing

Messages are packets consisting 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 entire size of the packet. This portion of the protcol resembles the PostgreSQL Wire Protocol. To support a new protocol, users write a `PacketWriter` class, which contains logic to write the payload of the packets. Finally, users write a `ProtocolInterpreter` class which parses and processes the payload based on the packet type.

The current packet types for Internal NoisePage Protocol are shown in fig. 4.4. The `ReplicationDataPacket` (fig. 4.4(a)) holds variable-length portions of the serialized log data from the master node's log manager to the replica. The `CommittedTransactionsPacket` (fig. 4.4(b)) is sent from the replica to the master to notify when the replica has succesfully replayed and committed transactions. This provides support for synchronous replication. The *start* timestamp of the replayed transactions is stored in the packet's payload. The `EndReplicationPacket` is sent by the master and tells the replica to end replication. The `ReplicaSyncedPacket` is sent from the replica and notifies the master that the data in both nodes is in sync. Both these packets (figure fig. 4.4(c)) require no payload as the type in the header entirely describes the purpose of the packet.

4.6 Replication

An overview of the replication architecture in NoisePage is shown in figure fig. 4.5. 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 log manager and the replica's network layer. When the logs are serialized by the log manager, they are placed into a `ReplicationDataPacket` and shipped over the network by an Internal NoisePage Protocol specific `PacketWriter` described in section 4.5. The packets reach the replica's network layer and are handed off to the recovery manager running in the system. We implement a `LogProvider` class called the `ReplicationLogProvider` to parse the log data from these arriving packets into log records. For replaying these logs, we discussed in section 4.4 that we can accomplish replication using the same recovery manager logic used for crash recovery.

If synchronous replication is enabled, the replica will send the master a `CommittedTransactionsPac` when a transaction is replayed. If the replica is in sync with the master (i.e. there are no more logs left to replay), it will also send a `ReplicaSyncedPacket`.

The recovery manager runing on the replica will sit in a loop, continually processing log records as they arrive over the network. This is unlike in crash recovery where the recovery manager will terminate when it reaches the end of the log. Instead, the master can terminate replication with a replica by sending it an `EndReplicationPacket`.

