# Scalable and Highly Available Database Systems in the Cloud

by

Umar Farooq Minhas

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2013

Umar Farooq Minhas

## Abstract

Cloud computing allows users to tap into a massive pool of shared computing resources such as servers, storage, and network. These resources are provided as a service to the users allowing them to “plug into the cloud” similar to a utility grid. The promise of the cloud is to free users from the tedious and often complex task of managing and provisioning computing resources to run applications. At the same time, the cloud brings several additional benefits including: a pay-as-you-go cost model, easier deployment of applications, elastic scalability, high availability, and a more robust and secure infrastructure.

One important class of applications that users are increasingly deploying in the cloud is database management systems. Database management systems differ from other types of applications in that they manage large amounts of state that is frequently updated, and that must be kept consistent at all scales and in the presence of failure. This makes it difficult to provide scalability and high availability for database systems in the cloud. In this thesis, we show how we can exploit cloud technologies and relational database systems to provide a highly available and scalable database service in the cloud.

The first part of the thesis presents RemusDB, a reliable, cost-effective high availability solution that is implemented as a service provided by the virtualization platform. RemusDB can make any database system highly available with little or no code modifications by exploiting the capabilities of virtualization. In the second part of the thesis, we present two systems that aim to provide elastic scalability for database systems in the cloud using two very different approaches. The three systems presented in this thesis bring us closer to the goal of building a scalable and reliable transactional database service in the cloud.

## Acknowledgements

I could not be more thankful to Almighty Allah who has always blessed me with success in life. Next in line are my mother, father, six siblings, and my wife who have always extended their unconditional support, love, and prayers that have helped me throughout my personal and professional life. I would like to thank all my friends in Waterloo for making my stay very special, and my friends in Pakistan and world over for being a special part of my life.

My supervisor, Professor Ashraf Aboulnaga, deserves a great deal of appreciation for his extensive support, both moral and financial, and his extraordinary patience. I would like to thank Professor Kenneth Salem for his contributions to the work presented in Chapter 3 and Chapter 5 of this thesis, and for his excellent advice and continued support throughout my studies at Waterloo. I would further like to thank Professor Kenneth Salem and Professor Tamer Özsu for serving on my committee, and for their critical feedback on an earlier version of this work. I would also like to thank my external examiner, Professor Donald Kossmann, and my internal external examiner, Professor Ladan Tahvildari, for serving on my thesis committee.

I would like to thank Professor Andrew Warfield and my colleagues Shriram Rajagopalan and Brendan Cully, at the University of British Columbia, for their collaboration on the RemusDB work presented in Chapter 3 of this thesis. I would further like to thank my colleagues at the University of Waterloo: Jonathan Ng, Rui Liu, Sean Robertson, and Taha Rafiq for their contributions to this work. I would like to thank Dr. David Lomet and Dr. Chandramohan Thekkath for exceptional mentoring during my internships at Microsoft Research and for their encouragement and support. I would like to thank Professor Robin Cohen, Professor Jie Zhang, and Professor Thomas Tran for providing me with an excellent opportunity to work on trust modeling for vehicular ad-hoc networks (VANETs), considerably expanding my research horizons beyond this thesis. And last but not least, I would like to thank the administrative and technical staff at the University of Waterloo for always helping me and providing me with an excellent environment to conduct top quality research.

## Dedication

I dedicate this work to my beloved parents, siblings, and my loving wife.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Cloud computing allows users to connect to a massive, shared pool of computing resources that are provided as a service to users, allowing them to "plug into the cloud" very similar to a utility grid. Such a computing model allows consolidation on the server side and reduces costs by taking advantage of economies of scale. The promise of the cloud is to free users from the tedious and often complex task of managing and provisioning computing resources to run applications. The cloud also brings several additional benefits including (1) a *pay-as-you-go* cost model, which means that users only pay for what they use, (2) much *easier and faster deployment* of applications, (3) *dynamic and elastic provisioning* of resources, which means that unlike a traditional data center setting, there is no need to provision for peak load since applications can grow and shrink processing capacity depending on the load, and d) a more *robust and secure infrastructure* that is highly available and elastically scalable. All these features of the cloud come to users at no additional cost, while at the same time providing them with a lot of flexibility in using and provisioning computing resources.

An important class of applications that are increasingly deployed in the cloud is database management systems (DBMSes). In order to provide services to users, applications deployed in the cloud need to store and query massive amounts of data, and they need to do so in a scalable, reliable, and efficient manner. Database management systems have been a very popular choice for storing data for many data-intensive applications for over four decades. Relational DBMSes offer users a simple and flexible data model in which data is stored persistently as tables. DBMSes also provide the notion of *transactions*, which greatly simplify the task of maintaining the database in a consistent state in the presence of concurrent users issuing conflicting updates and in the presence of failures. Furthermore, DBMSes allow users to define, manipulate, and query data using an easy-to-use declarative query language known as the *Structured Query Language (SQL)*. Database transactions combined with a SQL interface greatly simplify the implementation of applications on top of DBMSes which is a main reason for the popularity of DBMSes. However, deploying DBMSes in cloud computing environments presents some unique challenges in providing high availability and elastic scalability for these systems, which is the focus of this thesis. In the next sections, we first define what we mean by high availability and elastic scalability. Then we discuss various challenges related to building and deploying a highly available and elastically scalable DBMS in the cloud, and how this thesis addresses some of these challenges.

## 1.1 High Availability

*High availability (HA)* refers to the ability of a system to remain accessible to users in the presence of either software or hardware failures. For example, an application may crash or a network card may go down or an entire physical machine can fail. Various studies have shown that downtime is a “revenue killer”. Small to medium sized businesses lose 1% of revenue per year to system downtime [71], while 40–50% of businesses never fully recover after a major system outage [44]. The requirement to provide high availability is no longer limited to mission critical applications. Even the most simple applications require high availability, meaning that high availability is no longer an option but rather a requirement. Indeed, most businesses now implement some form of high availability for their IT infrastructure. Database systems also need to provide high availability, where the database remains accessible and consistent in the presence of failures, with little or no downtime. Since database systems are increasingly being deployed in the cloud, providing database HA in the cloud has become an important goal. Traditional HA mechanisms can be used in the cloud, but cloud technologies may actually enable simpler HA solutions. We present such a solution in this thesis.

## 1.2 Elastic Scalability

A software system is said to be *scalable* if it is able to handle increasing load simply by using more computing resources. A system can be *scaled up* by adding more computing resources to the physical machine it runs on. For example, by adding more memory, or adding another CPU. *Scale-out* permits a system to handle even larger workloads by adding more physical machines, for example in a cluster. Systems that are *elastically* scalable are able to respond to changes in load by growing and shrinking their processing capacity on the fly. Cloud computing infrastructures favor systems that are able to scale in an elastic fashion. Ideally, at any given time, an application deployed in the cloud should be using exactly the amount of resources required to handle its load, even as this load fluctuates [16]. Such elastic scalability coupled with the pay-as-you-go cost model of the cloud results in significant cost savings for the applications.

## 1.3 Challenges of Deploying Database Systems in the Cloud

Software systems vary in their ability to scale elastically and to provide high availability depending on how they are designed and what kind of work they handle. *Stateless* systems, such as web and application servers, can be designed to be highly available and scalable with little effort. One of the major challenges of deploying database systems in the cloud can be attributed to their *stateful* nature. Database management systems have to maintain the consistency of the database that they store. Therefore, it is very hard to provide high availability and elastic scalability while maintaining consistency of the database. This makes it particularly hard to deploy a database system in the cloud. And because database systems are usually coupled with other systems, e.g., with web and application servers in a three-tier architecture, they limit the scalability and availability of the entire system stack. In short, existing database management systems are not well suited for deployment in the cloud. To address this problem, cloud database deployments can forgo the power of SQL DBMSes and use simpler cloud storage systems, also called NoSQL systems. Another alternative, which is the focus of this thesis, is to design SQL DBMSes that are cloud friendly. We discuss the two alternatives in the next two sections.

## 1.4 Cloud Storage Systems

In order to overcome the challenges faced by traditional DBMSes in the cloud, recently various cloud storage systems [31, 62, 80] (also referred to as NoSQL systems), have been proposed. Such systems provide high availability and elastic scalability, among other things, by limiting certain features of the system. The two most common limitations are: ***(1) Weak or eventual consistency***: As opposed to traditional database systems, cloud storage systems, only guarantee that when a change is made to the database, clients will *eventually* see the effect of this change. Clients should be prepared to deal with inconsistent results. Some systems provide strong consistency, but only for updates on a single row, with no support for multirow transactions. ***(2) NoSQL interface***: Cloud storage system support a very restricted query model that is less expressive than the full SQL model supported by database systems. Typically, these systems only support read and write operations on individual rows. Because of these limitations of NoSQL cloud data storage systems, applications that are built on top of these systems need to be much more complex. A particular disadvantage of such systems is that the task of maintaining consistency of the database is delegated to application developers, and is therefore highly error prone.

## 1.5 Database as a Service

Cloud computing companies like Amazon, Google, and Microsoft offer what is called *Database-as-a-Service (DaaS)* [8, 57, 97]. These services bridge the gap between traditional database systems and cloud storage systems but only to some extent; they too make certain trade-offs. For example, some DaaS systems do not support elastic scalability, while others do not support the full SQL interface. There is a need to develop new techniques that provide high availability and elastic scalability similar to cloud storage systems, with support for strong consistency and the full SQL interface similar to relational database systems. Developing such techniques is the focus of this thesis.

## 1.6 About this Thesis

The goal of this thesis is to exploit our knowledge of database management systems and emerging cloud technologies (e.g., virtualization and distributed cloud storage systems) to improve the deployment and usability of database systems in the cloud. More specifically, we want to use cloud technologies and relational database systems to build a highly available and elastically scalable database service in the cloud while providing strong consistency and supporting a full SQL interface. The techniques presented throughout this thesis are a first step towards building "cloud-friendly" database systems.

To show the practicality of the techniques developed as part of this thesis, we have implemented three systems. The first system, called RemusDB, provides high availability for database systems by exploiting virtualization. The remaining two systems provide elastic scalability for database systems either by using *data sharing* or *data partitioning*. We now present a brief overview of each of these systems:

1. **RemusDB: Transparent High Availability for Database Systems:** Traditionally, expensive hardware coupled with customized software has been used to provide high availability for business critical applications, including database systems. The cost and complexity of deploying such a solution is prohibitive for most small to medium sized businesses. With recent advances in virtualization technologies, it is now possible to provide high availability for the masses on cheap commodity hardware in an application and operating system agnostic manner. In the first part of this thesis, we present a system called RemusDB for building a highly available database management system. The proposed techniques that are part of RemusDB can be applied to any DBMS with little or no customization, and with reasonable performance overhead. RemusDB is based on Remus [37], a commodity HA solution implemented in the virtualization layer that uses asynchronous virtual machine (VM) state replication to provide transparent HA and failover capabilities. We show that while Remus and similar systems can protect a

DBMS, database workloads incur a performance overhead of up to 32% as compared to an unprotected DBMS. We identify the sources of this overhead and develop optimizations as part of RemusDB to mitigate the problems. We present an experimental evaluation using two popular database systems and industry standard benchmarks showing that for the tested workloads, RemusDB provides very fast failover ($\leq$ 3 seconds of downtime) with low performance overhead when compared to an unprotected DBMS. More details of RemusDB can be found in Chapter 3.

2. **Chimera: Elastic Scale-out and Load Balancing Through Data Sharing:** The current parallel database market is split between shared nothing and data sharing systems. While shared nothing systems are easier to build and scale, data sharing systems have advantages in elastic scalability and load balancing. In this thesis, we explore adding data sharing functionality as an extension to a shared nothing database system. Our approach isolates the data sharing functionality from the rest of the system and relies on well-studied, robust techniques to provide the data sharing extension. This reduces the difficulty in building systems that provide data sharing, yet enables much of the flexibility of a data sharing system. We present the design and implementation of Chimera – a hybrid database system, targeted at load balancing for many workloads and scale-out for read-mostly workloads. The results of our experiments demonstrate that we can achieve almost linear scalability and effective load balancing with less than 2% overhead during normal operation. The details of this system can be found in Chapter 4.

3. **Elastic Scale-out for Partition-Based Database Systems:** An important goal for database systems today is to provide elastic scale-out, i.e., the ability to grow and shrink processing capacity on demand as the offered workload varies. Database systems are difficult to scale since they are stateful – they manage a large database. It is important when scaling to multiple server machines to provide mechanisms so that these servers can collaboratively manage the database and maintain its consistency. Database partitioning is often used to solve this problem, with each server being responsible for one or more partitions. In this thesis, we propose that the flexibility provided by a partitioned, shared nothing parallel database system can be exploited to provide elastic scale-out. The idea is to start with a small number of servers that manage all partitions, and to elastically scale out by dynamically adding new servers and redistributing database partitions among these servers. We present an implementation of this approach for elastic scale-out using VoltDB – an in-memory, partitioned, shared nothing parallel database system [152]. Our approach to elastic scale-out gives rise to many interesting, and challenging research problems in database manageability. This thesis focuses on one such problem – partition (data) placement with elastic scale-out. We present more details of our implementation and the partition placement problem in Chapter 5.

## 1.7 Organization

The remainder of this thesis is organized as follows. Chapter 2 provides an overview of cloud computing, virtualization, high availability, and techniques for scaling database systems. In Chapter 3, we present RemusDB, a system that provides high availability by exploiting virtualization. In Chapter 4, we present the details of Chimera, a database system that provides elastic scalability through data sharing. In Chapter 5, we present a system that provides elastic scalability by using data partitioning. Finally, Chapter 6 concludes this thesis.

# Chapter 2

# Background

In this chapter, we present background on cloud computing, virtualization, high availability, and elastic scalability for database systems. Without loss of continuity, more advanced readers can directly skip to Chapter 3.

## 2.1 Cloud Computing

In today's world, computing is central to running and managing a business. Business applications need computing resources, and in order to provide these resources, businesses need to make a significant investment in IT infrastructure. Depending on the size of the business, this infrastructure can range from just a small number of servers to an entire data center. However, provisioning and managing computing resources is costly. There are three major costs: (1) capital costs, i.e., the cost of buying the hardware, software, and installation costs, (2) maintenance and operation costs, i.e., the cost of personnel to maintain the infrastructure as well as cost of power, cooling, and sever storage, and (3) additional costs, for example, to buy extra hardware and software to provide high availability, reliability, scalability, and security. All of these costs add up fairly quickly and can amount to a significant portion of a company's budget. This creates a high barrier to entry for small to medium sized businesses which simply cannot afford spending much on IT infrastructure. There is a need to cut costs in order to improve profitability and lower the barrier to entry.

In addition to economic factors, there are other issues related to provisioning and managing computing resources. For example, in order to provision more capacity, we need to buy new hardware, install the required software stack, and then finally deploy or migrate existing applications to this new hardware. This can be a slow and complex process. As a result, capacity is usually provisioned for peak load, which results in under-utilization of computing resources. Secondly, upgrading software is difficult. In today's computing environments, applications typically run on top of a complex, multi-tiered software stack. If not done carefully, upgrading just one

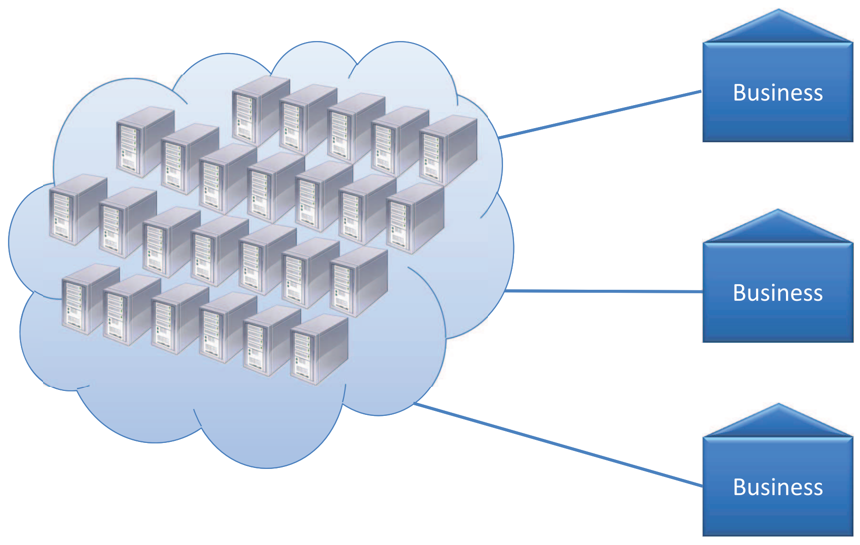


Figure 2.1: Enabling Consolidation with Cloud Computing

piece of software in the stack can potentially bring down the entire system. These problems become even more pronounced as the infrastructure grows and becomes more complex leading to even more manageability problems.

Cloud computing aims to solve the problems outlined above. Cloud computing provides computing resources at a low cost while at the same time freeing users from the tedious and often complex tasks of maintaining and provisioning resources. Instead of investing in an IT infrastructure in-house, businesses can connect to a massive shared pool of computing resources, "the cloud", and can outsource their computing needs to the cloud. As shown in Figure 2.1, these resources can be shared between multiple businesses, resulting in even better resource consolidation and much lower costs.

Different service models used in cloud computing provide different levels of abstraction to its users as shown in Figure 2.2. At the very bottom we have the Infrastructure-as-a-Service (IaaS) model, where hardware resources such as server, storage, and network are made available to the users in a flexible manner. Services like Amazon EC2 [7] and Rackspace Open Cloud [121] are examples of IaaS. In Platform-as-a-Service (PaaS), developers code and deploy application on the cloud using an application programming interface (API) that is given to them by the cloud providers. The specifics of how the applications are managed and provisioned on the platform are automatically managed by the infrastructure. Google App Engine [56] and Windows Azure [154] are examples of PaaS. Software-as-a-Service (SaaS) is at the highest level of abstraction where users are provided with a complete hardware and software stack, with pre-deployed applications to use. Examples of SaaS include Google Docs [58] and salesforce.com [128]. Users of cloud computing can choose a cloud service model depending on the desired level of abstraction that best suits their specific needs and usage scenarios.

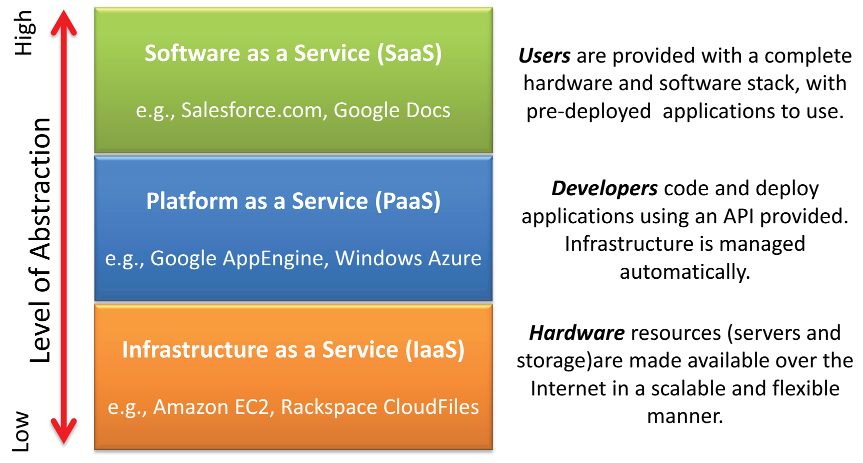

High
Level of Abstraction
Low
Software as a Service (SaaS)
e.g., Salesforce.com, Google Docs
Platform as a Service (PaaS)
e.g., Google AppEngine, Windows Azure
Infrastructure as a Service (IaaS)
e.g., Amazon EC2, Rackspace CloudFiles
Users are provided with a complete hardware and software stack, with pre-deployed applications to use.
Developers code and deploy applications using an API provided. Infrastructure is managed automatically.
Hardware resources (servers and storage)are made available over the Internet in a scalable and flexible manner.

Figure 2.2: Cloud Computing Abstractions

There are many new and emerging technologies which are the building blocks for cloud computing. One such enabling technology for cloud computing is virtualization. In the next section, we provide a brief overview of virtualization.

## 2.2 Virtualization

Machine virtualization is a technique that allows the resources of a physical machine to be shared among multiple partitions known as *virtual machines* (VMs). Each virtual machine runs an independent (possibly different) operating system and the associated set of applications. A layer of indirection known as a *virtual machine monitor* (VMM) is introduced between the physical resources of a machine and the virtual machines running over it. The virtual machine monitor manages the physical resources and provides a mapping between the physical resources and the abstractions known as *virtual devices*. A virtual device corresponding to each physical resource, e.g., CPU, disk, memory, and network, is exported to every virtual machine. Since a virtual machine is nearly an exact replica of the underlying hardware, this makes it possible to run applications unmodified inside a virtual machine. Xen [18] and VMware [150] are popular examples of virtual machine monitors. Figure 2.3 provides an overview of machine virtualization.

The virtual machine monitor provides (1) performance isolation – each virtual machine will use only its allocated share of resources thus will not affect the performance of other virtual machines running on the same physical machine, (2) fault isolation – a software bug or a security flaw in one virtual machine does not affect other virtual machines, (3) dynamic resource allocation – the share of physical resources allocated to each virtual machine can be adjusted on the fly, and (4) live migration – a virtual machine running on one physical host can be migrated to another physical host with minimum downtime.

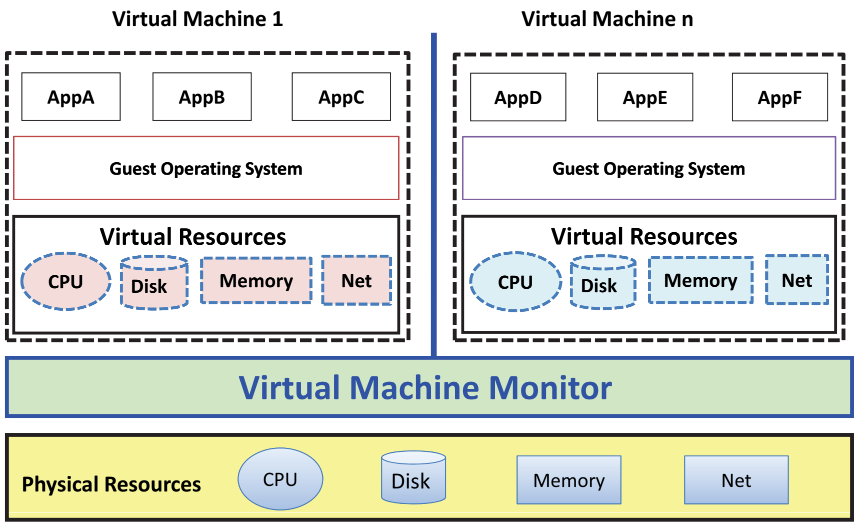


Figure 2.3: Machine Virtualization

Server consolidation has often been cited as one of the most important benefits of virtualization. Server consolidation allows multiple applications, each of which typically runs on a separate physical machine, to be co-located on a single physical machine, with each application running inside a separate virtual machine. For example, in a multi-tiered application, the web server, the application server, and the backend database management system can all be run on the same physical server. This increases the utilization of physical resources while reducing the total cost of ownership by requiring less hardware, less personnel to manage that hardware, and reduced operational costs (e.g., power and cooling costs).

### 2.2.1 Virtualization and Cloud Computing

Virtualization is changing the way we develop, deploy, and use applications. A *virtual appliance* is a novel way of deploying applications, enabled by virtualization. A virtual appliance has pre-installed and pre-configured copies of operating system and application software, ready to go out-of-the-box, which considerably reduces time required to deploy, configure, and tune an application. Owing to these benefits, virtual appliances are a popular way of deploying applications in the cloud.

In IaaS clouds such as Amazon's Elastic Computing Cloud (EC2) [7], the cloud provider manages and operates a pool of possibly tens of thousands of interconnected machines. Users can rent out as much computing power as they need from this pool of resources. In Amazon's EC2, this is made possible by the use of virtualization. Users are provided access to virtual appliances that come in varying sizes in terms of physical resources allocated (e.g., CPU, memory), allowing users to choose a size depending on their resource needs. By exploiting virtualization, cloud providers are able to achieve higher resource consolidation, better resource utilization, and flexibility in provisioning and managing computing resources.

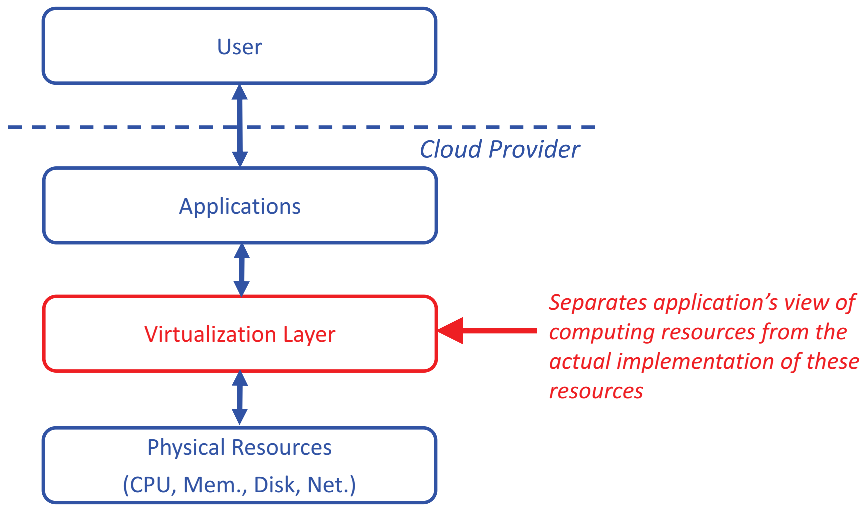

User
Cloud Provider
Applications
Virtualization Layer
Separates application's view of computing resources from the actual implementation of these resources
Physical Resources
(CPU, Mem., Disk, Net.)

Figure 2.4: Virtualization and Cloud Computing

As shown in Figure 2.4, IaaS cloud providers introduce a layer of indirection between the physical resources (e.g., CPU, memory, disk, and network) and the applications that use those resources. By using such an approach, the actual implementation details of the physical resources can be hidden from the applications. As long as the interface exposed to the applications remains the same, a cloud provider can change physical resources, for example, replace an older disk subsystem with a newer, faster system, without affecting the applications. This flexibility provided by virtualization is a key to the success of cloud computing platforms such as Amazon EC2.

### 2.2.2 Deploying Database Systems in the Cloud

Database systems are increasingly being run in virtual machines in cloud computing environments (the virtual appliance model). Optimizing and scaling database systems in such deployments presents unique challenges for database researchers. There has been some recent research to quantify the overhead of running database systems inside a virtual machine [103]. Researchers have also developed techniques for automatically tuning database appliances to meet Service Level Agreements (SLAs) in face of changing workloads by using the dynamic resource allocation capabilities provided by virtual machine monitors [135, 136]. Techniques to optimally provision CPU capacity of a physical machine among multiple database appliances competing for resources have also been proposed [1]. However, providing high availability and elastic scalability for database systems in the cloud are still open research problems. In Chapter 3, we show how to exploit virtualization to implement a highly available database system for cloud computing environments. As background for that work, the next section provides an overview of different techniques used to provide high availability for database systems.

## 2.3 Building Highly Available Database Systems

In this section we start by describing hardware solutions for high availability. We then present techniques that have been used to implement highly available database systems. Finally, we present an overview of some recent solutions that use virtualization to provide high availability at a reduced cost.

### 2.3.1 High Availability Through Hardware

Hardware reliability is an important part of implementing a highly available solution. Today, systems are built in a modular fashion where each hardware module can fail and be repaired or replaced independently of other modules in the system. Furthermore, over the years reliability of individual hardware modules has improved significantly due to improved designs and technological advancements. Hardware solutions for achieving high availability typically involve redundancy of individual modules, or of the entire system (through clustering).

Redundancy in hardware modules is the key in implementing a highly available system. If one module fails, it is readily replaced by another *spare* module while the failed module is repaired or replaced. Individual modules, in turn, may have also been designed with internal redundancy. In this case, the module will fail only if a majority of the internal (redundant) components fail. Techniques for automatically detecting the failure of hardware components and failing over to backup components have also become commonplace. Disk mirroring [115], redundant network connections, or a redundant power supply may protect against disk, network, or power failures, respectively.

By taking hardware redundancy to the level of an entire system, a *cluster* can be formed by combining multiple physical machines or *nodes*. The cluster can have one of the following three configurations:

1. Active/cold configuration where one of the servers is active while the second is a *cold* backup, i.e., not serving workloads or attempting to keep up with the active server.

2. Active/passive configuration where one of the servers is active while the second is running the same workload (with some lag) and thus is a *warm* standby.

3. Active/active configuration where both the servers are serving workloads actively.

In general, the warmer the configuration, the more expensive it would be. In all the above configurations, the secondary server takes over execution from the primary if the primary fails, providing high availability to the users of the system. Also, the primary and secondary servers may be geographically distributed. This provides better isolation from faults that may affect one site but not both sites at the same time.

Use of hardware solutions to provide high availability has some drawbacks. First, it requires redundant hardware such as disks for mirroring and replication, Storage Area Networks (SAN), Network Attached Storage (NAS), or multi-path storage devices for shared storage, and bonded network interfaces for added reliability. All of these components add to the cost of deploying a high availability solution. Second, for clustering solutions, we cannot just take any application and run it on a cluster and expect it to be highly available. Applications need to be made *cluster-aware* to exploit high availability features. For example, a clustered database system is different from a stand-alone database system owing to the differences in concurrency control, recovery, and cache management. This adds to the cost and complexity of building and deploying cluster-aware applications. And last but not least, the configuration and operation cost of a cluster is much higher than a single system, which can be prohibitive for small or medium sized businesses.

### 2.3.2 High Availability for Database Systems

Hardware solutions for high availability only provide limited advantage when faced with faults in software applications. Applications, such as database systems, couple hardware solutions with software techniques to implement a highly available system. We discuss two common approaches for providing high availability for database systems.

#### Database Replication

Replication is a technique where a database is fully or partially replicated locally or at a remote site to improve performance and data availability [66]. The original copy of the data is referred to as the primary (or master) replica while the replicated copy is called a secondary replica. If the primary replica becomes unavailable, the data still remains accessible through the secondary replica. We can also have $N$-way replication where a primary replica is copied to $N$-$1$ other replicas. However, this additional replication comes at a higher cost. Replication may be implemented within a single database system or between database systems across machines possibly distributed across geographical boundaries. Changes from the primary replica are periodically propagated to the secondary replica. *Synchronous* or *asynchronous* replication are the two options for keeping the secondary replica up-to-date with the primary replica.

Synchronous replication [20, 60] is usually implemented through a *read any, write all* mechanism. Where a read transaction can read data from any replica but a write transaction must update all replicas. For this type of replication, the updating transaction must acquire exclusive access to all the replicas which might involve lock requests across remote sites. Also, in order for an update transaction to successfully commit, all the replicas must remain accessible during the transaction. If the replicas are distributed among remote and local sites, a *two-phase commit* protocol is required to ensure that each transaction either commits at all replicas or is aborted. Synchronous replication offers the advantage of keeping all the replicas strongly consistent. However, due to the high operational costs associated with acquiring locks, and excessive message passing for two-phase commit, it is rarely used in practice [60].

On the other hand, asynchronous replication [14, 23, 33, 60, 112] propagates the changes from the primary to the secondary through transaction log shipping or snapshots. The changes in the log records or a snapshot are then applied to the secondary copy. Asynchronous replication offers a trade-off in terms of minimizing overhead of normal operation and data consistency. With asynchronous replication, it is possible for a transaction to get slightly different results when accessing different replicas because they are updated only periodically. In a typical setting, asynchronous replication is run with two servers, a primary or a master, and a secondary, with the secondary server's database state being transactionally consistent to that of primary server with some *replication lag*. Also, typically, only the primary server can update the replicated data (e.g., in primary site asynchronous replication). When the primary server fails, the secondary server takes over execution losing some work, for example, in-flight transactions are aborted and restarted on the secondary server. Also, the secondary server needs to be brought up-to-date after a failure, i.e., the transaction log of committed operations performed on the primary server that were not yet propagated to the secondary server needs to be replayed at the secondary. After a failure, the primary server can be recovered while the backup server acts as the primary. Later, the roles of the two serves can be switched again, or the backup server can remain as the primary, and vice versa.

By using replication, data remains accessible even when one or more replicas become unavailable. The effect of communication link failures can be minimized and performance can be improved by maintaining a local replica of data stored at a remote site. Also, since the data can be geographically distributed, individual sites are better shielded from a disaster, for example, fire in the data-center or a power outage. Replication is widely supported by all major database vendors [94, 106, 110] and is a popular tool for improving data accessibility and availability.

### Parallel Database Systems

Parallel database systems [25, 94, 106, 107, 110], typically running on clusters of physical machines, are a popular choice for implementing high availability for database systems. A parallel (or clustered) database system executes database operations (e.g., queries) in parallel across a collection of physical machines or nodes. Each node runs a copy of the database management system. Data is either partitioned among the nodes where each partition is owned by a particular node or is shared by all nodes. Failure of individual nodes in the cluster can be tolerated and the data can remain accessible. Such high availability and fault-tolerance is a

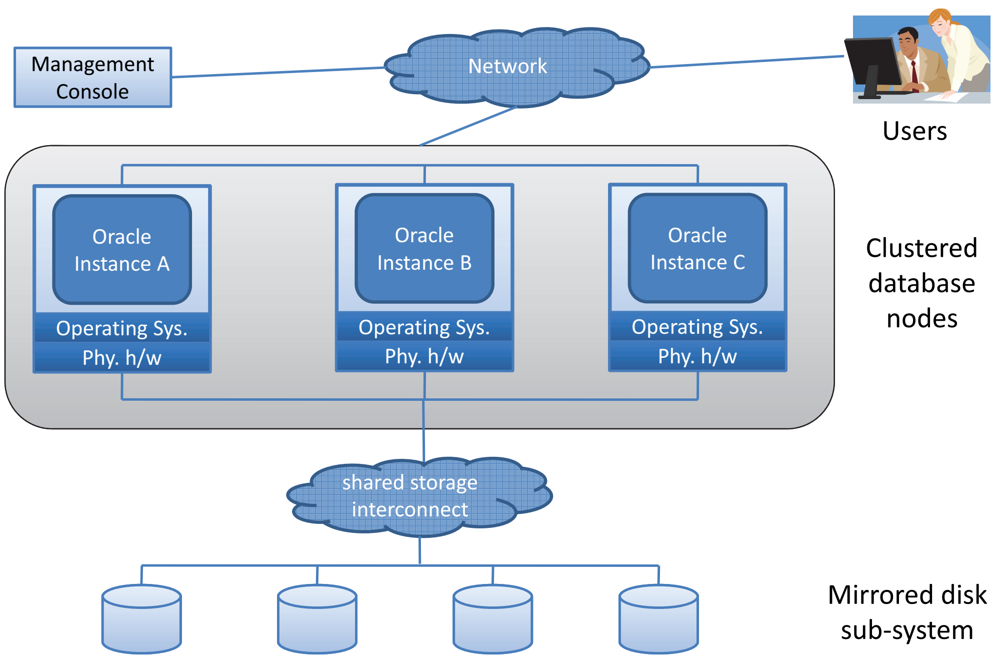


Figure 2.5: Oracle RAC System Architecture

primary reason for deploying a parallel database system, in addition to improved performance. Parallel database architectures in use today are mainly divided into *shared nothing* and *data sharing*.

In a shared nothing system, data is partitioned among nodes where each node hosts one or more partitions on its locally attached storage. Access to these partitions is restricted to only the node that hosts the data and so only the owner of a data partition can cache it in memory. This results in a simple implementation because data ownership is clearly defined and there is no need for access coordination. Shared nothing database systems have been a huge commercial success because they are easier to build and are highly scalable [139]. Examples of shared-nothing database systems include IBM DB2 [94], Microsoft SQL Server [106], and MySQL Cluster [107].

Nodes in a data sharing system share access to all the data. Data is usually hosted on shared storage (e.g., by using Storage Area Networks). Since any node in the cluster can access and cache any piece of data in its memory, access coordination is needed in order to synchronize access to common data for updates. This requires distributed locking, cache coherence, and recovery protocols which add to the complexity of a data sharing system. However, data sharing systems do not require a database to be partitioned and thus have more flexibility in doing load balancing. Examples of data sharing systems include Oracle Real Application Clusters (RAC) [110] and mainframe IBM DB2 [25]. We present the overall architecture of an Oracle RAC system in Figure 2.5 as an example of a parallel database system that is also used to provide high availability.

Figure 2.5 shows a cluster of three physical machines each running an instance of the Oracle DBMS. Each physical machine is connected through a high-speed interconnect to other machines and to the shared storage. Users and administrators connect to the cluster using a high speed network. In order to provide reliability against network failures, multiple network links are used. In order to provide fault tolerance for individual disk crashes, a mirrored disk sub-system is used. Physical host failures can be tolerated. As long as at least one of the hosts remains active, users will get some service. In the rare event of all hosts failing simultaneously, the database is left in a crash-consistent state on the disk sub-system which can be used to recover the database using standard crash recovery procedures [20].

Database replication and parallel database systems provide high availability at the cost of additional hardware and extra software complexity. An interesting proposition is to use virtualization for providing cost-effective high availability for businesses with modest budgets.

### 2.3.3 High Availability Through Virtualization

Exploiting virtualization to provide HA enables us to decrease the complexity of the system because virtual machines are easier to configure and manage. At the same time, using virtualization for HA lowers the cost of HA because less hardware is required. A key technology that comes with virtualization is the ability to migrate virtual machines from one physical host to another while they are running. This capability is known as *live migration* [34, 151]. Live migration has been successfully exploited to minimize planned downtime, and it can be extended to provide a solution that prevents unplanned downtime. Examples of commercial and research prototypes that use virtualization to provide high availability for applications running inside a VM include the following:

1. **VMware HA [149]:** VMware HA is a commercial offering that is part of VMware Infrastructure 3 and provides automatic failover and restart for applications running inside a virtual machine.

2. **Remus [37]:** Remus is a research prototype that provides whole virtual machine replication with the open source Xen in an application and operating system agnostic manner, running over commodity hardware. No external state is ever lost, and the virtual machine is not restarted after a failure. Instead, it continues execution from where it left off before failover.

3. **everRun VM [50]:** This technology provides HA for Xen Server virtualization. It provides automated fault detection and failover for individual VMs. Furthermore, it offers various levels of availability that can be selected on a per VM basis.

4. **Microsoft Hyper-V [96]:** Hyper-V by Microsoft can use Window Server 2008 failover clustering to provide HA for VMs.

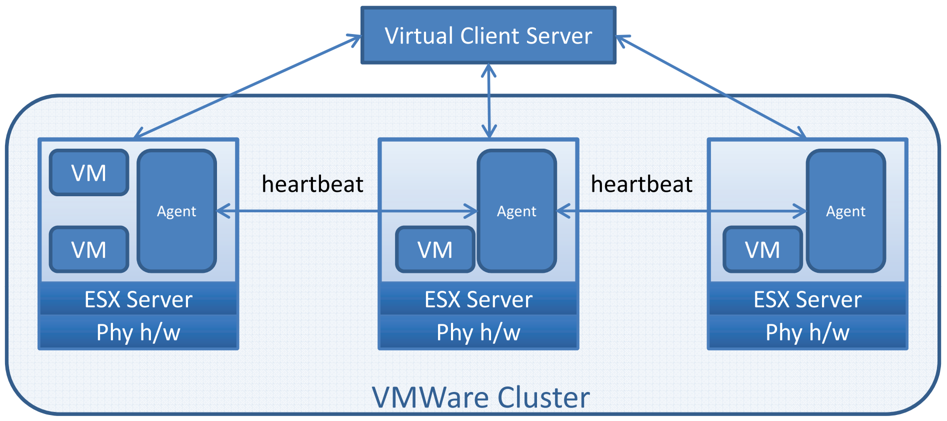


Figure 2.6: VMware HA System Architecture

We focus on VMWare HA and Remus as examples of two systems that provide high availability through virtualization.

### VMware HA

VMware HA is a part of VMware Infrastructure 3 which is a virtualization suite that offers various features to efficiently manage IT infrastructure. It comprises VMware's live migration technology known as VMotion, Distributed Resource Scheduling (VMware DRS) that allows for automated resource optimization, High Availability (VMware HA), and VMware Consolidated Backup – a centralized backup facility. VMware ESX Server provides server virtualization for this infrastructure. VMware ESX Server provides bare-metal virtualization capability, i.e., it runs directly on the physical hardware, without an interposing operating system, and allows virtual machines to run over it. This form of virtualization has some added flexibility and performance benefits as compared to other types of virtualization (e.g., desktop virtualization provided by VMware Workstation).

We present the overall system architecture for VMware HA in Figure 2.6. VMware HA protects applications that are running inside a VM. It provides automatic failover and restart capabilities in case of physical machine failures. Resources of multiple physical hosts are pooled together through what is called a *VMware Cluster*. VMs can then be allocated resources from this pool on demand. VMware HA can provide protection independent of the operating system and the applications running inside the VM. VMware HA works in the following fashion: a central *virtual client server* monitors all ESX servers running in the cluster. *Heartbeat* messages are exchanged between individual hosts and the centralized server. Loss of heartbeat is treated as a host failure and triggers the VM recovery process. Protected VMs from failed hosts are restarted on other surviving nodes in the cluster with minimum downtime.

In order to allow for VM failovers, VMware cluster needs to be over-provisioned. The exact capacity by which the cluster needs to be over-provisioned is determined

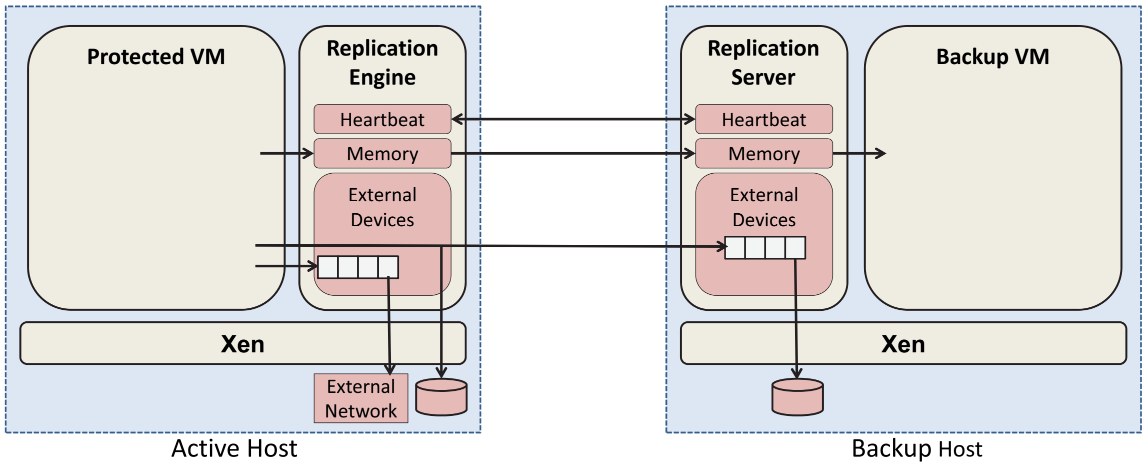


Figure 2.7: Remus System Architecture

by the number of host failures a user wants to protect against, the number of VMs hosted in the cluster, and the resource requirements of each VM. In case there is not enough capacity to failover all VMs, failover priorities dictate which VMs failover first. One thing to note here is that VMware Infrastructure depends on other services such as VMotion and DRS, which in turn depend on shared storage, usually implemented by Storage Area Networks (SANs). These other services and the hardware needed for them are expensive parts of the whole HA solution. Also, on failover a VM is not left running, but instead it has to be restarted from the disk image.

More recently, VMware vSphere Fault Tolerance (FT) [130] implements mechanisms to provide high availability for virtual machines by using *VWware deterministic replay*. In this approach, the two VMs are kept in synch by capturing the execution state of the primary VM and replaying it in exactly that same order at the backup VM. Such an approach requires low network bandwidth, imposes low performance overhead (10%), while allowing transparent failover and restart capabilities. However, this approach requires complex mechanisms to capture the state of the primary VM in a deterministic fashion. Furthermore, it requires shared storage, and does not work for multi-processor VMs.

### Remus

Remus [37] is a research prototype that provides high availability using Xen virtualization in an operating system and application agnostic fashion. We build upon Remus in Chapter 3 and discuss our solution in detail there. Here, we present an overview of Remus as background material.