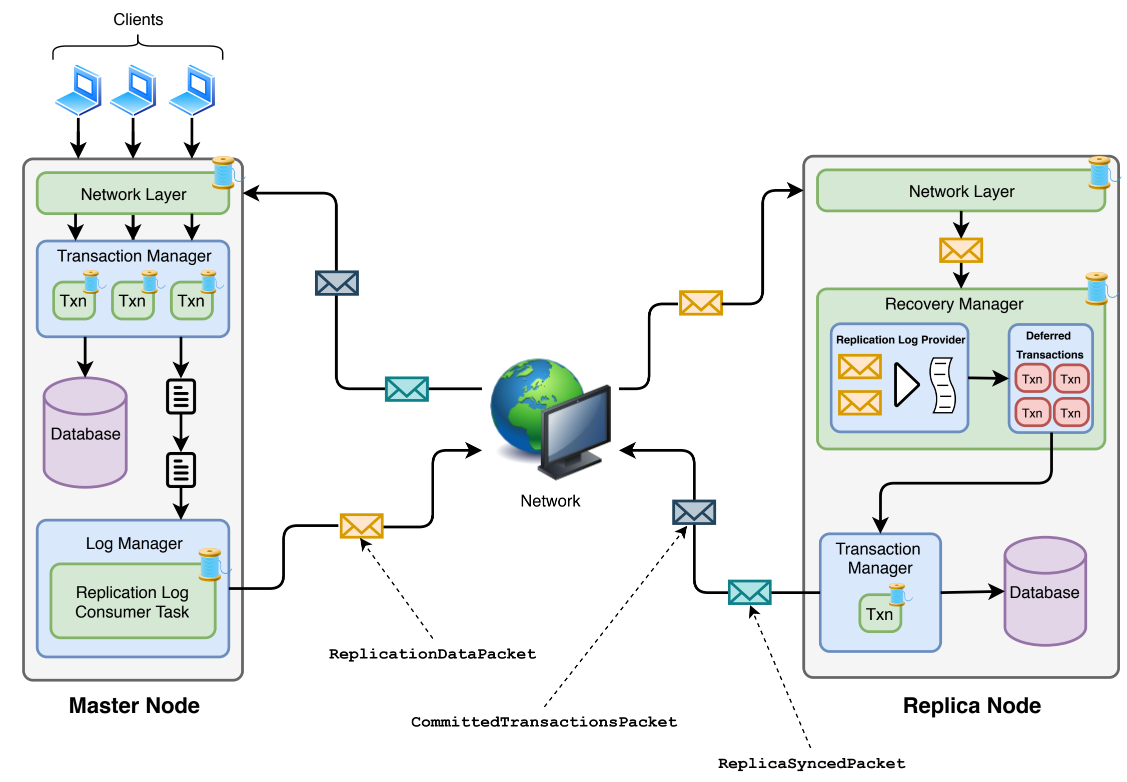


Figure 4.5: Replication Architecture - While there are other components in NoisePage, this highlights the processes involved in replicating data between a master and replica

Chapter 5

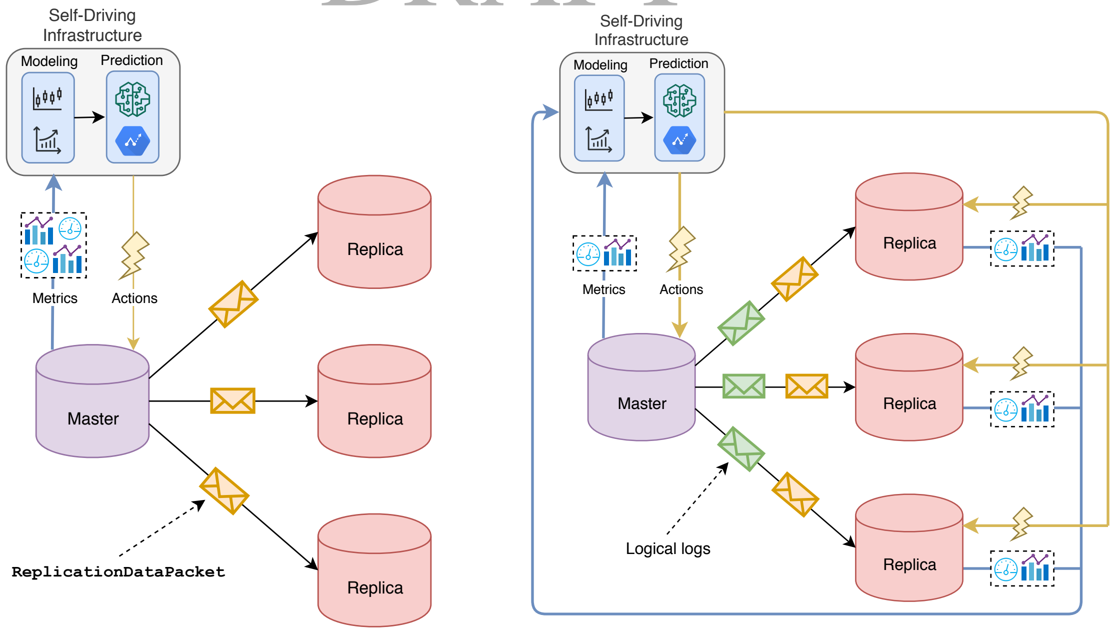
Replicated Training

We discussed in section 1.1 how self-driving capabilities can be beneficial to a system, but metrics collection can be detrimental to performance. Replicated Training leverages the existing architecture of a distributed DBMS to power self-driving infrastructure without paying the penalty of supplying training data solely from the master. Replicated Training can adhere to performance requirements set by the user through dynamic metrics collection while still generating useful models for the system. Lastly, we propose how Replicated Training can be taken further to help with action exploration in a self-driving DBMS.

5.1 Replicated Training Architecture

The main principle of Replicated Training is using resources available on existing replicas to generate additional training data for models in a self-driving DBMSs. In a traditional self-driving DBMS architecture (shown in fig. 5.1(a)), the master node generates metrics by executing queries. These metrics are aggregated and sent to the self-driving infrastructure, which uses the metrics as training data to train 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 [13]. The self-driving infrastructure can then use this model to predict future workloads and recommend actions such as building an index on a column the self-driving infrastructure expects will be frequently scanned. The key issue is that metrics collection penalizes performance on the master node, which some DBMS customers may not be willing to pay.