Remus maintains a replica of an entire VM on a secondary physical machine by extending Xen's live-migration to implement *live checkpointing*, thus providing efficient failover. High performance is achieved by running the primary VM slightly

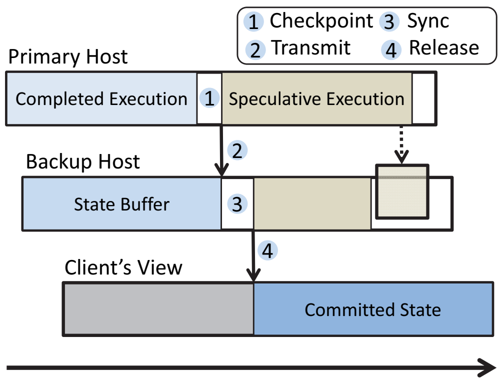


Figure 2.8: Remus Checkpointing

ahead of the back-up using *speculative execution*. One key feature of Remus is that once protection is enabled, no external state (e.g., network connections) is ever lost. For a database system, this means that clients connected to the database will retain their connections during/after failover. This is a key advantage of using Remus. Other solutions [50, 96, 149] only provide VM restart capabilities on a failover.

Figure 2.7 presents a high-level architecture of Remus. Remus utilizes two physical servers paired in an active/passive configuration. The application to be protected is encapsulated in a VM running on the active host. The virtual machine monitor is the open-source Xen hypervisor [18]. The *replication engine* copies the protected VM's state, such as CPU, memory, network, and disk, periodically to the backup server maintaining a near exact replica on the backup VM. This entire VM replication (known as *checkpointing*) can occur as frequently as 40 times per second, every 25ms. A heartbeat is maintained between the active and the backup host. Loss of heartbeat initiates the failover process. It is important to note that the backup VM is not running the full workload of the primary VM on the backup host until a failover happens. This means that only a small fraction of the backup host's resources are used. This allows for a many-to-one relationship between the active and the backup host.

Remus aims to provide protection against fail-stop failure of any single host. In case both hosts fail, the protected VM's data is left in crash-consistent state on persistent storage. And finally, no output is made externally visible until the corresponding system state has been committed to the backup VM. Remus does not aim to provide tolerance against non fail-stop failures, if some errors are caused due to software errors they will be propagated to the backup VM. Note that Remus provides a higher degree of protection when compared with other commercial products. Commercial products usually deal with host failures by simply restarting the protected VM on another host from its crash consistent state. On the other hand, Remus leaves the VM running while keeping all the network connections intact.

Remus uses an epoch-based system for checkpointing. Figure 2.8 presents the state of the active host, the backup host, and a client's view of execution. A checkpoint consists of four steps:

1. Once per epoch, the protected VM is paused and the changed state is copied into a buffer. The VM is then unpaused and continues execution on the active host.

2. The state copied in the buffer is transmitted asynchronously to the backup host.

3. The backup host acknowledges the checkpoint to the active host after the complete state has been received and committed to the backup VM.

4. Finally, the buffered network output is released to the client.

There is considerable overlap between normal execution of the protected VM and the replication process. This is the key to high performance of whole VM replication with Remus.

In order to replicate memory, Remus makes use of Xen's *shadow page tables* [18]. The Xen hypervisor maintains a replica of the page table for every VM and this replica is known as a *shadow page table*. By making all VM memory pages read-only, the Xen hypervisor can trap all updates to the pages that belong to the VM and can build a map of dirty pages which is then used to replicate only changed pages to the backup host. During each epoch, the protected VM is paused, the changed memory and CPU states are copied to a buffer and then the VM resumes execution. This entire process has been optimized for Remus in various ways by extending Xen's live migration mechanism. More details can be found in [37].

For network protection, it is very important that packets queued for transmission in each epoch are buffered and not released to the client until the corresponding checkpoint is acknowledged by the backup host. This is implemented using queuing mechanisms available in the Linux kernel.

For disk protection, Remus maintains a complete replica of the protected VM's disk image on the backup host. Before enabling protection, the initial disk state is copied to the backup. Each write to the primary disk image is then treated as "write-through", i.e., a write is issued to the primary disk at the same time it is sent to a buffer on the backup host. Writes remain in a main memory buffer on the backup host and are committed to disk only when the associated checkpoint is acknowledged.

### 2.3.4 Our Approach to High Availability for Database Systems

Figure 2.9 shows a high level overview of our approach to high availability for database systems. The basic idea in our approach is that a database system which is to be made highly available is encapsulated inside a virtual machine, and then that virtual machine is made highly available by the virtualization infrastructure.

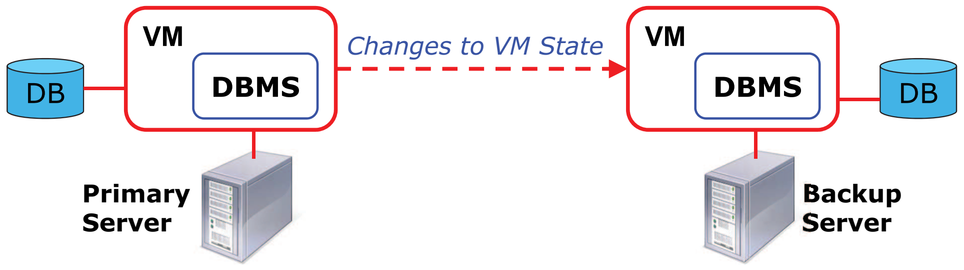


Figure 2.9: Transparent HA for DBMSes

The changes in the state of the virtual machine (including database updates) are propagated from a primary server to a backup server by the virtualization layer. By using such an approach, we can provide high availability for any database system without requiring any code changes, while providing very low performance overhead. In Chapter 3 of this thesis, we present a new system called RemusDB, based on Remus [37], that has been specifically designed to provide high availability for database systems through virtualization.

## 2.4 Scaling Database Systems

The scalability of a system is largely dependent on its design and use cases. Systems that do not have a lot of internal state can be designed to be highly scalable with little effort. On the other hand, systems that carry a lot of internal state are difficult to scale beyond a single node. Examples of *stateless* systems include web and application servers. On the other hand, a database management system is an example of a *stateful* system, the database that it manages being its internal state. This makes it particularly difficult to scale a database system. In this section, we provide an overview of the main approaches used to scale database systems, namely (1) replication, (2) data partitioning, (3) caching, and (4) data sharing.

### 2.4.1 Replication

As described in Section 2.3.2, replication [11, 14, 74, 153] is a fairly easy way to improve scalability of a database system for certain *read-mostly* workloads [10, 12]. Synchronous (or eager) replication [20, 60] keeps all replicas consistent at all times at the cost of reduced performance. Asynchronous (or lazy) replication [14, 23, 33, 60, 112] somewhat ameliorates this overhead, but in that case replicas are only weakly consistent. Also, write-heavy workloads gain limited performance and scalability benefits with the use of replication.

There has been some research to combine strong consistency of eager replication with high scalability of lazy replication. As opposed to the popular belief that synchronous replication in not practical owing to the high overhead [60], researchers have shown that it can be implemented efficiently in clusters of computers and database systems can be made to scale to a large number of nodes [74, 75, 153]. These techniques propose optimizations to reduce the message and synchronization overheard, typical of an eager replication approach. Writes are propagated to all replicas in an eager way, using group communication primitives [65, 76] that ensure global serializablity.

Replication is widely used to scale backend database servers in a three-tier architecture used for dynamic content web sites [11, 14, 74, 153]. Techniques based on content-aware scheduling [10, 11, 30, 119, 125] of database queries for dynamic sites build on earlier techniques [74, 153] to provide strong consistency with high scalability. A total ordering is used for updates which are applied to all replicas asynchronously in that same order. Maintaining information about the level of consistency of each database replica helps schedule read queries to a fully consistent replica, thus hiding any inconsistencies from the users. Such techniques typically require an advance knowledge of all the tables accessed in a transaction and whether it is a read or a write transaction for detecting conflicts [10].

Researchers have proposed various techniques to scale database systems using dynamic replication on clusters formed out of commodity hardware [12, 119, 137]. Amza et al. evaluate various query scheduling and load balancing strategies for a cluster of replicated backend database servers [12]. Conflict-aware scheduling strategies [10] are shown to have a greater positive impact on performance and scalability than other optimizations. Plattner et al. [119] propose the use of a transaction scheduler to separate update queries from read queries and route them to different nodes in the cluster. In their system, all transactions are guaranteed to see a consistent database state, while providing better scalability as compared to eager replication techniques. One advantage of their approach is that it does not need to extract query access patterns as in [12]. Soundararajan et al. [137] propose a technique to scale database backends by using dynamic replications on a cluster. Multiple applications can share physical machines in the cluster. Replicas are allocated on a per application basis to meet performance goals. Additional replicas can be dynamically allocated/deallocated based on the workload. They use *partial overlap* of replicas between different applications to reduce replica addition delay. A *delay-aware* allocation policy is used to avoid fluctuations due to replica allocations. Like other replication approaches, their approach is optimized for read-heavy workloads.

Some previous work is specifically targeted at implementing a solution at the middleware layer for database replication in a cluster. A few approaches [12, 29, 98, 125] focus on techniques for query routing and load balancing. While others [30, 114, 116] are based on eager replication protocols coupled with group communication. These approaches typically assume that there is a 1-1 relationship between a middleware component (e.g., a scheduler) and a database replica. Yet others [11, 119] use a single middleware component, possibly with a backup, with either eager or lazy replication.

Maintaining consistency of data replicated to many nodes has a high overhead which in many cases limits the scalability of the database system. To work around this problem, most solutions based on replication relax consistency requirements to improve scalability, as discussed above. In this work, we propose the use of data partitioning (discussed in the next section) or data sharing to scale database systems while providing strong consistency.

### 2.4.2 Data Partitioning

An alternate technique for scaling database systems is to partition the data and spread it across the nodes of a cluster [36, 93]. A database is usually divided into mutually exclusive partitions and distributed among the nodes of the cluster, where each node is responsible for only its own partition. This technique is commonly used in shared nothing parallel database systems [94, 106, 107, 141]. Database systems that use partitioning can exploit *partitioned parallelism* because now database operations can be performed in parallel on each partition (or node) [46]. Since each node exclusively accesses its partition, inter-node coordination is not required for transaction and cache management. This allows for a simple implementation that is ideal for scale-out [139]. Also, partitioning does not have consistency issues as in the case of replication because typically there is just a single copy of the database stored in a distributed fashion. Therefore partitioning based strategies can potentially provide better scalability.

There are two common methods for defining partitions. In *vertical partitioning*, a table with many columns can be split into multiple partitions where each partition consists of a subset of the columns of the original table. Such a partitioning strategy is beneficial if some columns of a table are more frequently accessed than others. In *horizontal partitioning*, each partition consists of a subset of rows from the original table. For example, rows where the value of the *partitioning key* is between 1 and 1000 might be stored at one node, while rows with the key value from 1001 to 2000 might be stored on another node, and so on. Database systems today offer different ways of selecting rows for each partition. As the example above shows, in *range based partitioning*, a range of value of a certain column defines each partition. Another example is splitting a table containing information about customers on zip codes, where rows belonging to a certain zip code range (e.g., for a county or a state) are always stored on a particular node. One problem with this approach is data skew, i.e., if the values of zip codes are not uniformly distributed in the customers table, some nodes will be more heavily loaded than others. In *hash based partitioning*, a hash function is applied on a partitioning key (e.g., `customer_id`), and the output of this function determines which node will store the corresponding row. If the hash function is chosen carefully, hash based partitioning can avoid the data skew problem. A less commonly used way of partitioning is *list based partitioning* where a list of values is associated with each partition. If the partitioning key of a row is equal to any value in the list belonging to a particular partition, that partition is chosen for this row.

As described above, designing a partitioning strategy for a database involves choosing a *partitioning key* that is used to distribute data over a set of nodes, e.g., by applying a hash function on the partitioning key. Good partitioning allows the majority of data accesses to be executed locally to each node, i.e., it minimizes the need to access data from other nodes. Despite some recent efforts to automate the process of partitioning [2, 113, 158], choosing a good partitioning still remains a complex process. Furthermore, repartitioning has a high overhead due to the cost of physically moving data around in the cluster and thus cannot be done to accommodate short-term load fluctuations. Consequently, nodes have to be over-provisioned, wasting valuable computing and storage resources.

There has been little research done on using dynamic partitioning for database scale-out. As noted above, one of the difficulties with dynamic partitioning is the high cost associated with repartitioning which requires physical movement of the data. This can have undesirable effects such as disruption to ongoing transaction processing [137, 148]. Copeland et al. [36] propose a technique for repartitioning of database tables for load balancing but their method does not have any provisions for dynamic partitioning or modeling the costs of repartitioning. Brunstrom et al. [26] propose simple data repartitioning strategies for changing workloads in a non-replicated partitioned distributed database system. Their goal is to maximize access to local data for each node. Their strategy keeps a count of database blocks accessed by different nodes and tries to repartition to maximize local accesses at each node. The proposed technique was tested only on a two node system, assumes that the workload does not change too rapidly, and does not model the cost of moving partitions from one node to another. Finally, Rivera-Vega et al. [124] propose scheduling strategies to minimize the time it takes to transfer database files for repartitioning and to minimize the impact of repartitioning on transactions.

### 2.4.3 Caching

Caching [6, 82, 95] is often used in combination with other optimizations (e.g., replication [12]) for database scalability. Many database scaling techniques used in practice involve a combination of replication, partitioning, and caching strategies all combined in one solution [133]. Caching can be treated as a form of data replication, though in most cases the granularity of caching is finer than the granularity of replication. Some approaches [9, 41] allow caching query results and check if incoming queries can be satisfied from the cache. Other approaches [6, 82] use a full-fledged database system that *fully* or *partially* caches a backend database or table. In most cases, caching is fully transparent to the applications. Mechanisms for *cache invalidation* are used to maintain consistency between a cached copy of the data and the corresponding copy in the backend database. Typically, such mechanisms simply discard outdated copies of the data from the cache, and repopulate it with the latest copy from the backend database. All updates still go to the backend database. Most of the approaches proposed in the literature use caching in the middle tier of a three-tier architecture to boost performance and scalability of a backend database for web applications [6, 9, 19, 43, 82, 91], while some other approaches provide caching in multiple tiers [118]. Effectiveness of caching largely depends on the objects being cached, the location of the cache, and most importantly on the frequency of updates in the backend database that affect the cached objects. Therefore, as with replication, read-mostly workloads greatly benefit from caching while write-mostly workloads have limited or no benefit.

### 2.4.4 Data Sharing

Data sharing is another technique for scaling database systems. As described in Section 2.3.2, ownership of data is less clearly defined in data sharing systems as compared to shared nothing systems that use data partitioning. Data sharing systems require complex concurrency and coherency control protocols, a global locking scheme, and a distributed recovery mechanism. The added complexity of implementing data sharing systems brings some advantages when compared to much simpler shared nothing systems. One of the key advantage of a data sharing system is their ability to provide elastic scale-out and load balancing at a much finer time granularity. In order to scale a data sharing system, one can readily bring up other database server nodes that are provided access to the database by means of shared storage. As opposed to partitioned database systems, data sharing systems do no require any data movement for load balancing or elastic scale-out. Examples of database systems that achieve scalability by using data sharing include mainframe IBM DB2 [25] and Oracle Real Application Clusters (RAC) [110].

### 2.4.5 Our Approach to Elastic Scale-out for Database Systems

In a recent study [79], both data sharing and partition-based database architectures have been shown to scale really well in cloud computing environments. In this thesis, we present two systems that use either data sharing or partitioning to achieve elastic scale-out and load balancing for database systems.

As noted above, data sharing systems are complex systems that require very careful design and implementation to achieve good performance. In this thesis, we explore providing data sharing as an extension to an existing shared nothing database system. We start with a stand-alone database system and extend it to provide some of the capabilities of a data sharing system, e.g., the ability to do load balancing and elastic scale-out in response to short term load fluctuations. The details of this system, which we call Chimera, are provided in Chapter 4.

An alternative approach to scaling database systems involves using data partitioning. Current data partitioning schemes are not suitable for dynamic scale-out because of disruptive effects of data movement on transaction processing, making it particularly hard to use partitioning for elastic scale-out. Our basic approach for elastic scale-out for partition based database systems involves coming up with new, efficient techniques for moving data partitions to offload some of the work to less loaded nodes in a cluster setting, e.g., in a cloud. The number of nodes serving a workload can be expanded or contracted based on the workload, i.e., if the load is high, a new node can be dynamically introduced in the cluster, and some existing partitions can be moved from existing nodes to this new node. The new node is then assigned the task of serving queries against the partitions allocated to it. Similarly, when the load subsides, partitions can be moved to a smaller set of nodes and free nodes can be deallocated. We present the details of an elastically scalable database system built using data partitioning in Chapter 5.

# Chapter 3

# RemusDB: Transparent High Availability for Database Systems

We now present RemusDB, the first technical contribution of the thesis. Maintaining availability in the face of hardware failures is an important goal for any database management system (DBMS). Users have come to expect 24×7 availability even for simple non-critical applications, and businesses can suffer costly and embarrassing disruptions when hardware fails. Many database systems are designed to continue serving user requests with little or no disruption even when hardware fails. However, this *high availability (HA)* comes at a high cost in terms of complex code in the DBMS, complex setup for the database administrator, and sometimes extra specialized hardware. In this research, we present a reliable, cost-effective HA solution that is transparent to the DBMS, runs on commodity hardware, and incurs a low performance overhead. A key feature of our solution is that it is based on *virtual machine (VM) replication* and leverages the capabilities of the underlying virtualization layer.

Providing HA guarantees as part of the DBMS can add a substantial amount of complexity to the DBMS implementation. For example, to integrate a simple *active-standby* approach, the DBMS has to support propagating database updates from the active to the standby (e.g., by shipping log records), coordinating transaction commits and aborts between the active and standby, and ensuring consistent atomic handover from active to standby after a failure.

We present an active-standby HA solution that is based on running the DBMS in a virtual machine and pushing much of the complexity associated with HA out of the DBMS, relying instead on the capabilities of the *virtualization layer*. The virtualization layer captures changes in the state of the whole VM at the active host (including the DBMS) and propagates them to the standby host, where they are applied to a backup VM. The virtualization layer also detects failure and manages the *failover* from the active host to the standby, transparent to the DBMS. During failover, all transactional (ACID) properties are maintained and client connections are preserved, making the failure completely transparent to the DBMS clients.

As we already mentioned in Chapter 2, database systems are increasingly being run in virtual machines for easy deployment (e.g., in cloud computing environments [1]), flexible resource provisioning [136], better utilization of server resources, and simpler administration. A DBMS running in a VM can take advantage of different services and capabilities provided by the virtualization infrastructure such as live migration, elastic scale-out, and better sharing of physical resources. These services and capabilities expand the set of features that a DBMS can offer to its users while at the same time simplifying the implementation of these features. Our view in this research is that adding HA to the set of services provided by the virtualization infrastructure continues down this road: any DBMS running on a virtualized infrastructure can use our solution to offer HA to its users with little or no changes to the DBMS code for either the client or the server. Our design decisions ensure that the setup effort and performance overhead for this HA is minimal.

The idea of providing HA by replicating machine state at the virtualization layer is not new [24], and we presented a few examples of such systems in Chapter 2. Our system is based on *Remus* [37], a VM checkpointing system that is already part of the Xen hypervisor [18]. Remus targets commodity HA installations and transparently provides strong availability guarantees and seamless failure recovery (details in Chapter 2). However, the general VM replication used by systems such as Remus imposes a significant performance overhead on database systems. In this research, we develop ways to reduce this overhead and implement them in a *DBMS-aware VM checkpointing* system that we call *RemusDB* [101, 102].

We identify two causes for the performance overhead experienced by a database system under Remus and similar VM checkpointing systems. First, database systems use memory intensively, so the amount of state that needs to be transferred from the primary VM to the backup VM during a checkpoint is large. Second, database workloads can be sensitive to network latency, and the mechanisms used to ensure that client-server communication can survive a failure add latency to the communication path. RemusDB implements techniques that are completely transparent to the DBMS to reduce the amount of state transferred during a checkpoint (Section 3.4). To reduce the latency added to the client-server communication path, RemusDB provides facilities that are not transparent to the DBMS, but rather require minor modifications to the DBMS code (Section 3.5). We also describe how RemusDB reprotects a VM after failure by synchronizing the primary VM with the backup VM after the primary VM comes back online (Section 3.6). We use RemusDB to add high availability to Postgres and MySQL, and we experimentally demonstrate that it effectively recovers from failures and imposes low overhead on normal operation (Section 3.7). For example, as compared to Remus, RemusDB achieves a performance improvement of 29% and 30% for TPC-C workload running under Postgres and MySQL, respectively. It is also able to recover from a failure in $\leq 3$ seconds while incurring only 3% performance overhead with respect to an unprotected VM. After establishing the effectiveness of RemusDB for HA, we turn our attention to administrative issues related to RemusDB. We present an approach for modeling the network bandwidth required by RemusDB and the performance

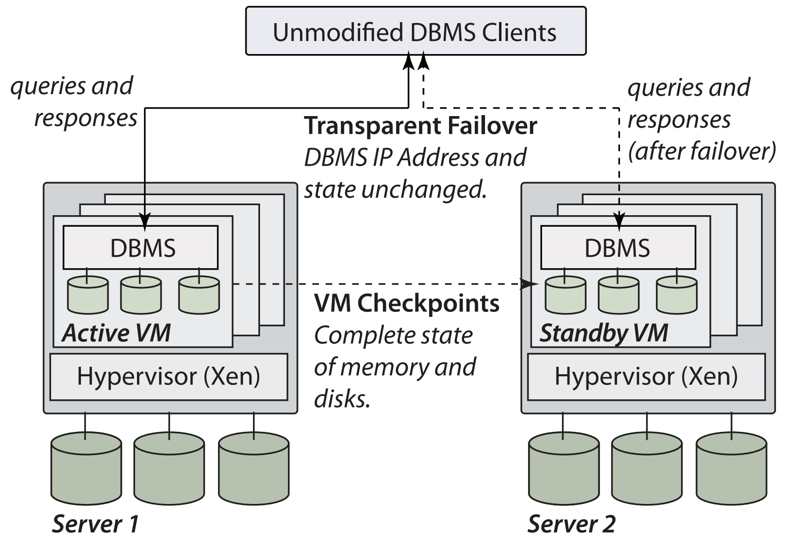


Figure 3.1: RemusDB System Architecture

overhead that it imposes on database workloads (Section 3.8). This modeling would be useful for a database administrator deploying RemusDB.