With Replicated Training, we can balance this cost by distributing the task of training



(a) Without Replicated Training (b) With Replicated Training

Figure 5.1: Self-driving DBMS Architectures - Replicated Training enhances a self-driving DBMS by leveraging database replicas for training data generation

data generation across the entire distributed topology. As shown in fig. 5.1(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 traditional self-driving architecture. 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 5.3, the self-driving infrastructure can also choose to test actions on a replica before applying them on the master node.

5.2 Dynamic Metrics Collection

Customers often have strict SLAs for their DBMS deployments. For instance, some customers may require that the replication delay between their master and replica nodes is at most 100 milliseconds. We showed in fig. 1.1 that metrics collection can cause replication delay to significantly increase, as the overhead incurred slows down the replica's log replaying, leaving it unable to keep up with the master node. To combat this issue, we propose dynamically controlling when to collect metrics under Replicated Training. Instead of metrics collection being permanently enabled, we enable metrics when replication complies with some policy. We also tune the granularity of metrics collection under the policy, which we call Partial Metrics Collection. Finally, we can dynamically perform hybrid logging to exercise more layers of the system in the replica node.

5.2.1 Metrics Collection Policies

To control when metrics collection is enabled, we can impose a policy that determines when to enable or disable metrics collection. For this section, we will assume that metrics are either entirely enabled or disabled, but as we will see in section 5.2.2, this is not always the case.

In the case of Replicated Training, we want to ensure that the replica can keep up with the master. Since replication delay gives a numerical estimate of this goal, we can construct a policy around this value. The simple policy we implement in NoisePage is as follows: if the replication delay during the previous time window (default: 1 second) exceeds a fixed threshold (e.g., 100 Milliseconds), disable metrics collection until the delay drops below the threshold, after which the system can enable metrics once again. Assuming a reasonable threshold, 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.

The above policy can be considered a strict policy as it firmly adheres to the user's requirements to minimize the effects of Replicated Training. In contrast, one can use a more flexible policy which allows Replicated Training to operate longer, while allowing replication delay to fluctuate more but still under control. For example, under a flexible policy, the replica may keep a history of replication delays, and enable metrics collection when the current delay is within one standard deviation of the mean delay.

Flexible policies have two main advantages. The first is that in theory, metrics collection will be enabled for more time and thus will generate more training data relative to a strict policy. The second is that a flexible policy can produce more diverse training data. Consider an environment with high resource contention. It is possible that the replica can only keep up with the master when resources are abundant. Therefore, under a strict 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 utilizing a flexible policy, we can still generate training data during periods of high resource contention.

5.2.2 Partial Metrics Collection

The metrics collection in NoisePage allows for fine-grained metrics control than simply ”all-or-nothing.” The system can toggle metrics for individual components during runtime, which we call Partial Metrics Collection. By enabling or disabling metrics for parts of the system, we 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 to prioritize what metrics to keep enabled, and which ones can be disabled. For example, if our replica is violating our collection policy, we can use active learning to determine what components our ML models no longer need training data from, and disable metrics for those components. Inversely, if we are complying with our collection policy, we may choose to enable metrics for component's whose collection is currently disabled if our models require their specific training data.

5.2.3 Dynamic Hybrid Logging

We discussed in section 4.4.2 how NoisePage uses physical logging to replicate data across machines. Physical logs have the great advantage that they can be replayed easily 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, for example, sending the raw SQL string, requires processing through the entire DBMS stack (e.g., parsing, optimizing, execution) to replay. 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 fig. 5.1(b)).

Similar to the policies mentioned in section 5.2.1, we propose two policies to control which read queries we 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, we sample a binomial distribution to decide if we send this query to the replica. By adjusting the success probability of the binomial distribution, we can control the number of queries sent to the replica, thereby limiting the amount of extra work. The second approach is similar to the active learning approach proposed in section 5.2.2. We can use active learning to determine what components of the system we need to exercise and choose queries based on that. For example, if we have a model on executing aggregations that active learning determines needs more training data, we can sample queries containing `GROUP BY` clauses. This technique would work well with the active learning based partial metrics collection approach proposed in section 5.2.2, as we can use active learning to decide which components we need metrics for and which queries we sample to target them.

5.3 Action Exploration

An essential capability of self-driving DBMSs is applying actions to the system to optimize for some objective function. The user can set objective function (e.g., maximize throughput, minimize latency) depending on their system requirements. Actions applied to the master are not always guaranteed to impact the objective function positively. The self-driving infrastructure might incorrectly predict an action to have some impact on the system, but it actually 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 a candidate replica. The self-driving infrastructure then monitors the metrics received from Replicated Training on the candidate replica to evaluate the performance impact of the candidate 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. Using Replicated Training prevents the performance monitoring of the action from having severe performance degradations by using the dynamic metrics collection described in section 5.2.

Chapter 6

Evaluation

We now evaluate and analyze 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 replication between two nodes experiments, and are wired together using add info about network connectivity here

We first give an analysis of metrics collection in NoisePage. We next evaluate the OLTP performance of our replication architecture described in chapter 4. 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.

REMOVE LATER benchmarks for metrics exporting:

- How much data is outputted with the following heuristics
  - No limiting
  - Hardcoded replication delay
  - Delay exceeds some standard deviation of running mean

benchamrks for ML model

- Use the previous heuristics and evaluate how they affect model performance
    - Make sure to get baseline value

6.1 Replication Architecture

To evaluate our system architecture, we want an OLTP workload. Even though reads are not replicated, a write-only workload is not representative of a real OLTP workload. Towards this goal, we use the TPC-C benchmark, which simulates a delivery system, with orders being placed and received.

To simulate a distributed environment, we execute our benchmarks between two NoisePage instances running on Type 2 machines. We use a simple master-replica architecture where one instance is the master and serves requests, and the other machine is a replica node. We replicate data asynchronously across the two machines. For synchronizing clocks in our machines to get measurements for replication delay, we use the Network Time Protocol (NTP). We do not require extreme clock precision because we are not doing any logic based on the times, we are simply trying to estimate delay.

We now evaluate our replication architecture using microbenchmarks, and define some important test configurations and baselines used moving forward.

6.1.1 Arrival Rate

One important consideration to make with any benchmark is the arrival rate. The arrival rate is defined as the frequency the database is queried across all worker threads. For example, if there are four worker threads, 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 6.1 showcases this effect in NoisePage. We can see how the replica remains in sync wth the master with a reasonable 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 system,

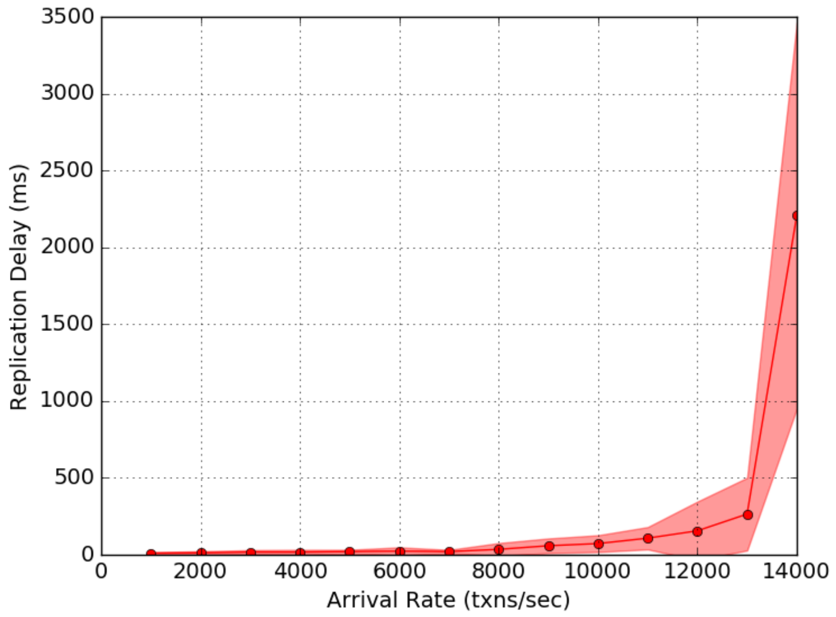


Figure 6.1: Sensitivity of replication delay - Measuring the average replication delay with varying arrival rates in NoisePage over TPC-C with 4 warehouses on Type 2 machines. When the arrival rate exceeds the transaction replaying rate, delay sharply increases. The shaded region denotes one standard deviation from the mean for each data point.

although they may vary in the arrival rate they are able to tolerate. We assume an arrival rate of 10,000 txns/sec for future experiments to get stable delay measurements.