## 3.1 Background and System Overview

In our setup, shown in Figure 3.1, two servers are used to provide HA for a DBMS. One server hosts the *active* VM, which handles all client requests during normal operation. As the active VM runs, its entire state including memory, disk, and active network connections are continuously checkpointed to a *standby* VM on a second physical server. Our objective is to tolerate a failure of the server hosting the active VM by *failing over* to the DBMS in the standby VM, while preserving full ACID transactional guarantees. In particular, the effects of transactions that commit (at the active VM) before the failure should persist (at the standby VM) after the failover, and failover should not compromise transaction atomicity.

During normal operation, Remus takes frequent, incremental checkpoints of the complete state of the virtual machine on the active server. The time between two checkpoints is referred to as an *epoch*. These checkpoints are shipped to the standby server and "installed" in the virtual machine there. If the standby times out while waiting for a checkpoint, it assumes that the active server has failed. This causes a failover, and the standby VM begins execution from the most recent checkpoint that was completed prior to the failure. This failover is completely transparent to clients. The standby VM has the same IP address as the active VM, and similar to live migration [34], after failover the standby host issues an Address Resolution Protocol (ARP) update which ensures that network packets going to the (dead) active VM are automatically routed to the (live) standby VM. In checkpoint-based

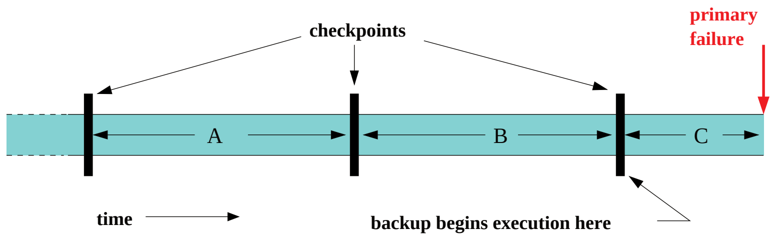


Figure 3.2: A Primary Server Execution Timeline

whole-machine protection systems like Remus, the virtual machine on the standby server does *not* mirror the execution at the active server during normal operation. Rather, the activity at the standby server is limited to installation of incremental checkpoints from the active server, which reduces the resource consumption at the standby.

Remus's checkpoints capture the entire state of the active VM, which includes disk, memory, CPU, and network device state. Thus, this captures both the state of the database and the internal execution state of the DBMS, e.g., the contents of the buffer pool, lock tables, and client connection state. After failover, the DBMS in the standby VM begins execution with a completely warmed up buffer pool, picking up exactly where the active VM was as of the most recent checkpoint, with all session state, TCP state, and transaction state intact. This fast failover to a warm backup and with no loss of client connections is an important advantage of our approach. Some DBMS-level HA solutions provide similar features, but these features add more code and complexity to the already complex systems. With our approach, these features are essentially free.

Figure 3.2 shows a simplified timeline illustrating checkpoints and failover. In reality, checkpoint transmission and acknowledgement is carefully overlapped with execution to increase performance while maintaining consistency [37]. However, the simplified timeline shown in Figure 3.2 is sufficient to illustrate the important features of this approach to DBMS high availability. When the failure occurs in Figure 3.2, all of the work accomplished by the active server during epoch C is lost. If, for example, the active server had committed a database transaction T during epoch C, any trace of that commit decision will be destroyed by the failure. Effectively, the execution of the active server during each interval is *speculative* until the interval has been checkpointed, since it will be lost if a failure occurs. Remus controls *output commit* [140] to ensure that the external world (e.g., the DBMS clients) sees a consistent view of the server's execution, despite failovers. Specifically, Remus queues and holds any outgoing network packets generated by the active server until the completion of the next checkpoint. For example, outgoing packets generated by the active server during epoch B in Figure 3.2 will be held by Remus until the completion of the checkpoint at the end of B, at which point they will be released. Similarly, a commit acknowledgement for transaction T, generated during epoch C, will be held by Remus and will be lost when the failure occurs. This *network buffering* ensures that no client will have been able to observe the speculative commit of T and conclude (prematurely or incorrectly) that T is durably committed. The output commit principle is also applied to the disk writes generated at the active server during an epoch. At the standby server, Remus buffers the writes received from active server during epoch B and releases them to its disk only at the end of the epoch. In the case of failure during epoch C, Remus discards the buffered writes of this epoch, thus maintaining the overall consistency of the system.

For a DBMS, the size of a Remus checkpoint may be large, which increases checkpointing overhead. Additionally, network buffering introduces message latency which may have a significant effect on the performance of some database workloads. RemusDB [101, 102] extends Remus with optimizations for reducing checkpoint size and for reducing the latency added by network buffering. We present these optimizations in the Sections 3.4 and 3.5.

## 3.2 Related Work

Widely-used logging and checkpointing techniques, such as ARIES [104], together with database backups, allow DBMSes to recover from server failures. After a failure, the DBMS runs a recovery protocol that uses the contents of the log to ensure that the database (or a restored database backup) is in a consistent state that includes all of the effects of transactions that committed before the failure. Once the database has been restored, the DBMS can begin to accept new work. However, since the DBMS cannot perform new work until the database has been restored, the recovery process can lead to an unacceptably long period of unavailability. Thus, many DBMSes provide additional high-availability features, which are designed to ensure that little or no down time will result from a server failure.

As described in Chapter 2, several types of HA techniques are used in database systems, sometimes in combination. In *shared access* approaches, two or more database server instances share a common storage infrastructure, which holds the database. The storage infrastructure stores data redundantly, e.g., by mirroring it on multiple devices, so that it is reliable. In addition, the storage interconnect (e.g., a SAN), through which the servers access the stored data, must be made reliable through the use of redundant access pathways. In case of a database server failure, other servers with access to the same database can take over the failed server's workload. Examples of this approach include Oracle RAC [110], which implements a virtual shared buffer pool across server instances, failover clustering in Microsoft SQL Server [78], and synchronized data nodes accessed through the NDB backend API in MySQL Cluster [108]. RemusDB differs from these techniques in that it does not rely on a shared storage infrastructure.

Active-standby approaches, which we introduced earlier, are designed to operate in a shared-nothing environment. Many database systems [32, 78, 108, 109] implement some form of active-standby HA. In some cases, the primary and backup can be run in an active-active configuration, allowing some read-only application work to be performed against the slave database, which may be slightly stale with respect to the primary.

In active-standby systems, update propagation may be physical, logical (row-based), or statement-based. Propagation, which is sometimes known as *log shipping*, may be synchronous or asynchronous. In the former case, transaction commits are not acknowledged to the database client until both the active and standby systems have durably recorded the update, resulting in what is known as a *2-safe* system [61, 120]. A 2-safe system ensures that a single server failure will not result in lost updates, but synchronous update propagation may introduce substantial performance overhead. In contrast, asynchronous propogation allows transactions to be acknowledged as soon they are committed at the primary. Such *1-safe* systems impose much less overhead during normal operation, but some recently-committed (and acknowledged) transactions may be lost if the primary fails. RemusDB, which is itself an active-standby system, uses asynchronous checkpointing to propagate updates to the standby. However, by controlling the release of output from the primary server, RemusDB ensures that committed transactions are not acknowledged to the client until they are recorded at the standby. Thus, RemusDB is 2-safe. RemusDB also differs from other database active-standby systems in that it protects the entire database server state, not just the database.

Like active-standby systems, multi-master systems (also known as update anywhere or group systems [60]) achieve high availability through replication. Multi-master systems relax the restriction that all updates must be performed at a single site. Instead, all replicas handle user requests, including updates. Replicas then propagate changes to other replicas, which must order and apply the changes locally. Various techniques, such as those based on quorum consensus [54, 144] or on the availability of an underlying atomic broadcast mechanism [74], can be used to synchronize updates so that global one-copy serializability is achieved across all of the replicas. However, these techniques introduce both performance overhead and complexity. Alternatively, it is possible to give up on serializablility and expose inconsistencies to applications. However, these inconsistencies must then somehow be resolved, often by applications or by human administrators. RemusDB is based on the simpler active-standby model, so it need not address the update synchronization problems faced by multi-master systems.

Virtualization has been used to provide high availability for arbitrary applications running inside virtual machines, by replicating the entire virtual machine as it runs. Replication can be achieved either through event logging and execution replay or whole machine checkpointing. While event logging requires much less bandwidth than whole machine checkpointing, it is not guaranteed to be able to reproduce machine state unless execution can be made *deterministic*. Enforcing determinism on commodity hardware requires careful management of sources of non-determinism [24, 48], and becomes infeasibly expensive to enforce on shared-memory multiprocessor systems [5, 49, 157]. Respec [83] does provide deterministic execution recording and replay of multithreaded applications with good performance by lazily increasing the level of synchronization it enforces depending on whether it observes divergence during replay, but it requires intricate modifications to the operating system. It also requires re-execution to be performed on a different core of the same physical system, making it unsuitable for HA applications. For these reasons, the replay-based HA systems of which we are aware support only uniprocessor VMs [130]. RemusDB uses whole machine checkpointing, so it supports multiprocessor VMs.

## 3.3 System Design

While the simplicity and transparency with which Remus provides high availability is desirable, applying Remus to database workloads is not an ideal fit for a number of reasons. First, as described above, Remus continuously transmits checkpoints of the running virtual machine to the backup host, resulting in a steady flow of replication traffic that is commensurate with the amount of memory that has changed between checkpoints; the large amount of memory churn in database workloads results in a high degree of replication traffic. The large amount of replication data makes checkpoints slower and results in a significant performance overhead for database workloads.

Second, the fact that Remus controls output commit by buffering every transmitted packet is over-conservative for database systems, which already provide higher-level transactional semantics. Client-server interactions with a database system typically involve several round trips on the fast, local area network. Within a transaction, the delay introduced by network buffering on messages from the server in each round trip results in an amplification of Remus's existing latency overheads. Moreover, the high memory churn rate of database workloads compounds this problem by requiring longer checkpoint epochs, resulting in longer delays for network buffering. For example, in a run of the TPC-C benchmark on Postgres in our experimental setting (described in Section 3.7) we observed that Remus introduced an overhead of 32% compared to the unprotected case. Turning off network buffering for this benchmark run reduced the overhead to 7%.

In designing RemusDB, we aimed to adapt Remus to address these two issues. In both cases, we observed that Remus's goal of providing HA that is completely transparent to the DBMS was excessively conservative and could be relaxed, resulting in a large reduction in overhead. More precisely, we made the following two observations:

1. **Not all changes to memory need to be sent to the backup.** In attempting to maintain an exact replica of the protected VM on the backup system, Remus was transmitting every page of memory whose contents changed between epochs. However, many page updates can either be reconstructed, as

with clean pages in the buffer pool that can be reloaded from disk, or thrown away altogether, in the case of working memory that can be recomputed or safely lost.

2. **Not all transmitted messages need output commit.** Buffering transmitted messages until the checkpoint that generated them has been protected prevents the system from exposing execution state that is rolled back (and so lost) in the event of failure. In a DBMS environment, this intermediate state is already protected by transaction boundaries. In light of this, we may relax output commit to the point that it preserves transactional semantics.

In addition to relaxing the comprehensiveness of protection in Remus to reduce overhead, our analysis of database workloads revealed one additional insight about these workloads that allowed further optimization:

3. **While changes to memory are frequent, they are often small.** Remus uses hardware page protection to identify the pages that have changed in a given checkpoint, and then transfers those pages to the backup at page granularity. Our analysis revealed that memory updates in database workloads were often considerably smaller than page size, and could consequently be compressed fairly effectively.

Remus was adapted in light of each of these observations, in order to provide more efficient high availability for database workloads. Section 3.4 discusses optimizations related to how memory is tracked on the primary VM and replicated over the network to the backup. Section 3.5 describes how latency overheads have been reduced by relaxing network buffering in some situations.

## 3.4 Memory Optimizations

Remus takes a deliberately simple approach to memory checkpointing: at every checkpoint, it copies all the pages of memory that change from the active host and transmits them over the network to the backup host. The authors of Remus argue that this simplicity is desirable: it provides high availability with an acceptable degree of overhead, with an implementation that is simple enough that one can have confidence in its correctness, regardless of the target application or hardware architecture. This is in stark contrast to the complexity of previous systems, even those implemented in the hypervisor [24]. And while this argument for simplicity holds for database systems, the overhead penalty is higher: database workloads tend to modify more memory in each checkpoint epoch than other workloads. This section describes a set of optimizations designed to reduce this overhead.

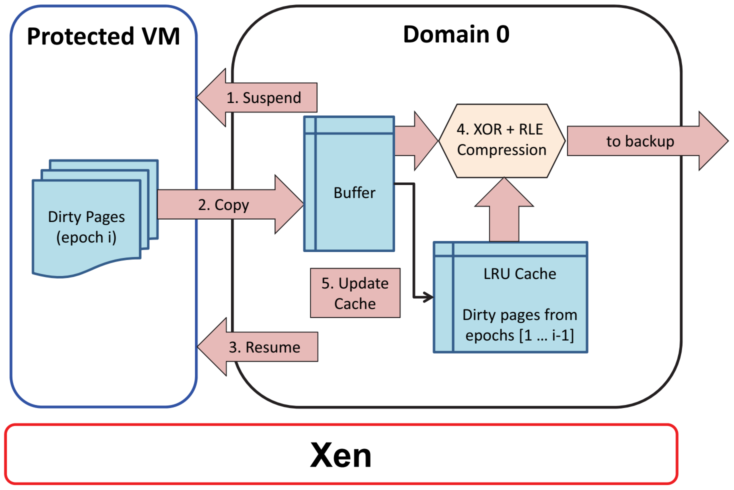


Figure 3.3: Checkpoint Compression Workflow

### 3.4.1 Sending Less Data

Compressing checkpoints is beneficial when the amount of data to be replicated is large, and the data contains redundancy. Our analysis found that both of these conditions apply to database workloads: (1) they involve a large set of frequently changing pages of memory (most notably buffer pool pages), and (2) the memory writes often change only a small part of the pages on which they occur. This presents an opportunity to achieve a considerable reduction in replication traffic by only sending the actual changes to these pages.

To achieve this, we implemented an LRU-based cache of frequently changing pages from previous checkpoints. This cache is maintained in *domain 0*, the privileged VM used for control by the Xen hypervisor. Our experimentation showed that a cache size of 10% of VM memory offers the desired performance improvement while maintaining an acceptable memory footprint in domain 0. When sending pages to the backup, we first check to see if the previous version of the page exists in this cache. If it does, the contents of the two pages are XORed, usually resulting in a page that contains mostly zeros, reflecting the large amount of identical data. The result is then run-length encoded for transmission. If the page is not found in the cache, it is sent uncompressed, and is added to the cache using the standard LRU eviction policy. Figure 3.3 illustrates the workflow for checkpoint compression.

The original Remus work maintained that asynchronous, pipelined checkpoint processing while the active VM continues to execute is critical to minimizing its performance impact. The benefits of this approach were evident in implementing checkpoint compression: moving the implementation into an asynchronous stage and allowing the VM to resume execution in parallel with compression and replication in domain 0 halved the overhead of RemusDB.

### 3.4.2 Protecting Less Memory

Compressed checkpoints help considerably, but the work involved in taking and sending checkpoints is still proportional to the amount of memory changed between checkpoints. In this section, we discuss ways to reduce checkpoint size by selectively *ignoring* changes to certain parts of memory. Specifically, a significant fraction of the memory used by a DBMS goes into the buffer pool. Clean pages in the buffer pool do not need to be sent in Remus checkpoints if they can be regenerated by reading them from the disk. Even dirty buffer pool pages can be omitted from Remus checkpoints if the DBMS can recover changes to these pages from the transaction log.

In addition to the buffer pool, a DBMS uses memory for other purposes such as lock tables, query plan cache, working memory for query operators, and connection state. In general, memory pages whose contents can be regenerated, or alternatively can be safely thrown away may be ignored during checkpointing. Based on these observations, we developed two checkpointing optimizations: *disk read tracking* and *memory deprotection.*

#### Disk Read Tracking

Remus, like the live VM migration system on which it is based [34], uses hardware page protection to track changes to memory. As in a copy-on-write process fork, all of the page table entries of a protected virtual machine are set to read only, producing a trap when any page is modified. The trap handler verifies that the write is allowed, then updates a bitmap of "dirty" pages, which determines the set of pages to transmit to the backup server at each checkpoint. This bitmap is cleared after the checkpoint is taken.

Because Remus keeps a synchronized copy of the disk on the backup, any pages that have been read from disk into memory may be safely excluded from the set of dirty pages, as long as the memory has not been modified after the page was read from disk. Our implementation interposes on disk read requests from the virtual machine and tracks the set of memory pages into which the reads will be placed, and the associated disk addresses from which those pages were read. Normally, the act of reading data from disk into a memory page would result in that page being marked as dirty and included in the data to be copied for the checkpoint. Our implementation does *not* mark that page dirty, and instead adds an annotation to the replication stream indicating the sectors on disk that may be read to reconstruct the page remotely.

Normally, writes to a disk pass through the operating system's (or DBMS's) buffer cache, and this will inform Remus to invalidate the read-tracked version of the page and add it to the set of pages to transmit in the next checkpoint. However, it is possible that the contents of the sectors on disk that a read-tracked page refers to may be changed without touching the in-memory read-tracked page. For example, a process different from the DBMS process can perform a direct (unbuffered) write to the file from which the read-tracked page is to be read after failure. In this case, read tracking would incorrectly recover the newer version of the page on failover. Although none of the database systems that we studied exhibited this problem, protecting against it is a matter of correctness, so RemusDB maintains a set of backpointers from read-tracked pages to the associated sectors on disk. If the VM writes to any of these sectors, we remove the page from the read tracking list and send its contents normally.

### Memory Deprotection

Our second memory optimization aims to provide the DBMS with a more explicit interface to control which portions of its memory should be deprotected (i.e., not replicated during checkpoints). We were surprised to find that we could not produce performance benefits over simple read tracking using this interface.

The idea for memory deprotection stemmed from the Recovery Box [17], a facility for the Sprite OS that replicated a small region of memory that would provide important recent data structures to speed up recovery after a crash (Postgres session state is one of their examples). Our intuition was that RemusDB could do the opposite, allowing the majority of memory to be replicated, but also enabling the DBMS to flag high-churn regions of working memory, such as buffer pool descriptor tables, to be explicitly deprotected and a recovery mechanism to be run after failover.

The resulting implementation was an interesting, but ultimately useless interface: The DBMS is allowed to deprotect specific regions of virtual memory, and these addresses are resolved to physical pages and excluded from replication traffic. On failover, the system would continue to run but deprotected memory would suddenly be in an unknown state. To address this, the DBMS registers a *failover callback handler* that is responsible for handling the deprotected memory, typically by regenerating it or dropping active references to it. The failure handler is implemented as an idle thread that becomes active and gets scheduled only after failover, and that runs with all other threads paused. This provides a safe environment to recover the system.

While we were able to provide what we felt was both a natural and efficient implementation to allow the deprotection of arbitrary memory, it is certainly more difficult for an application writer to use than our other optimizations. More importantly, we were unable to identify any easily recoverable data structures for which this mechanism provided a performance benefit over read tracking. One of the reasons for this is that memory deprotection adds CPU overhead for tracking deprotected pages during checkpointing, and the savings from protecting less memory need to outweigh this CPU overhead to result in a net benefit. We still believe that the interface may be useful for other applications and workloads, but we have decided not to use it in RemusDB.

To illustrate our reasoning, we ran a TPC-H benchmark on Postgres with support for memory deprotection in our experimental setting. Remus introduced 80% overhead relative to an unprotected VM. The first data structure we deprotected was the shared memory segment, which is used largely for the DBMS buffer pool. Unsurprisingly, deprotecting this segment resulted in roughly the same overhead reduction we achieved through read tracking (bringing the overhead down from 80% to 14%), but at the cost of a much more complicated interface. We also deprotected the dynamically allocated memory regions used for query operator scratch space, but that yielded only an additional 1% reduction in overhead. We conclude that for the database workloads we have examined, the *transparency vs. performance* tradeoff offered by memory deprotection is not substantial enough to justify investing effort in complicated recovery logic.

## 3.5 Commit Protection

Irrespective of memory optimizations, the single largest source of overhead for many database workloads on the unmodified Remus implementation was the delay introduced by buffering network packets for controlling output commit. Client-server interactions in DBMS environments typically involve long-lived sessions with frequent interactions over low-latency local area networks. For example, a TPC-C transaction on Postgres in our experiments has an average of 32 packet exchanges between client and server, and a maximum of 77 packet exchanges. Remus's network buffering delays all these packets; packets that might otherwise have round trip times on the order of hundreds of microseconds are held until the next checkpoint is complete, potentially introducing two to three orders of magnitude in latency per round trip.

In RemusDB, we exploit database transaction semantics to avoid much of Remus' network buffering, and hence to eliminate much of the performance overhead that network buffering introduces. The purpose of network buffering in Remus is to avoid exposing the client to the results of speculative server processing until it has been checkpointed. In RemusDB, we relax this behavior by allowing server communications resulting from speculative processing to be released immediately to the client, *but only within the scope of an active database transaction.* If the client attempts to commit a transaction, RemusDB will buffer and delay the commit acknowledgement until that transaction is safe, i.e., until the processing of that transaction has been checkpointed. Conversely, if a failure occurs during the execution of such a transaction, RemusDB will ensure that the transaction is aborted on failover. Relaxing Remus's network buffering in this way allows RemusDB to release most outgoing network packets without delay. However, failover is no longer completely transparent to the client, as it would be in Remus, as a failover may necessitate the abort of some in-progress transactions. As long as failures are infrequent, we expect this to be a desirable tradeoff.

**At COMMIT WORK:**

```
protect the client's socket
perform normal DBMS commit processing
send the COMMIT acknowledgement
deprotect the socket
```

**On failover (at the standby host):**

```
for each active transaction $t$ do
    if $t$ is not committing then ABORT $t$
end for
```

Figure 3.4: The Commit Protection Protocol

To implement this approach in RemusDB, we modified the hosted DBMS to implement a protocol we call *commit protection.* The commit protection protocol requires fine (message level) control over which outgoing messages experience network buffering and which do not. To support this, RemusDB generalizes Remus's communication abstraction. Stream sockets, which are implemented on top of TCP, guarantee in-order message delivery, and database systems normally use stream sockets for communication with clients. In RemusDB, each stream socket can be in one of two states: protected or unprotected. RemusDB provides the hosted DBMS with *protect* and *deprotect* operations to allow it to change the socket state. Outgoing messages sent through a protected socket experience normal Remus network buffering, i.e., they are delayed until the completion of the next Remus commit. Messages sent through an unprotected socket are not subjected to network buffering and are released immediately. RemusDB preserves in-order delivery of all messages delivered through a socket, regardless of the state of the socket when the message is sent. Thus, an unprotected message sent shortly after a protected message may be delayed to ensure that it is delivered in the correct order.