6.1.2 Replication delay over time

As we 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, measuring the replication delay over the span of the benchmark in fig. 6.2. The sharp spikes in replication are as a result of the delivery transaction of TPC-C that takes longer to replay relative to the other transactions. Despite the spikes, we see from the running mean (red line) that the replication delay remains fairly stable throughout the benchmark execution. Over the entire benchmark, the average replication delay is approximately 30ms.

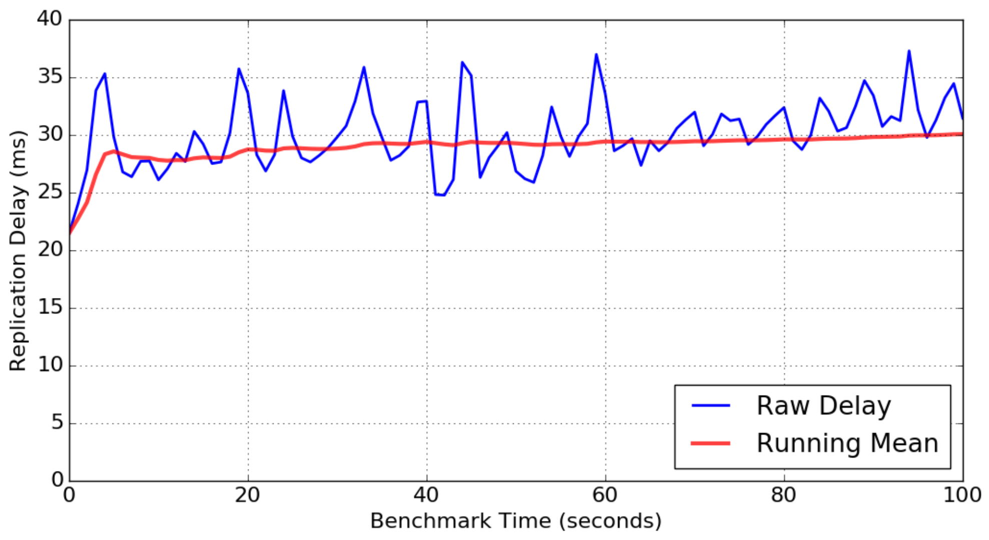


Figure 6.2: Replication Delay in NoisePage - Measuring the average replication delay in NoisePage over TPC-C with 4 warehouses on Type 2 machines. We average the delay for each second, and plot the average over 10 benchmark runs. The red line shows the running mean of the replication delay.

6.2 Metrics Overhead

In section 1.1, we motivated our decision to use database replicas by discussing the metrics collection overhead, and observed it is approximately 11% in PostgreSQL. For our this analysis, we used PostgreSQL instead of NoisePage because NoisePage is still a relatively new, unfinished system. In particular, NoisePage still has an immature metrics collection and does not yet output as many metrics as a system of its size should. Due to this, we decide simulate NoisePage having similar metrics overheads to PostgreSQL.

NoisePage currently has two (out of 10) high level components, the `TransactionManager` and log manager that export metrics. Collectively, these two components export 11 unique metrics (e.g. disk write speed, transaction latch wait time). For reference, we calculated PostgreSQL to output approximatly 300 unique metrics by looking at its internal metadata tables.

The approach we took to simulate a realistic DBMS metrics overhead was to scale up the amount of metrics data exported by each component i.e when a metric is generated, it is exported multiple times. To show the effect of this approach, we execute the TPC-C

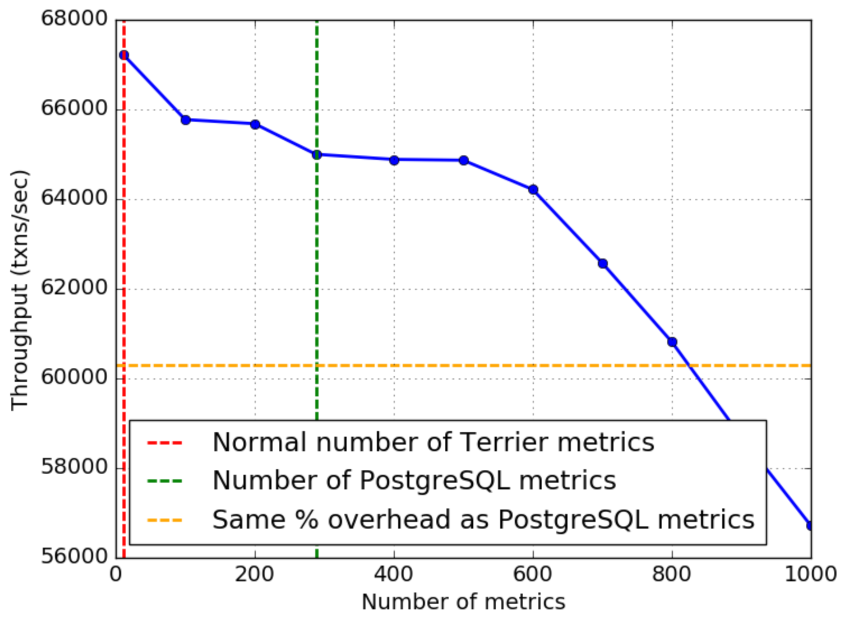


Figure 6.3: Metrics overhead in NoisePage - Overhead of metrics collection as we scale the number of metrics exported

benchmark with 6 warehouses on machine Type 1. We compare the transaction throughput while scaling up the number of metrics exported using the approach described. Figure 6.3 shows the effects of this technique on transactional throughput. With the current metrics in the system (red line), NoisePage executes $\sim 67,000$ txns/sec. If we scale number of metrics to the number of metrics we estimate PostgreSQL exports (green line), we see a throughput of $\sim 65,000$ txns/sec, only a 3% overhead. This does not equate the 11% we expect to see because throughout the lifecycle of a tranaction, different metrics are exported at different frequencies. Therefore, scaling NoisePage's metrics to the same number of metrics that PostgreSQL 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 approximatly 800 metrics. We use this scale of metrics collection for future experiments.

Processing Time (μs)	**Percentage of Samples**
0 - 8	17.6
8 - 12	49.6
12 - 16	17.5
16 - 20	9.9
>20	6.4

Table 6.1: Test set data distribution - Processing time distribution in Log Serializer Task for test set (30,000 total samples)

6.3 Dynamic Metrics Collection

6.4 Self-Driving Models

Describe what model doesn

Describe what it could be used for (spin up more thread if serialization is backed up)

Chapter 7

Conclusions and Future Work

Our goal is to parallelize individual transactions in a database, using FOO.

Bibliography

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

[2] 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`. 4.1

[3] 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

[4] 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

[5] 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

[6] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. Tuning database configuration parameters with ituned. *PVLDB*, 2:1246–1257, 2009. 1, 3

[7] Google. Cloud spanner documentation: Replication. 2.2, 3

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

[9] 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

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

[11] 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

[12] Tianyu Li. Supporting Hybrid Workloads for In-Memory Database Management Systems via a Universal Columnar Storage Format. Master's thesis, May 2019. 4.5

[13] 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, 5.1

[14] 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. 2.2.1

[15] 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

[16] 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

[17] MemSQL. Memsql docs: Using replication. 3

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

[19] Microsoft. Sql server: Transactional replication. . **??**, 2.2, 3

[20] 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`. 4.3, 4.4

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

[22] MongoDB. Mongodb documentation: Replication. . 2.2, 3

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

[24] MySQL. Mysql documentation: Replication. . **??**, **??**, 2.2, 3

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

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

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

[28] 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

[29] PostgreSQL. High availability, load balancing, and replication. **??**, **??**, 2.2, 3

[30] D. P. Reed. Naming and synchronization in a decentralized computer system. Technical report, Cambridge, MA, USA, 1978. 4.1

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

[32] 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`. 3

[33] 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

[34] 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`. 3

[35] 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`. 2.2.1, 3

[36] 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