The hosted DBMS implements the commit protection protocol using the socket protection mechanism. The commit protocol has two parts, as shown in Figure 3.4. The first part of the protocol runs when a client requests that a transaction commit. The server protects the transaction's socket before sending the commit acknowledgement to the client. All transaction output up until the arrival of the commit request is sent unprotected. The second part of the protocol runs at the standby server after a failover, and causes all active transactions that are not committing to abort. Remus is designed to run a recovery thread in the standby VM as soon as it takes over after a failure. In RemusDB, the recovery thread runs inside the standby DBMS and implements the failover part of the commit protection protocol. Once the recovery thread finishes this work, the DBMS resumes execution from state captured by the most recent pre-failover checkpoint. Note that, on failover, the recovery handler will see transaction states as they were at the time of the last pre-failure checkpoint.

### 3.5.1 Correctness of Commit Protection

In this section we state more precisely what it means for the commit protocol to behave correctly. Essentially, if the client is told that a transaction has committed, then that transaction should remain committed after a failure. Furthermore, if an active transaction has shown speculative results to the client and a failure occurs, then that transaction must ultimately abort. These guarantees are stated in the following lemmas.

**Lemma 3.5.1** (Fail-safe Commit)**.** *For all transactions $T$ that are created at the active server prior to the point of failure, if a client receives a commit acknowledgement for $T$, then $T$ will be committed at the standby site after failover.*

*Proof.* COMMIT WORK acknowledgements are always sent using a protected socket, which does not release messages to the client until a checkpoint has occurred. If the client has received a commit acknowledgement for $T$, then a server checkpoint must have occurred after $T$'s commit message was sent and thus after the active server made the commit decision for $T$. Thus, the active server's commit decision for $T$ (and $T$'s effects) will be captured by the checkpoint and reflected at the standby site after the failure. Furthermore, at the time of the checkpoint, $T$ will either have been committing at the active site or it will have finished. Since the recovery thread at the standby site only aborts active transactions that are not committing, it will not attempt to abort $T$. □

**Lemma 3.5.2** (Speculation)**.** *For all transactions $T$ that are created at the active server prior to the point of failure, if $T$'s client does not submit a COMMIT WORK request for $T$ prior to the failure, then either $T$ will be aborted at the standby server after the failure, or it will not exist there at all.*

*Proof.* Let $C$ represent the last checkpoint at the active server prior to its failure. There are three cases to consider. First, $T$ may have started after $C$. In this case, $T$ will not exist at the time of the checkpoint $C$, and therefore it will not exist at the standby server after failover. Second, $T$ may have started before $C$ and remained active at $C$. In this case, some of $T$'s effects may be present at the standby site because they are captured by $C$. Since the client has not submitted a COMMIT WORK request, $T$ cannot have been committing at the time of $C$. Therefore, the recovery thread will see $T$ as an active, non-committing transaction at failover and will abort $T$. The third case is that $T$ may have started, aborted, and finished prior to $C$. In this case, the checkpoint will ensure that $T$ is also aborted at the standby site after failover. □

### 3.5.2 Implementation of Protection and Deprotection

To provide a DBMS with the ability to dynamically switch a client connection between protected and deprotected modes, we added a new `setsockopt()` option

|  | Virtualization Layer | Guest VM Kernel | DBMS |
|---|---|---|---|
| Commit Protection | 13 | 396 | 103(Postgres), 85(MySQL) |
| Disk Read Tracking | 1903 | 0 | 0 |
| Compression | 593 | 0 | 0 |

Table 3.1: RemusDB Source Code Modifications (lines of code)

to Linux. A DBMS has to be modified to make use of protection and deprotection via the commit protection protocol shown in Figure 3.4. We have implemented commit protection in Postgres and MySQL, with minor modifications to the client connection layer. Because the changes required are for a small and well-defined part of the client/server protocol, we expect them to be easily applied to any DBMS. Table 3.1 provides a summary of the source code changes made to different subsystems to implement the different optimizations that make up RemusDB.

One outstanding issue with commit protection is that while it preserves complete application semantics, it exposes TCP connection state that can be lost on failover: unbuffered packets advance TCP sequence counters that cannot be reversed, which can result in the connection becoming inconsistent after failover and stalling until it times out. In the current implementation of RemusDB we have not addressed this problem: only a small subset of connections are affected, and the transactions occurring over them will be recovered when the connection times out just like any other timed out client connection. One possible direction for future work is to explore techniques by which we can track sufficient state to explicitly close TCP connections that have become inconsistent at failover time, in order to speed up transaction recovery time for those sessions.

## 3.6 Reprotection After Failure

When the primary host crashes, the backup host takes over and becomes the new primary. When the original, now-crashed primary host comes back online, it needs to assume the role of backup host. For that to happen, the storage (i.e., disks) of the VMs on the two hosts must be resynchronized. The storage of the VM on the host that was failed and is now back online must catch up with the storage of the VM on the other, now-primary host. After this storage synchronization step, checkpointing traffic can resume between the primary and backup host.

For the protected DBMS to remain available during storage synchronization, this synchronization must happen online, while the DBMS in the primary VM is running. The storage replication driver used by Remus is based on Xen's Blktap2 driver [156] and does not provide a means for online resynchronization of storage. One way to perform the required online resynchronization is to use a brute-force approach and copy all the disk blocks from the primary to the backup. This is sufficient to ensure correctness, but it would impose unnecessary load on the disk subsystem and increase the time to restart HA. A better approach, which we adopt in RemusDB, is used by the SecondSite system [122] that uses Distributed Replicated Block Device (DRBD) [47] to perform online synchronization of disk blocks between the new primary/old backup VM and the old primary VM. In this approach, only the disk blocks changed by the new primary/old backup VM after failure are copied over. The system also overwrites disk blocks written by the old primary VM during the last unfinished checkpoint with data from the backup. Note that we do not require the new backup VM to be present on the same physical host as the old primary VM. DRBD can synchronize a disk to an empty target that may reside on a new physical host. However, in this case the whole disk will have to be synchronized, thus requiring more time. In general, the time to synchronize varies depending on the amount of data, I/O bandwidth, and network bandwidth.

## 3.7 Experimental Evaluation

In this section we present an evaluation of RemusDB. The objectives of this evaluation are as follows:

- First, we wish to demonstrate that RemusDB is able to survive a failure of the primary server, and to illustrate the performance of RemusDB during and after a failover.
- Second, we wish to characterize the performance overhead associated with RemusDB during normal operation. We compare the performance of unoptimized Remus and optimized RemusDB against that of an unprotected DBMS to measure this overhead. We also consider the impact of specific RemusDB optimizations on different types of database workloads.
- Third, we consider how key system parameters and characteristics, such as the size of the DBMS buffer pool and the length of the Remus checkpoint interval, affect the overhead introduced by Remus.

### 3.7.1 Experimental Environment

Our experimental setup consists of two servers each equipped with two quad-core Intel Xeon processors, 16GB RAM, and two 500GB SATA disks. We use the Xen 4.0 hypervisor (64-bit), Debian 5.0 (32-bit) as the host operating system, and Ubuntu 8.04 (32-bit) as the guest operating system. XenLinux Kernel 2.6.18.8 is used for both host and guest operating systems, with disks formatted using the *ext3* filesystem.

We evaluate RemusDB with PostgreSQL 8.4.0 (referred to as Postgres) and MySQL 5.0, using three widely accepted benchmarks namely: TPC-C [145], TPC-H [146], and TPC-W [147]. We run TPC-C experiments on both Postgres and

| **DBMS** | **Postgres** | | | **MySQL** |
|---|---|---|---|---|
| **Benchmark** | TPC-C | TPC-H | TPC-W | TPC-C |
| **Performance Metric** | TpmC | Execution Time | WIPSb | TpmC |
| **Default Scale** | 20W/200C | 1 | 10K Items | 30W/300C |
| **Test Duration (mins)** | 30 | – | 20 | 30 |
| **DB Size (GB)** | 1.9 | 2.3 | 1.0 | 3.0 |
| **BP Size (MB)** | 190 | 750 | 256 | 300 |
| **VM Mem (GB)** | 2 | 1.5 | 2 | 2 |
| **vCPUs** | 2 | 2 | 2 | 2 |
| **Remus CPI (ms)** | 50 | 250 | 100 | 50 |

Table 3.2: Experimental Settings

MySQL while TPC-H and TPC-W experiments are run on Postgres only. We use a Remus checkpointing interval (CPI) of 50ms, 100ms, and 250ms for TPC-C, TPC-W, and TPC-H experiments, respectively. These different CPIs for each type of benchmark are chosen because they offer the best trade-off between overhead during normal execution and availability requirements of that particular workload. We evaluate the effect of varying CPI on the TPC-C and TPC-H benchmarks in Section 3.7.6.

Our default settings for TPC-C experiments are as follows: The virtual machine is configured with 2GB memory and 2 virtual CPUs. For MySQL, we use the Percona benchmark kit [117] with a database of 30 warehouses and 300 concurrent clients (10 clients per warehouse). The total size of the database on disk is 3GB. For Postgres, we use the TPCC-UVa benchmark kit [86] with a database of 20 warehouses (1.9GB database on disk) and 200 concurrent clients. We modified the TPCC-UVa benchmark kit so that it uses one TCP connection per client; the original benchmark kit uses one shared connection for all clients. We choose different scales for MySQL and Postgres due to differences in how they scale to larger workloads when provided with fixed (equal) resources. The database buffer pool is configured to be 10% of the database size on disk. We do not use connection pooling or a transaction monitor; each client directly connects to the DBMS.

Our default settings for TPC-H experiments are a virtual machine with 1.5GB memory, 2 virtual CPUs, and a database with TPC-H scale factor 1. The total size of the database on disk is 2.3GB. We configure Postgres with a buffer pool size of 750MB. Our TPC-H experiments consist of one *warmup* run where we execute the 22 read-only TPC-H queries sequentially, followed by one *power stream* run [146] where we execute the queries sequentially and measure the total execution time. We do not perform TPC-H throughput tests or use the refresh streams.

Lastly, for TPC-W experiments we use the TPC-W implementation described in [72]. We use a two tier architecture with Postgres in one tier and three instances of Apache Tomcat v6.0.26 in the second tier, each running in a separate VM. Postgres runs on a virtual machine with 2GB memory, and 2 virtual CPUs. We use a TPC-W database with 10,000 items (1GB on disk). Postgres's buffer pool is configured to be 256MB. Each instance of Apache Tomcat runs in a virtual machine with 1GB memory, and 1 virtual CPU. In these experiments, when running with Remus, only the Postgres VM is protected. In order to avoid the effects of virtual machine scheduling while measuring overhead, we place the Tomcat VMs on a separate well provisioned physical machine.

Table 3.2 provides a summary of our experimental settings including the Remus checkpointing interval (CPI). We use the following abbreviations to refer to different RemusDB optimizations in our experiments: RT – Disk Read Tracking, ASC – Asynchronous Checkpoint Compression, and CP – Commit Protection.

### 3.7.2 Behavior of RemusDB During Failover

In the first experiment, we show RemusDB's performance in the presence of failures of the primary host. We run the TPC-C benchmark against Postgres and MySQL and plot throughput in transactions per minute (TpmC). We run the test for 1 hour, and a failure of the primary host is simulated at 30 minutes by cutting power to it. We compare the performance of a database system protected by unoptimized Remus and by RemusDB with its two transparent optimizations (ASC, RT) in Figures 3.5 and 3.6. The performance of an unprotected database system (without HA) is also shown for reference. The throughput shown in the figure is the average throughput for a sliding window of 60 seconds. Note that MySQL is run with a higher scale (Table 3.2) than Postgres because of its ability to handle larger workloads when provided with the same resources.

Without any mechanism for high availability in place, the unprotected VM cannot serve clients beyond the failure point, i.e., throughput immediately drops to zero. All clients lose connections to the database server and cannot reconnect until someone (e.g., a DBA) manually restores the database to its pre-failure state. After restart, the database will recover from its crash consistent state using standard log recovery procedures [104]. The time to recover depends on how much state needs to be read from the write-ahead log and reapplied to the database and is usually in the order of several minutes. Furthermore, the unprotected VM will have to go through a warm-up phase again before it can reach its pre-failure steady state throughput (not shown in the graph).

Under both versions of Remus, when the failure happens at the primary physical server, the VM at the backup physical server recovers with $\leq 3$ seconds of downtime and continues execution. The database is running with a *warmed up buffer pool, no client connections are lost, and in-flight transactions continue to execute normally* from the last checkpoint. We only lose the speculative execution state generated at the primary server since the last checkpoint. In the worst case, Remus loses one checkpoint interval's worth of work. But this loss of work is completely transparent to the client since Remus only releases external state at checkpoint boundaries. After the failure, throughput rises sharply and reaches a steady state comparable

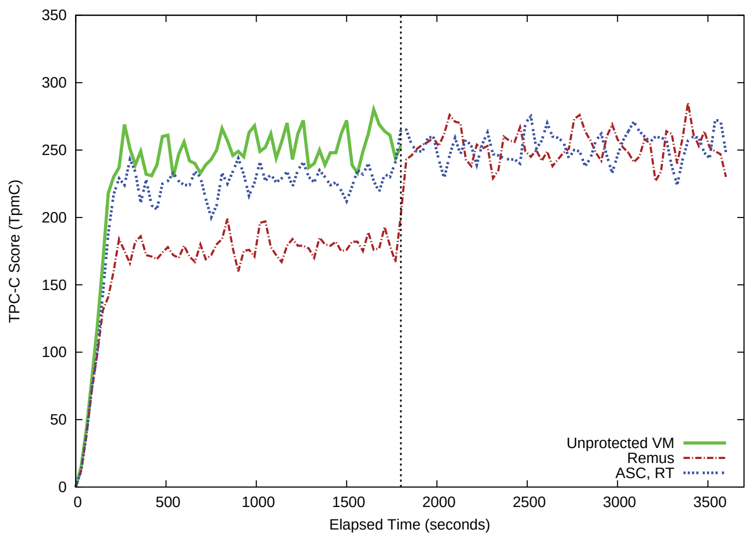


Figure 3.5: TPC-C Failover (Postgres)

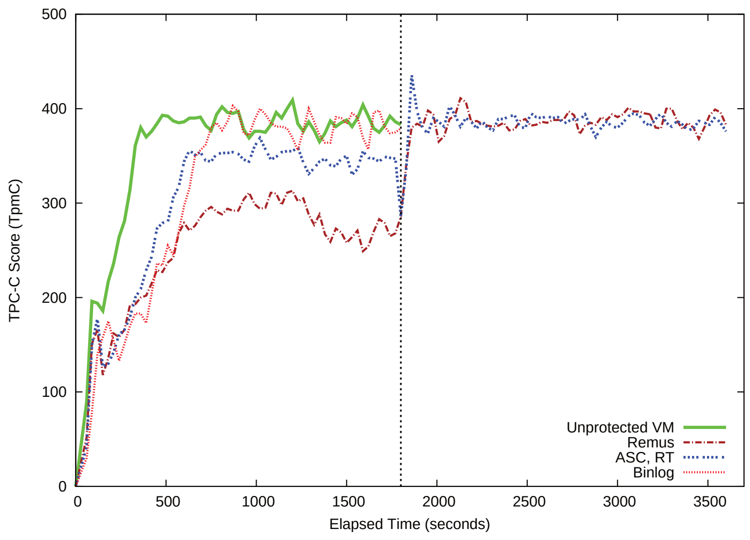


Figure 3.6: TPC-C Failover (MySQL)

to that of the unprotected VM before the failure. This is because the VM after the failure is not protected, so we do not incur the replication overhead of Remus.

Figure 3.6 also shows results with MySQL's integrated replication solution, Binlog [111, Section 5.2.3]. Our experiments use the stable release of Postgres at the time of conducting this research, and this version does not provide integrated HA support. MySQL Binlog replication, in combination with monitoring systems like Heartbeat [85], provides performance very close to that of an unprotected VM and can recover from a failure with $\leq 5$ seconds of server downtime. However, we note that RemusDB has certain advantages when compared to Binlog replication:

- *Completeness.* On failover, Binlog replication can lose up to one transaction even under the most conservative settings [111, Section 16.1.1.1]. In contrast, even with aggressive optimizations such as commit protection, RemusDB never loses transactions.
- *Transparency.* Client-side recovery is more complex with Binlog, which loses all existing client sessions at failure. To measure Binlog performance after recovery (not shown in Figure 3.6), we had to modify the TPC-C client to reconnect after the failure event. This violates the TPC specification, which requires that clients not reconnect if their server context has been lost [145, Section 6.6.2]. Because we are comparing server overhead, we minimized the client recovery time by manually triggering reconnection immediately upon failover. In practice, DBMS clients would be likely to take much longer to recover, since they would have to time-out their connections.
- *Implementation complexity.* Binlog accounts for approximately 18K lines of code in MySQL, and is intricately tied to the rest of the DBMS implementation. Not only does this increase the effort required to develop the DBMS (as developers must be cautious of these dependencies), but it also results in constant churn for the Binlog implementation, ultimately making it more fragile. Binlog has experienced bugs proportionate to this complexity: more than 700 bugs were reported over the last 3 years before the time of this writing.

### 3.7.3 Reprotection After a Failure

In Figure 3.7, we show RemusDB's reprotection mechanism in action. Similar to the failover experiment in Section 3.7.2, we run the TPC-C benchmark against Postgres and plot throughput in transactions per minute (TpmC). We run the test for 1 hour, a failure of the primary host is simulated at 30 minutes by cutting power to it. The performance of an unprotected database system (without HA) is also shown for reference. The setting used for these experiments is slightly different from the other experiments in this section. In particular, the storage backend used for reprotection experiments is different since it supports online resynchronization of VM disks after a failure. Because of that, the performance numbers are slightly

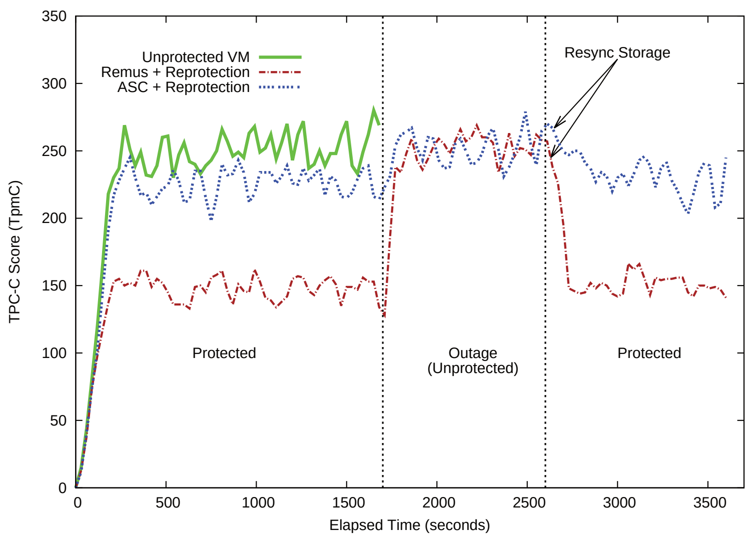


Figure 3.7: TPC-C Failover and Reprotection After Failure (Postgres)

lower than other experiments, as can be clearly seen from the line for unmodified Remus.

During the outage period, the VM does not incur any checkpointing overhead and hence the throughput rises to that of an unprotected system. After a 15 minute outage period, the primary host is brought back online. Note that we let the outage last for 15 minutes in this experiment in order to adequately observe performance during an outage. In reality, we can detect a failure within 3 seconds, and resynchronize storage within approximately 29 seconds. The time required to resynchronize can vary depending on the amount of data that needs to be resynchronized. Once storage resynchronization completes, we restart the replication process: all of the VM's memory is copied to the backup, and then Remus checkpoints are resumed. This process takes approximately 10 seconds in our settings for a VM with 2GB of memory and a gigabit ethernet connection between the primary and backup hosts. After this point, the VM is once again HA, i.e., it is protected against a failure of the backup. Note that after reprotection, the throughput returns back to pre-failure levels as shown in Figure 3.7. For implementing reprotection, we utilized a new storage backend namely DRBD [47] which allows efficient online resynchronization of VM disks. This storage backend does not support read tracking (adding this support is a matter of implementation). Hence, Figure 3.7 only shows RemusDB's performance with the ASC optimization.

### 3.7.4 Overhead During Normal Operation

Having established the effectiveness of RemusDB at protecting from failure, its fast failover time, and its effectiveness in reprotection after failure, we now turn our attention to the overhead of RemusDB during normal operation. This section serves two goals: (1) it quantifies the overhead imposed by unoptimized Remus on normal operation for different database benchmarks, and (2) it measures how the RemusDB optimizations affect this overhead when applied individually or in combination. For this experiment we use the TPC-C, TPC-H, and TPC-W benchmarks.

Figures 3.8 and 3.9 present TPC-C benchmark results for Postgres and MySQL, respectively. In each case the benchmark was run for 30 minutes using the settings presented in Table 3.2. On the x-axis, we have different RemusDB optimizations and on the y-axis we present TpmC scores normalized with respect to an unprotected (base) VM. The normalized score is defined as: (TpmC with optimization being evaluated)/(Base TpmC). The TpmC score reported in these graphs takes into account all transactions during the measurement interval irrespective of their response time requirements. Base VM scores are 243 and 365 TpmC for Postgres and MySQL, respectively. The score of unoptimized Remus (leftmost bar) is around 0.68 of the base VM score for both DBMS – representing a significant performance loss. It is clear from the graph that without optimizations, Remus protection for database systems comes at a very high cost. The next three bars in the graph show the effect of each RemusDB optimization applied individually. RT provides very little performance benefit because TPC-C has a small working set and dirties many of the pages that it reads. However, both ASC and CP provide significant performance gains. Performance with these optimizations is 0.9-0.97 of the base performance. TPC-C is particularly sensitive to network latency and both of these optimizations help reduce latency either by reducing the time it takes to checkpoint (ASC) or by getting rid of the extra latency incurred due to Remus's network buffering for all but commit packets (CP). The rightmost two bars in the graph show the effect of combining optimizations. The combination of all three optimizations (ASC, RT, CP) yields the best performance at the risk of a few transaction aborts (not losses) and connection failures. In multiple variations of this experiment we have observed that the variance in performance is always low, and that when the combination of (ASC, RT, CP) does not outright outperform the individual optimizations, the difference is within the range of experimental error.

The improvement in performance when adding (ASC, RT, CP) to Remus can be seen not only in throughput, but also in latency. The average latency of the NewOrder transactions whose throughput is plotted in Figures 3.8 and 3.9 for Postgres and MySQL, respectively, is 12.9 seconds and 19.2 seconds for unoptimized Remus. This latency is 1.8 seconds and 4.5 seconds for RemusDB. Compare this to the latency for unprotected VM which is 1.2 seconds for Postgres and 3.2 seconds for MySQL. Other experiments (not presented here) show that on average about 10% of the clients lose connectivity after failover when CP is enabled. In most cases, this is an acceptable trade-off given the high performance under (ASC, RT, CP) during normal execution. This is also better than many existing solutions where there is a possibility of losing not only connections but also committed transactions, which never happens in RemusDB.

Figure 3.10 presents the results for TPC-H with Postgres. In this case, the y-axis presents the total execution time of a warmup run and a power test run

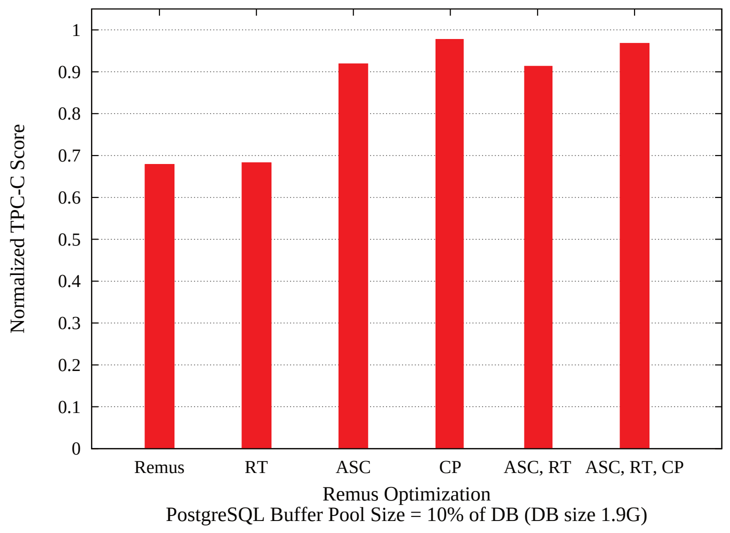


Figure 3.8: TPC-C Overhead (Postgres) [Base Score = 243 tpmC]

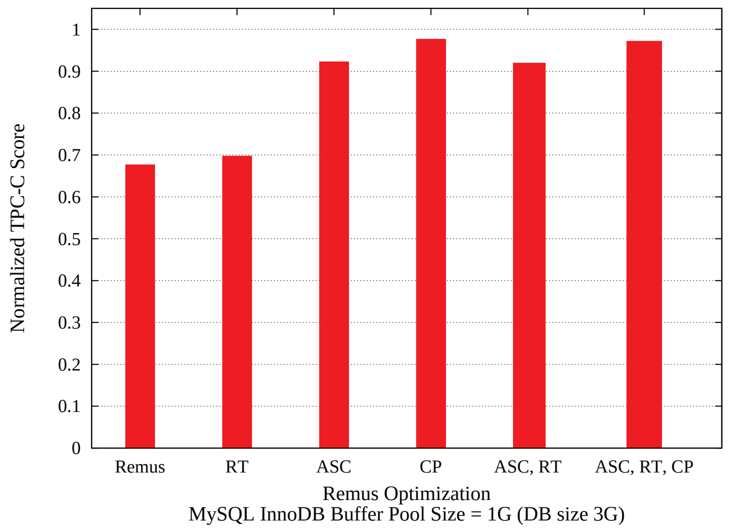


Figure 3.9: TPC-C Overhead (MySQL) [Base Score = 365 tpmC]

normalized with respect to the base VM's execution time (921 s). The normalized execution time is defined as: (Base execution time)/(Execution time with optimization being evaluated). Since TPC-H is a decision support benchmark that consists of long running compute and I/O intensive queries typical of a data warehousing environment, it shows very different performance gains with different RemusDB optimizations as compared to TPC-C. In particular, as opposed to TPC-C, we see some performance gains with RT because TPC-H is a read intensive workload, and absolutely no gain with CP because it is insensitive to network latency. A combination of optimizations still provides the best performance, but in case of TPC-H most of the benefits come from memory optimizations (ASC and RT). These transparent memory optimizations bring performance to within 10% of the base case, which is a reasonable performance overhead. Using the non-transparent CP adds no benefit and is therefore not necessary. Moreover, the opportunity for further performance improvement by using the non-transparent memory deprotection interface (presented in Section 3.4.2) is limited to 10%. Therefore, we conclude that it is not worth the additional complexity to pursue it.

Finally, we present the results for TPC-W with Postgres in Figure 3.11. Each test was run with the settings presented in Table 3.2 for a duration of 20 minutes. We drive the load on the database server using 252 Emulated Browsers (EBs) that are equally divided among three instances of Apache Tomcat, which in turn access the database to create dynamic web pages and return them to EBs, as specified by the TPC-W benchmark standard [147]. We use the TPC-W *browsing mix* with image serving turned off at the clients. The y-axis on Figure 3.11 presents TPC-W scores, Web Interactions Per Second (WIPS), normalized to the base VM score (36 WIPS). TPC-W behaves very similar to TPC-C workload: ASC and CP provide the most benefit while RT does not provide any benefit.

RemusDB has a lot to offer for a wide variety of workloads that we study in this experiment. This experiment shows that a combination of memory and network optimizations (ASC and CP) work well for OLTP style workloads, while DSS style workloads gain the most benefit from memory optimizations alone (ASC and RT). It also shows that by using the set of optimizations that we have implemented in RemusDB, we gain back almost all of the performance lost when going from an unprotected VM to a VM protected by unoptimized Remus.

### 3.7.5 Effects of DB Buffer Pool Size

In the previous experiment, we showed that memory optimizations (ASC and RT) offer significant performance gains for the TPC-H workload. The goal of this experiment is to study the effects of database buffer pool size on different memory optimizations on a micro level. In doing so, we hope to offer insights about how each of these optimization offers its performance benefits.

We run a scale factor 1 TPC-H workload, varying the database buffer pool size from 250MB to 1000MB. We measure the total execution time for the warmup

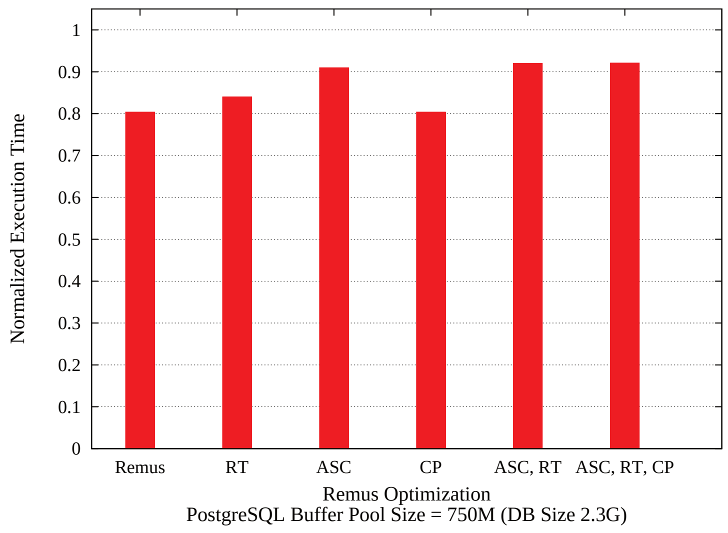


Figure 3.10: TPC-H Overhead (Postgres) [Base Runtime = 921 s]

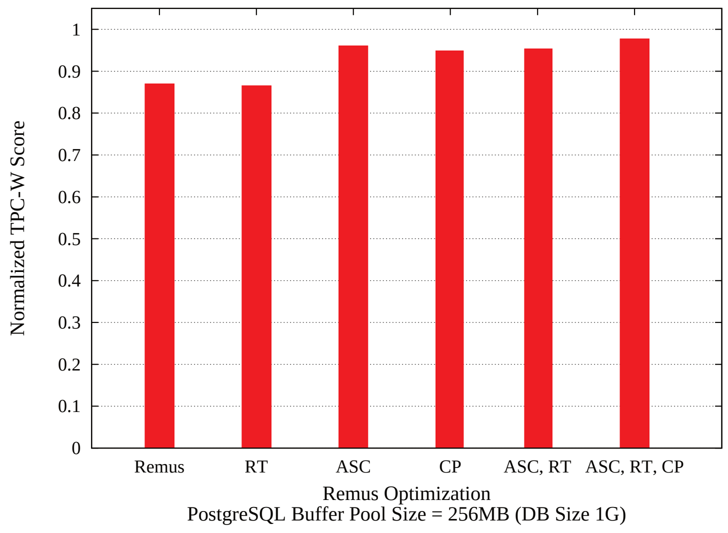


Figure 3.11: TPC-W Overhead (Postgres) [Base Score = 36 WIPS]

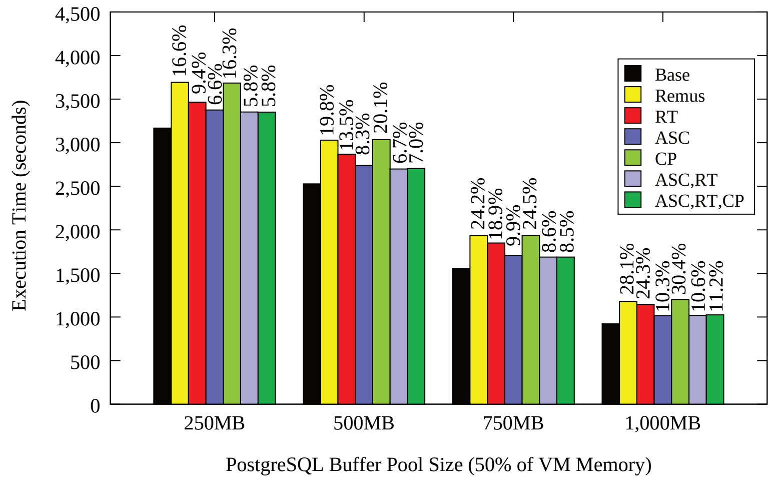


Figure 3.12: Effect of DB Buffer Pool Size on Performance (TPC-H)

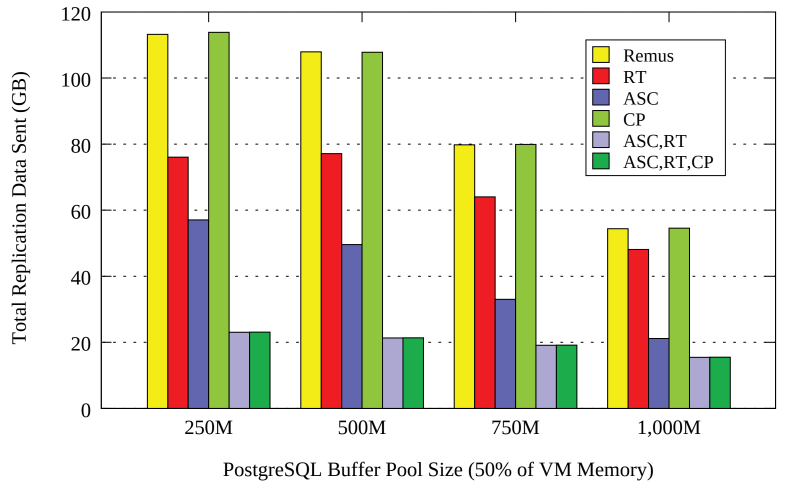


Figure 3.13: Effect of DB Buffer Pool Size on Amount of Data Transferred During RemusDB Checkpointing (TPC-H)

run and the power test run in each case, and repeat this for different RemusDB optimizations. To have reasonably realistic settings, we always configure the buffer pool to be 50% of the physical memory available to the VM. For example, for a 250MB buffer pool, we run the experiment in a 500MB VM and so on. Results are presented in Figure 3.12. The numbers on top of each bar show the relative overhead with respect to an unprotected VM for each buffer pool setting. We calculate this overhead as:

$$Overhead\ (\%) = \frac{X-B}{B} \times 100$$

where $B$ is the total execution time for an unprotected VM and $X$ is the total execution time for a protected VM with a specific RemusDB optimization.

Focusing on the results with a 250MB buffer pool in Figure 3.12, we see a 16.6% performance loss with unoptimized Remus. Optimized RemusDB with RT and ASC alone incurs only 9.4% and 6.6% overhead, respectively. The RemusDB memory optimizations (ASC, RT) when applied together result in an overhead of only 5.8%. As noted in the previous experiment, CP does not offer any performance benefit for TPC-H. We see the same trends across all buffer pool sizes. It can also be seen from the graph that the overhead of RemusDB increases with larger buffer pool (and VM memory) sizes. This is because the amount of work done by RemusDB to checkpoint and replicate changes to the backup VM is proportional to the amount of memory dirtied, and there is potential for dirtying more memory with larger buffer pool sizes. However, this overhead is within a reasonable 10% for all cases.

Another insight from Figure 3.12 is that the benefit of RT decreases with increasing buffer pool size. Since the database size is 2.3GB on disk (Table 3.2), with a smaller buffer pool size (250 and 500MB) only a small portion of the database fits in main memory, resulting in a lot of "paging" in the buffer pool. This high rate of paging (frequent disk reads) makes RT more useful. With larger buffer pool sizes, the paging rate decreases drastically and so does the benefit of RT, since the contents of the buffer pool become relatively static.

In Figure 3.13, we present the total amount of data transferred from the primary server to the backup server during checkpointing for the entire duration of the experiment. The different bars in Figure 3.13 correspond to the bars in Figure 3.12. With a 250MB buffer pool size, unoptimized Remus sends 113GB of data to the backup host while RemusDB with ASC and RT together sends 23GB, a saving of 90GB (or 80%). As we increase the buffer pool size, the network bandwidth savings for RemusDB also decrease for the same reasons explained above: with increasing buffer pool size the rate of memory dirtying decreases, and so do the benefits of memory optimizations, both in terms of total execution time and network savings. Recall that CP is not concerned with checkpoint size, and hence it has no effect on the amount of data transferred.

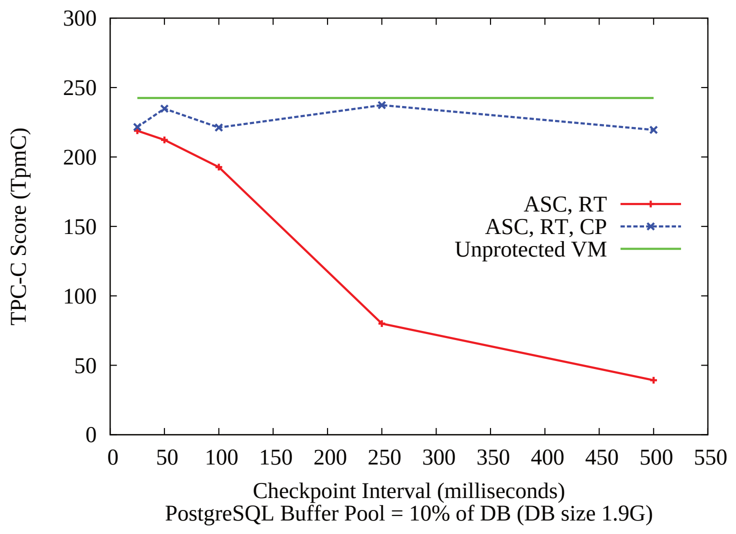


Figure 3.14: Effect of Checkpoint Interval on RemusDB (TPC-C)

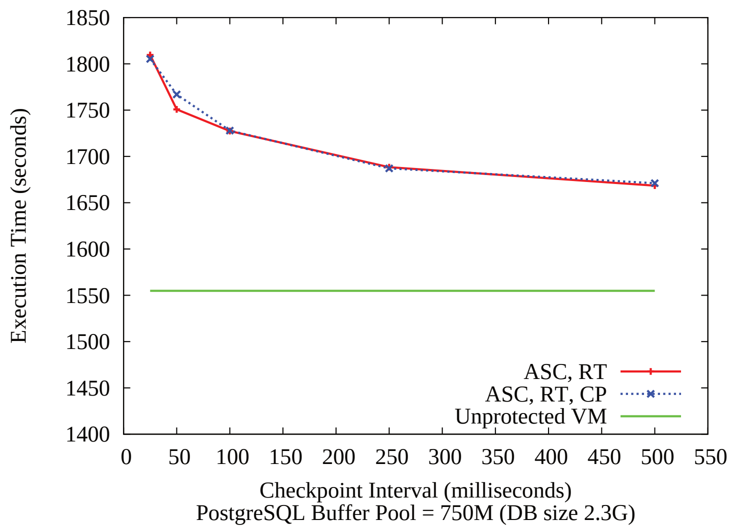


Figure 3.15: Effect of Checkpoint Interval on RemusDB (TPC-H)

### 3.7.6 Effects of RemusDB Checkpoint Interval

This experiment aims to explore the relationship between RemusDB's checkpoint interval (CPI) and the corresponding performance overhead. We conducted this experiment with TPC-C and TPC-H, which are representatives of two very different classes of workloads. We run each benchmark on Postgres, varying the CPI from 25ms to 500ms. Results are presented in Figures 3.14 and 3.15 for TPC-C and TPC-H, respectively. We vary CPI on the x-axis, and we show on the y-axis TpmC for TPC-C (higher is better) and total execution time for TPC-H (lower is better). The figures show how different CPI values affect RemusDB's performance when running with (ASC, RT) and with (ASC, RT, CP) combined, compared to an unprotected VM.

From the TPC-C results presented in Figure 3.14, we see that for (ASC, RT) TpmC drops significantly with increasing CPI, going from a relative overhead of 10% for 25ms to 84% for 500ms. This is to be expected because, as noted earlier, TPC-C is highly sensitive to network latency. Without RemusDB's network optimization (CP), every packet incurs a delay of $\frac{CPI}{2}$ milliseconds on average. With a benchmark like TPC-C where a lot of packet exchanges happen between clients and the DBMS during a typical benchmark run, this delay per packet results in low throughput and high transaction response times. When run with memory (ASC, RT) and network (CP) optimizations combined, RemusDB's performance is very close to that of unprotected VM, with a relative overhead $\leq 9\%$ for all CPIs.

On the other hand, the results of this experiment for TPC-H (Figure 3.15) present a very different story. In contrast to TPC-C, increasing CPI actually leads to reduced execution time for TPC-H. This is because TPC-H is not sensitive to network latency but is sensitive to the overhead of checkpointing, and a longer CPI means fewer checkpoints. The relative overhead goes from 14% for 25ms CPI to 7% for 500ms. We see a similar trend for both (ASC, RT) and (ASC, RT, CP) since CP does not help TPC-H (recall Figure 3.12).

There is an inherent trade-off between RemusDB's CPI, work lost on failure, and performance. Choosing a high CPI results in more lost state after a failover since all state generated during an epoch (between two consecutive checkpoints) will be lost, while choosing a low CPI results in a high runtime overhead during normal execution for certain types of workloads. This experiment shows how RemusDB's optimizations, and in particular the network optimization (CP), helps relax this trade-off for network sensitive workloads. For compute intensive workloads that are also insensitive to latency (e.g., TPC-H), choosing a higher CPI actually helps performance.

### 3.7.7 Effect of Database Size on RemusDB

In the last experiment in this section, we want to show how RemusDB scales with different database sizes. Results for the TPC-C benchmark on Postgres with varying

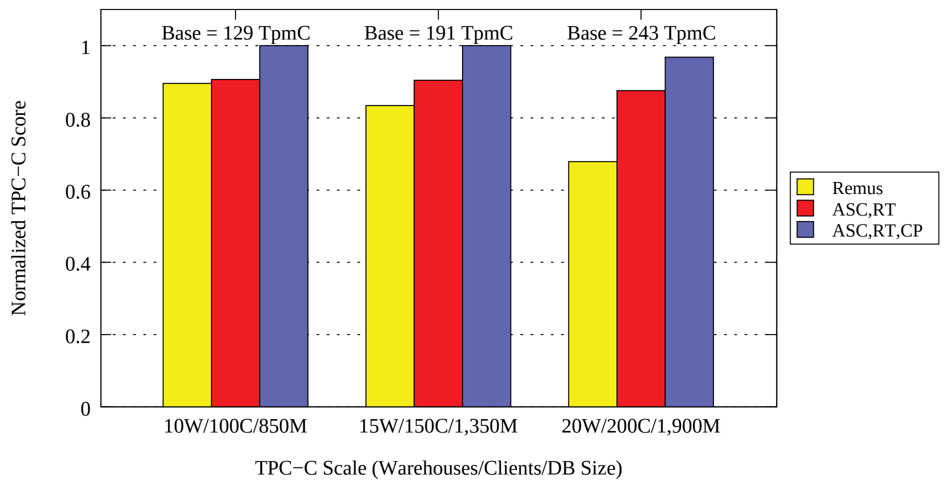


Figure 3.16: Effect of Database Size on RemusDB (TPC-C)

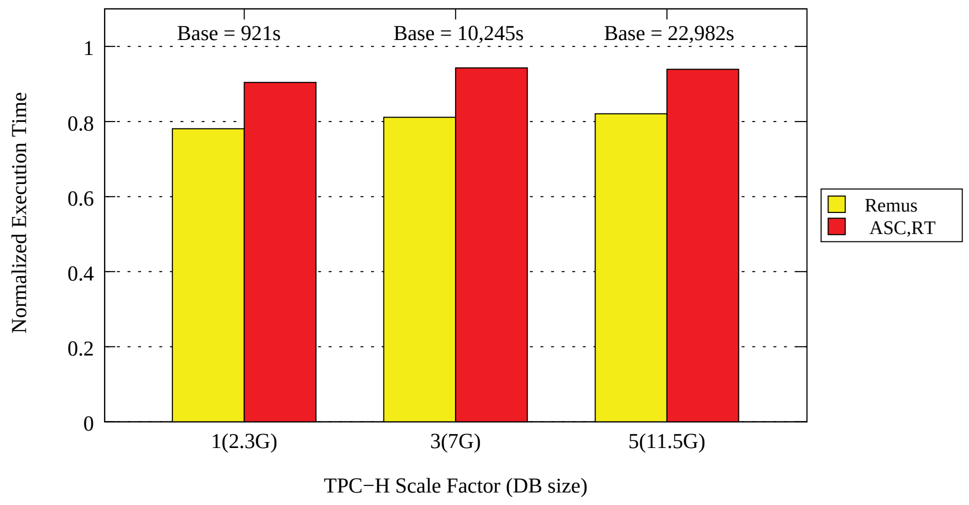


Figure 3.17: Effect of Database Size on RemusDB (TPC-H)

scales are presented in Figure 3.16. We use three different scales: (1) 10 warehouses, 100 clients, 850MB database; (2) 15 warehouses, 150 clients, 1350MB database; and (3 20 warehouses, 200 clients, 1900MB database. The Postgres buffer pool size is always 10% of the database size. As the size of the database grows, the relative overhead of unoptimized Remus increases considerably, going from 10% for 10 warehouses to 32% for 20 warehouses. RemusDB with memory optimizations (ASC, RT) incurs an overhead of 9%, 10%, and 12% for 10, 15, and 20 warehouses, respectively. RemusDB with memory and network optimizations (ASC, RT, CP) provides the best performance at all scales, with almost no overhead at the lower scales and only a 3% overhead in the worst case at 20 warehouses.

Results for TPC-H with scale factors 1, 3, and 5 are presented in Figure 3.17. Network optimization (CP) is not included in this figure since it does not benefit TPC-H. Unoptimized Remus incurs an overhead of 22%, 19%, and 18% for scale factor 1, 3, and 5, respectively. On the other hand, RemusDB with memory optimizations has an overhead of 10% for scale factor 1 and an overhead of 6% for both scale factors 3 and 5 – showing much better scalability.

## 3.8 Modeling RemusDB's Behavior for Effective Resource Scheduling

Having established the effectiveness of RemusDB as a transparent HA system, we now turn our attention to modeling its performance impact to help in the administration of a RemusDB deployment. RemusDB has been highly optimized for database workloads, but it still imposes some overhead during normal operation when compared to an unprotected database system, as shown in the previous section. Also, since RemusDB uses whole virtual machine replication to provide high availability, it requires sufficient network bandwidth for checkpointing. Both of these factors are important considerations for a database administrator (DBA) responsible for deploying RemusDB to protect database workloads, for example on a cluster. Given a database workload and a VM resource allocation, it would be ideal to know how much performance degradation this particular workload is expected to experience and how much network bandwidth will be required when using RemusDB protection *before* executing the workload. This can help a DBA maintain service level agreements (SLAs) for this given workload while effectively utilizing system resources (e.g., network bandwidth).

In this section, our goal is demonstrate how to build a model that takes workload type (OLTP, DSS), database size, buffer pool size, physical memory allocated to the VM, and Remus checkpointing interval (CPI) and possibly some other high level parameter(s) that a DBA can easily specify and then predict: (1) network bandwidth required for RemusDB checkpointing, and (2) performance degradation for a particular database workload when moving from an unprotected VM to a VM protected by RemusDB. Such a model can be used by either a DBA or an automatic

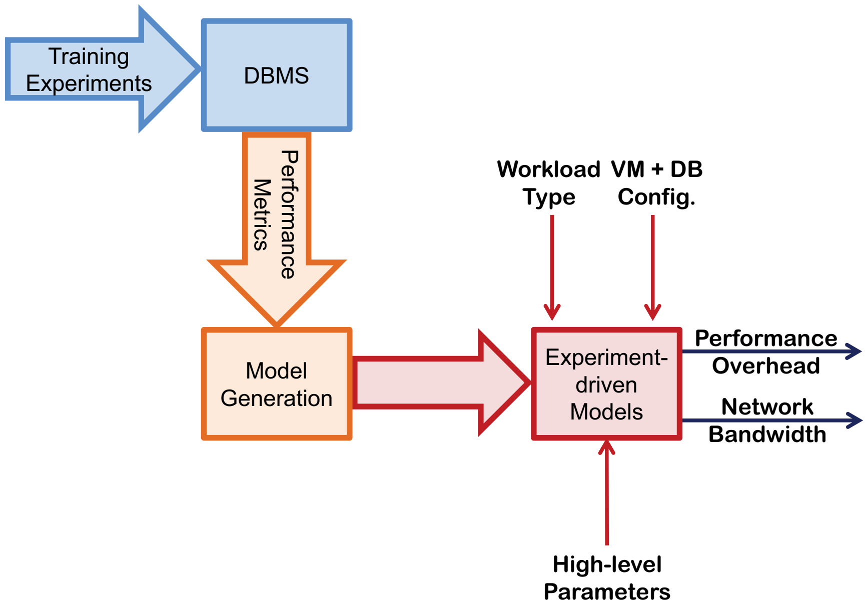


Figure 3.18: Model Building Workflow

tool to do initial placement of the VMs on a cluster depending on their network bandwidth requirements and also help meet SLAs by providing accurate estimates of the overhead of RemusDB protection. A DBA should be able to answer questions like: *"What is the cost of enabling protection for a certain VM in terms of network bandwidth requirements and performance overhead?"*. Alternatively, for VMs that are already protected a DBA can ask: *"What is the effect of changing physical memory allocation to the VM, DBMS BP size, and/or CPI on network bandwidth utilization and performance?"*. Our models aim to answer such questions.

In this work, we use *experiment-driven* modeling techniques [3, 4, 134] that have recently gained wide popularity due to their general applicability, ability to model complex interactions in workloads, and robustness. Experiment-driven modeling relies on: (1) designing and running experiments to collect training data, then (2) fitting statistical or machine learning models to this data. Figure 3.18 shows the overall modeling workflow. In subsequent sections, we explain how we apply these techniques to build models for predicting network bandwidth utilization and performance overhead of RemusDB protection. The focus of this section is on choosing the features to use for modeling and the structure of the statistical model to use (e.g., linear vs. non-linear). We illustrate these steps by building models for TPC-C as a representative of online transaction processing (OLTP) workloads, and TPC-H as a representative of decision support (DSS) workloads. The same approach we describe here can be applied for other OLTP and DSS workloads.

### 3.8.1 Collecting Training Data

The first step in experiment-driven modeling is to conduct experiments to collect training data. For the experiments in this section, we use a different infrastructure from the one used in Section 3.7. Our modeling experiments use two IBM blades each with two dual-core AMD Opteron 2216 HE processors running at 2.2GHz, 8GB RAM, two 1Gb ethernet cards, and a 45GB SCSI hard disk. Each machine runs Xen 3.2.3 with Linux Kernel 2.6.18 over Ubuntu Linux. One of the machines is designated as the active host and the other as the standby host. The protected VM is configured with two virtual CPUs and a 10GB disk image. Our training data is collected using TPC-C and TPC-H runs over Postgres using the benchmarking infrastructure described in Section 3.7.1. For building models we use R [142] – an open source statistical modeling software.

For TPC-C, we run experiments to collect training samples with fixed VM size and fixed buffer pool size for three different TPC-C scale factors: 10, 15, and 20 warehouses. In each case, the number of clients is ten times the number of warehouses, as required by the TPC-C specification. Each experiment is run with 5 minutes ramp-up time followed by a 25 minutes measurement interval. For each experiment, the VM is allocated 2GB RAM, and Postgres is configured with a 190MB buffer pool. Remus's CPI is varied from 25ms to 500ms in 25ms increments, resulting in a total of 20 experiments per scale factor or a total of 60 experiments for all three scale factors.

For TPC-H, we run experiments with two TPC-H database sizes (scale factor 1 and 3) at a fixed configuration with 1.5GB RAM for the VM, and a 750MB buffer pool size. Again, Remus's CPI is varied from 25ms to 500ms in 25ms increments, resulting in a total of 20 experiments per scale factor.

For the TPC-C experiments we measure total transaction throughput, and for the TPC-H experiments we measure total execution time. In addition, we also measure the total network bandwidth consumed. These variables are the response variables that our models will predict based on some features. One of the questions that we need to answer in this section is which features are best for modeling network bandwidth and performance overhead. To answer this question, we collect a large set of low level parameters showing CPU, memory, disk and network utilization using the Linux `sar` monitoring tool which is run from inside the VM. These resource utilization parameters are described in Table 3.3.

The result of our data collection experiments form our training and test samples. We randomly select 80% of the collected samples for training and the remaining 20% for testing our models. We then construct a model using the R statistical package. To evaluate the quality of each model we use the correlation coefficient ($R^2$), which measures how well the model fits the training data. We also use the average prediction error (APE) which measures how well the model performs on unseen test data. The APE is computed as:

$$AveragePredictionError(APE) = \frac{|Actual - Predicted|}{Actual} \times 100$$

| **Parameter** | **Description** |
|---|---|
| CPI | Remus's checkpointing interval (ms) |
| netBand | Total network bandwidth utilized (GB) |
| TpmC | Transactions per minute (TPC-C only) |
| ExecTime | Total execution time (s) (TPC-H only) |
| pginPS | Pages swapped in from disk per second (KB/s) |
| pgoutPS | Pages swapped out to disk per second (KB/s) |
| breadPS | Disk blocks read per second (blocks/s) |
| bwrtnPS | Disk blocks written per second (blocks/s) |
| rxpckPS | Network packets received per second (packets/s) |
| txpckPS | Network packets transmitted per second (packets/s) |
| cpuUtil | Average CPU utilization (%) |
| diskUtil | Average disk utilization (%) |

Table 3.3: Parameters Collected During Modeling Experiments

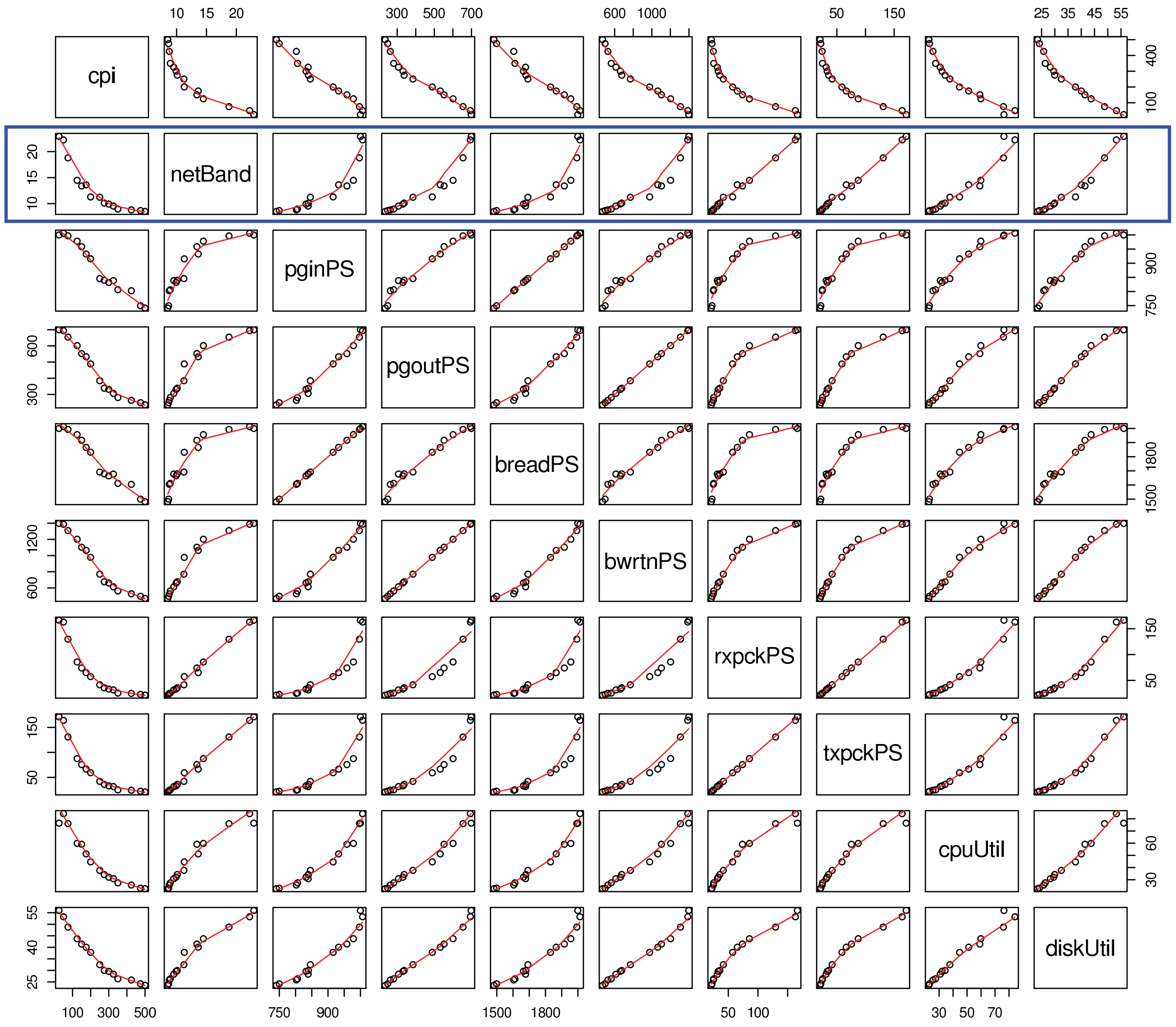
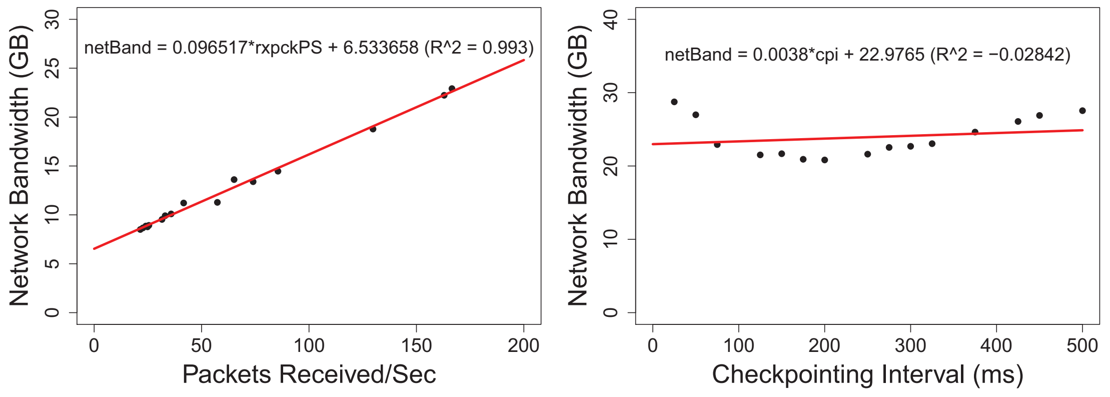


Figure 3.19: Pairwise Plot of All Parameters (TPC-C, 10W/100C)



(a) TPC-C (b) TPC-H

Figure 3.20: Linear Regression Models for Network Bandwidth

### 3.8.2 Modeling Network Bandwidth Utilization

We first focus on how to build regression models for predicting network bandwidth consumption for OLTP and DSS style workloads. The training data for these models is collected as described in the previous section. Our focus in this section is on the model structure to use (linear vs. non-linear) and on the features to base the modeling on. We will show that accurate models can easily be constructed using simple model structures and features that are easy to identify and easy for a DBA to collect.

#### Linear Regression Models

We first show the performance of linear regression models for TPC-C and TPC-H. Focusing on TPC-C, we show linear regression models for TPC-C with a scale of 10 warehouses and 100 clients. Linear models for higher scales are very similar to the ones presented here. We start with the question of which parameters to use as the input parameters to the model. From the training samples, we calculate a pairwise correlation matrix for all the parameters which showed that every parameter is either positively or negatively correlated with network bandwidth. The strongest positive correlation was found between network bandwidth (netBand) and packets received per second (rxpckPS). Figure 3.19 presents a pairwise plot with every parameter against every other parameter. The row showing graphs for the response variable, i.e., netBand, is highlighted. This graph shows that packets received (transmitted) per second has an almost linear relationship with network bandwidth. TPC-C is a closed loop benchmark, i.e., a TPC-C client submits a requests to the server and then waits for a response before issuing the next request. Furthermore, the requests and responses all have a similar size. Therefore, the number of requests (packets) received by the VM and the corresponding responses provides a relatively accurate measure of the load on the DBMS server. Since the network bandwidth consumed by the benchmark varies with the amount of work being done, the load

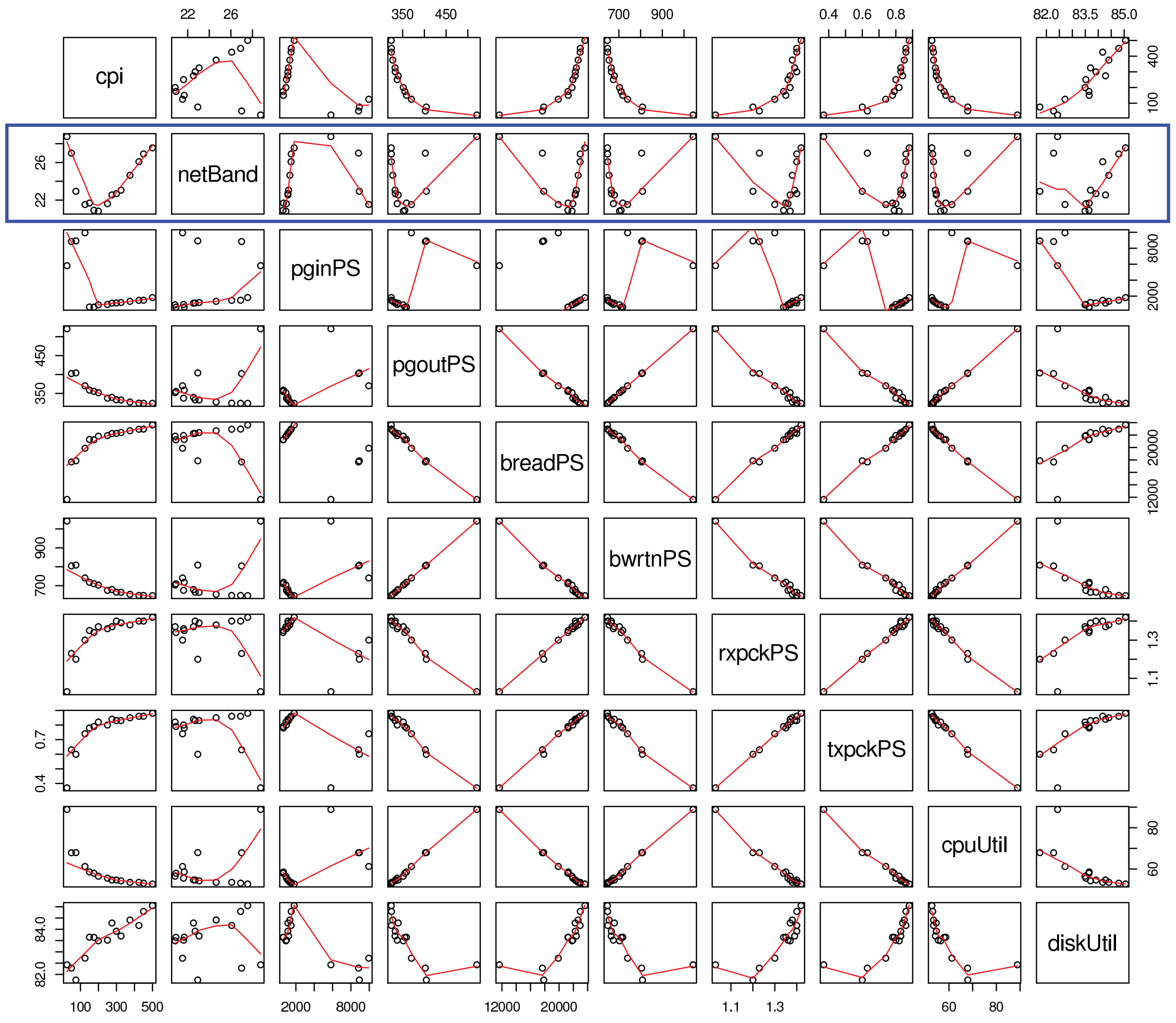


Figure 3.21: Pairwise Plot of All Parameters (TPC-H, SF=1)

on the DBMS server is a good indicator of the network bandwidth requirements. Building on these insights, we learn a simple linear regression model with network bandwidth (netBand) as the response variable and packets received per second (rxpckPS) as the predictor variable. Figure 3.20(a) shows a graphic representation of the linear model learned for TPC-C with 10 warehouses. The $R^2$ value for the model is quite high and the APE (not seen in the figure) is 4%, indicating the high accuracy of the model.

To make predictions for a target TPC-C style workload, our model requires the DBA to provide an estimate for only a single parameter, namely packets received per second (rxpckPS). In a typical TPC-C setup every SQL statement incurs a single round trip since the statement is submitted to the DBMS by the client (one outgoing packet), and the DBMS sends the response in return (one incoming packet). Since TPC-C consists of 5 transaction types each of which has a relatively deterministic behavior, this allows a DBA to easily estimate the number of SQL statements on average each type of transaction issues per second. The total rate of SQL statements issued per second for the workload can then simply be calculated by adding together the rate for each individual transaction type. Once the DBA has this number, it can be directly translated to number of packets received per second. We have experimentally verified the accuracy of such conversions.

Having looked at TPC-C, we now focus on TPC-H. Figure 3.21 shows a pairwise plot of every parameter against every other parameter for a TPC-H scale factor 1 database. It is evident from this graph that network bandwidth (netBand) does not depend linearly on any other parameter. However, there is a strong correlation between netBand and CPI. As the graph shows, netBand varies non-linearly with CPI so any linear model will be inaccurate. To show this we build a simple linear model using CPI as the predictor variable and netBand as the response variable. Figure 3.20(b) shows this model along with the model equation and $R^2$ correlation coefficient. The $R^2$ value is quite low, but we observe that the model still has a reasonable APE of 9%. The reason is that the variation in network bandwidth usage is not high for TPC-H. This means that the predicted value will not deviate from the actual value by much resulting in a low prediction error.

### Non-linear Models

In the previous section we saw that linear models work well for predicting network bandwidth for TPC-C, but not for TPC-H. In this section, we ask if more complex non-linear models can give us more accuracy. We start with a TPC-C workload with 10 warehouses and 100 clients. Models for the other two TPC-C scales are very similar to the ones that we present here. Figures 3.22(a) and 3.22(b) present polynomial and b-spline models, respectively, of degree 2, 3, and 9. Table 3.4 presents the $R^2$ and average prediction error (APE) for these models. It is evident that these models do not offer much advantage in terms of goodness of fit or prediction accuracy over the simpler linear model presented in Section 3.8.2. With a small number of training samples, these models also have a higher chance of over-fitting

| **Model** | $R^2$ | **APE (%)** |
|---|---|---|
| Polynomial (degree=2) | 0.99 | 4 |
| Polynomial (degree=3) | 0.99 | 4 |
| Polynomial (degree=9) | 0.99 | 5 |
| B-Spline (degree=2) | 0.99 | 4 |
| B-Spline (degree=3) | 0.99 | 3 |
| B-Spline (degree=9) | 0.99 | 6 |

Table 3.4: Accuracy of Non-linear Models for Network Bandwidth for TPC-C

| **Model** | $R^2$ | **APE (%)** |
|---|---|---|
| Polynomial (degree=2) | 0.7707 | 3 |
| **Polynomial (degree=3)** | **0.9536** | **2** |
| Polynomial (degree=9) | 0.9767 | 3 |
| B-Spline (degree=2) | 0.7584 | 3 |
| B-Spline (degree=3) | 0.9300 | 2 |
| B-Spline (degree=9) | 0.9519 | 2 |

Table 3.5: Accuracy of Non-linear Models for Network Bandwidth for TPC-H

the data. We therefore conclude that the additional complexity of more sophisticated models (polynomial, b-spline) is not worthwhile and henceforth consider only linear models for TPC-C style workloads.

We now present non-linear models for TPC-H using CPI as the predictor and netBand as the response variable. Figures 3.23(a) and 3.23(b) present polynomial and b-spline models respectively, of degree 2, 3, and 9 for TPC-H scale factor 1. Table 3.5 presents the correlation coefficient and average prediction error for each model. Either a polynomial or b-spline model with a degree $\geq 3$ provides an excellent fit and a very low prediction error. We see that the additional model complexity is warranted for TPC-H. Among the choice of models, we choose a polynomial model with degree 3 since it provides an excellent fit and the lowest average prediction error with less chance of overfitting the data as compared with higher degree polynomial (or b-spline) models.

### Sensitivity Analysis

The goal of this section is to show that our models for TPC-C and TPC-H presented in the previous section are robust to small changes in configuration, e.g., DBMS buffer pool size. We also show that if we make substantial changes to the configuration, e.g., VM size or buffer pool size, the average prediction error of our model rises sharply. This shows that the parameters of the model are indeed representing a particular state of the configuration. That is, while the model is robust

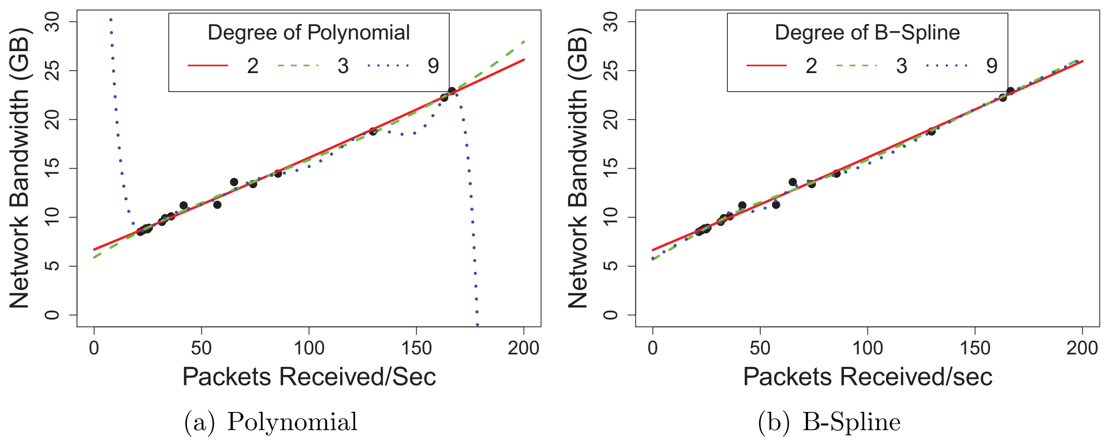


Figure 3.22: Non-linear Models for Network Bandwidth (TPC-C)

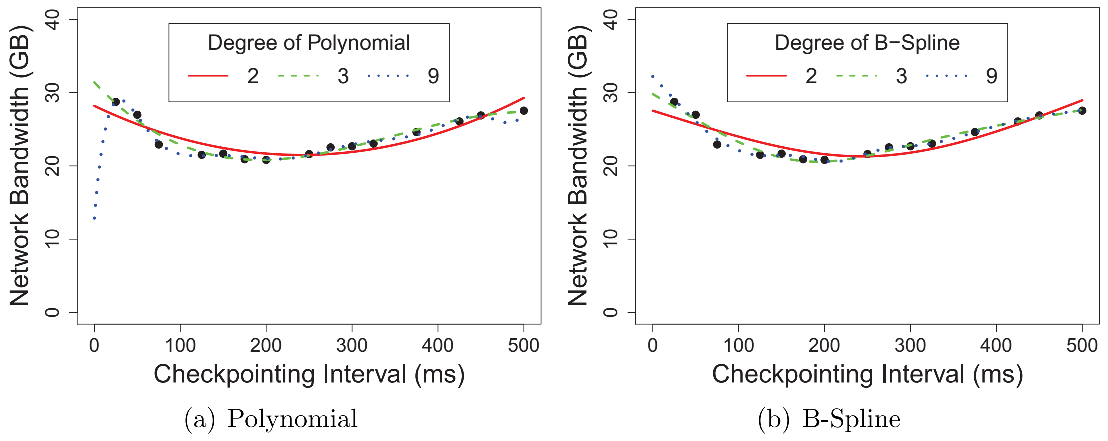


Figure 3.23: Non-linear Models for Network Bandwidth (TPC-H)

| **Training Configuration** | **Testing Configuration** | **APE(%)** |
|---|---|---|
| 20W/2.0G-VM/190M-BP | 20W/2.0G-VM/190M-BP | 2 |
| | 20W/2.0G-VM/256M-BP | 17 |
| | 20W/2.0G-VM/512M-BP | 33 |
| | 20W/1.5G-VM/190M-BP | 18 |
| | 20W/1.0G-VM/190M-BP | 23 |
| | 10W/2.0G-VM/190M-BP | 40 |

Table 3.6: Robustness of Models for Network Bandwidth for TPC-C

| **Training Configuration** | **Testing Configuration** | **APE(%)** |
|---|---|---|
| SF-3/1.5G-VM/750M-BP | SF-3/1.5G-VM/750M-BP | 3 |
| | SF-3/2.0G-VM/750M-BP | 7 |
| | SF-3/1.5G-VM/512M-BP | 10 |
| | SF-3/2.0G-VM/512M-BP | 10 |
| | SF-1/1.5G-VM/750M-BP | 354 |
| | SF-1/1.5G-VM/512M-BP | 356 |
| | SF-1/2.0G-VM/512M-BP | 396 |

Table 3.7: Robustness of Models for Network Bandwidth for TPC-H

to small changes in configuration, it is still sensitive enough to detect large changes in configuration.

Starting with TPC-C, we train a linear regression model presented in Section 3.8.2, for 20 warehouses and 200 clients at 2GB VM and a 190MB buffer pool and then test it at different configurations. Table 3.6 shows the default training configuration, the varying test configurations, and the average prediction error. These results show that even though our models are robust to reasonably small changes in configurations, DB, BP, and VM size are important parameters that significantly affect network bandwidth utilization and therefore need to be incorporated in the model learning process.

Next, we analyze the sensitivity of our best model for TPC-H, i.e., a polynomial model of degree 3, with respect to changes in configurations. We learn a model for a TPC-H scale factor 3 database with 1.5GB VM, and a 750MB buffer pool and test it against the same configuration as well as a set of varying configurations. Table 3.7 summarizes the results. These results show that our learned model for TPC-H is robust to changes in VM and BP sizes. However, when we change the scale factor from 3 to 1 while keeping all the other parameters constant, the average prediction error reaches 354%. Additional results for scale factor 1 also show a very high average prediction error. This shows that model learned at a certain scale factor can only make predictions effectively when tested for that same scale factor, i.e., scale factor has a very significant effect on the total network bandwidth utilization. In order to build a model that is able to make accurate predictions across different

DB sizes, we also have to learn the effects of DB size on total network bandwidth utilized.

**Unified Model**

We show in the previous section that the linear model learned for TPC-C is sensitive to changes in configuration parameters such as database size, VM size, and BP size. The goal of this section is to build a model that learns the effects of all these configuration parameters on network bandwidth utilization. Such a model is more powerful than a model which is learned for a specific configuration, but a more powerful model also requires more training data to capture the effects of all the configuration parameters. Experimental design becomes much more important in this case. With each additional parameter to learn, the number of experiments required grows exponentially.

In order to establish the importance of each input parameter, and therefore possibly reduce the number of total experiments required, we first use a $2^k$ full factorial experimental design, where $k$ is the number of input parameters to the model and each parameter takes two different values (high or low). We choose $k = 4$ with input parameters: CPI, DB size, VM size, and BP size, resulting in a total of 16 experiments ($2^4$). Analysis of the results shows that 83% of the total variation in the response variable (netBand) is explained by CPI alone, with CPI, DB size, VM size, and BP size together explaining 94% of the variation. This analysis also yields that second degree interactions between these parameters are not important and can therefore be ignored. This justifies an experimental design that chooses more levels for CPI since it explains the most variation in the response variable and fewer levels for other factors. This insight substantially reduces the total number of experiments a DBA needs to conduct for building such models.

Reasonable values of CPI for TPC-C style workloads can range from 25ms to 500ms, although the sensitivity of TPC-C to network latency means that lower values of CPI (25–100ms) will be chosen in practice. To collect training data, we conduct a total of 90 experiments ($5 \times 3 \times 3 \times 2$) using 5 different levels for CPI, 3 DB sizes, 3 VM sizes, and 2 different BP sizes. Each experiment takes about 20 minutes to finish. Note that we are still using a full factorial design, and there is room for further optimization by using partial factorial design, especially since our results with $2^k$ full factorial design show that the interactions of all factors are not meaningful, i.e., these interactions explain very little variation in the response variable. Another possibility is to use full factorial design and then choose a subset of the experimental space by using random sampling or Latin Hypercube Sampling [64]. In this thesis we focus on building models, leaving the refinement of experimental design to future work.

After collecting the training data from 90 experiments, we learn a unified linear regression model that takes as input the CPI, DB size, VM size, BP size, and packets received per second and predicts network bandwidth utilization. This model has a very high correlation coefficient, i.e., $R^2 = 0.9718$ and a very low average prediction error of 5%, showing the effectiveness of our learning approach. When equipped with such a model a DBA is able to accurately predict network bandwidth utilization for any combination of input parameters, making the job of setting up RemusDB protected virtual machines much easier.

Let us now focus on TPC-H. Section 3.8.2 shows that the model that we learn for TPC-H is relatively non-sensitive to changes in VM size and BP size but very sensitive to changes in database size (i.e., scale factor). We now present a model that learns the effects of these different parameters on network bandwidth utilization. Again for building such a model, we have to very carefully select a subset from the possible set of experiments that still captures all the important effects of different configuration parameters.

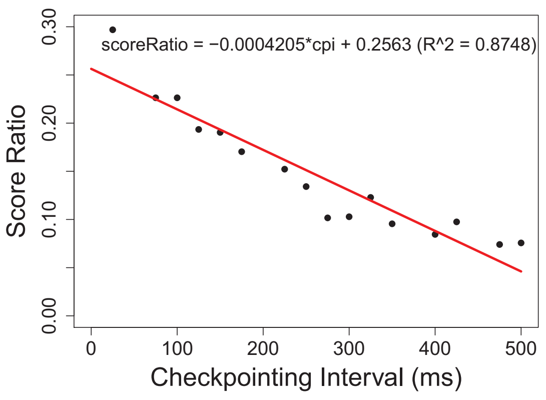
We perform an analysis similar to the one presented above for TPC-C. We first start with a $2^k$ full factorial experimental design with $k = 4$ and the same input parameters as for TPC-C. Analysis of the results of these experiments shows that CPI and database size explain 4% and 94% of variation in network bandwidth utilization, respectively. This demonstrates that a unified model must learn the effects of database size on network bandwidth utilization. This analysis also reveals that interactions between individual parameters are not significant which means that we do not need a full factorial experimental design. Furthermore, for a specific database size, 85% of the variation in network bandwidth is explained by CPI alone, with CPI, VM, and BP size together explaining 89% of the variation. These insights indicate that DB size is the most important parameter affecting network bandwidth. For a specific DB size, CPI has the most significant impact on network bandwidth. These insights are very useful in reducing the number of experiments required for training.

To collect training data for building a unified model, we conduct a total of 40 experiments using 2 DB sizes, 5 different levels for CPI, 2 VM sizes, and 2 BP sizes. We then learn a unified linear regression model that takes as input the CPI, DB, VM, and BP size and predicts network bandwidth utilization. The resulting model has a very high correlation coefficient of $R^2 = 0.9541$ and an acceptable average prediction error of 18%.

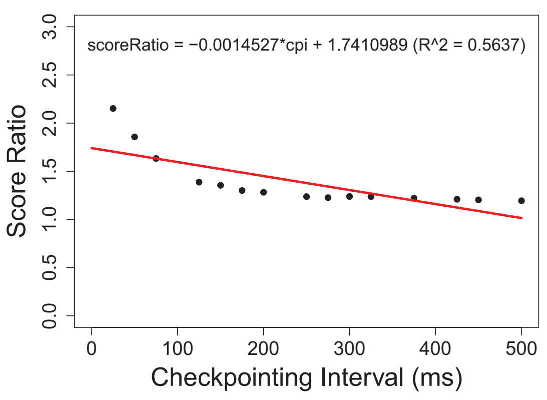
We conclude that simple models (linear or low-order polynomial) can be used to predict the network bandwidth requirement of RemusDB for OLTP and DSS workloads. We have also seen that these models can be generalized so that they take into account the effects of parameters such as CPI, database size, and VM size. The number of experiments needed to collect training data is not prohibitive since the higher order interactions between parameters are low.

### 3.8.3 Modeling Performance Degradation

In this section we present models that predict the performance degradation that a database workload is expected to incur when moving from an unprotected VM



(a) TPC-C



(b) TPC-H

Figure 3.24: Linear Regression Models for Performance Degradation

to a RemusDB protected VM. These models can be used by a DBA to check if any SLAs will be violated if a database is protected using RemusDB. We use the same learning strategy as that for building models to predict network bandwidth consumption.

**Linear Regression Models**

First focusing on TPC-C, we start with simple linear regression models for a fixed configuration, i.e, DB size, VM size, and BP size, and we vary only the CPI. We learn a model for 20 warehouses and 200 clients with 2GB VM and a 190MB BP. Again, models for other DB sizes are very similar. In this case, the response variable that we are trying to predict is the ratio of the TPC-C score (TpmC) under RemusDB to the TPC-C score under an unprotected VM, which we refer to as the *scoreRatio*. Our experimental data shows that CPI is a very good predictor of scoreRatio and the relationship between CPI and scoreRatio is almost linear. Therefore a simple linear regression model that uses CPI as the predictor variable and scoreRatio as the response variable, has a high correlation coefficient ($R^2$=0.8748) and a very low prediction error of 2%, as shown in Figure 3.24(a).

For TPC-H, we build a simple linear regression model with scale factor 3 using 1.5GB VM and 750MB BP, and we vary only the CPI. In this case, the scoreRatio is the the ratio of the total execution time under RemusDB to the total execution time of an unprotected VM. Our experimental data shows that CPI is a very good predictor of scoreRatio but the relationship between CPI and scoreRatio is not linear. As expected, a simple linear regression model that uses CPI as the predictor variable and scoreRatio as the response variable, has a low correlation coefficient ($R^2$=0.5637) but still relatively low prediction error of 8%. This low error is a result of low variation in scoreRatio. We present the resulting model in Figure 3.24(b).

| **Model** | $R^2$ | **APE (%)** |
|---|---|---|
| Polynomial (degree=2) | 0.97 | 0.8 |
| Polynomial (degree=3) | 0.97 | 0.9 |
| Polynomial (degree=9) | 0.95 | 1.5 |
| B-Spline (degree=2) | 0.97 | 0.8 |
| B-Spline (degree=3) | 0.97 | 0.9 |
| B-Spline (degree=9) | 0.97 | 0.5 |

Table 3.8: Accuracy of Non-linear Models for Performance Degradation for TPC-C

| **Model** | $R^2$ | **APE (%)** |
|---|---|---|
| Polynomial (degree=2) | 0.87 | 5.6 |
| Polynomial (degree=3) | 0.98 | 2.2 |
| Polynomial (degree=9) | 0.99 | 0.4 |
| B-Spline (degree=2) | 0.86 | 5.1 |
| B-Spline (degree=3) | 0.97 | 2.8 |
| B-Spline (degree=9) | 0.99 | 0.5 |

Table 3.9: Accuracy of Non-linear Models for Performance Degradation for TPC-H

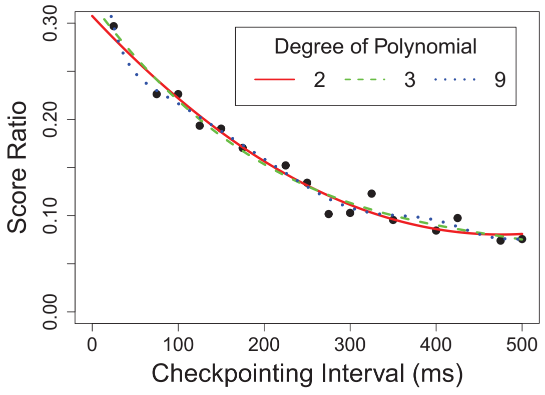
### Non-linear Models

We present non-linear models for predicting scoreRatio for TPC-C that try to improve on the linear model presented above. Figures 3.25(a) and 3.25(b) present polynomial and b-spline models, respectively, of degree 2, 3, and 9. Table 3.8 presents the correlation coefficient and average prediction error for each model. A polynomial or b-spline model of degree 2 has a very high correlation coefficient ($R^2$=0.97) and the lowest prediction error (0.8%). With higher degree models there is once again a chance of overfitting the data with a limited number of learning samples. Therefore we choose a polynomial model of degree 2 as the best performing model. Note that a b-spline model of degree 2 will work equally well.

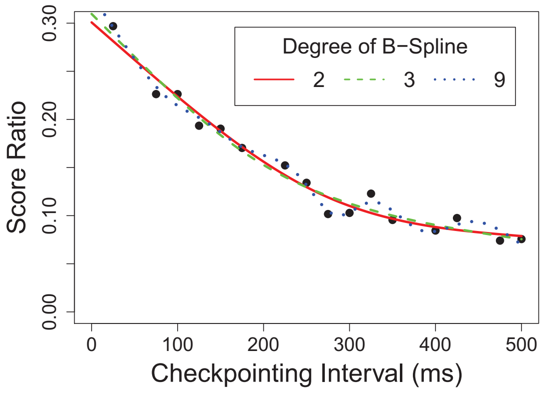
For TPC-H with scale factor 3, Figures 3.26(a) and 3.26(b) present polynomial and b-spline models, respectively, of degree 2, 3, and 9 for predicting scoreRatio. Table 3.9 presents the correlation coefficient and average prediction error for each model. A polynomial model of degree 3 has a very high correlation coefficient ($R^2$=0.98) and a low prediction error (2.2%). The polynomial and b-spline models of degree 9 have the highest correlation coefficient and the lowest prediction errors but they are very likely to overfit the data. Therefore, we choose a polynomial model of degree 2 as the best performing model.

### Unified Model

Linear and non-linear models for predicting scoreRatio that we presented in the previous sections work well but they have certain limitations. The biggest limitation

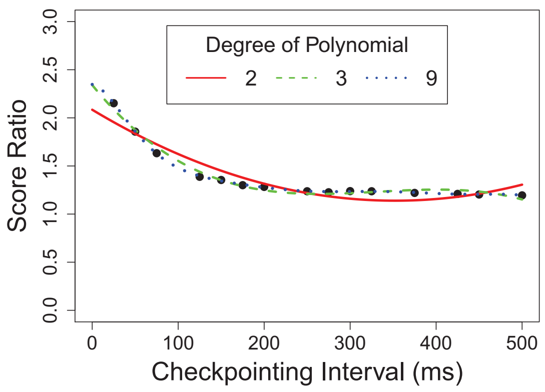


(a) Polynomial

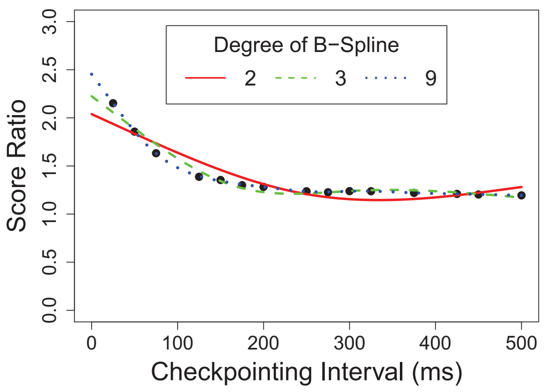


(b) B-Spline

Figure 3.25: Non-linear Models for Performance Degradation (TPC-C)



(a) Polynomial



(b) B-Spline

Figure 3.26: Non-linear Models for Performance Degradation (TPC-H)

is that these models were learned for a given DB, VM, and BP size and therefore will have to be re-learned for a target configuration that is significantly different from the training configuration. Our goal in this section is to build a unified model that is able to handle changes in these configuration parameters.

To learn a unified model for TPC-C, we use the same training data for the database running under RemusDB as in Section 3.8.2. In addition to those 90 experiments, we run 18 additional experiments with an unprotected VM using 3 DB sizes, 3 VM sizes, and 2 BP sizes (there is no CPI parameter for an unprotected VM) and calculate the scoreRatio for each RemusDB training experiment. From the previous section, we already know that a polynomial model of degree 2 in CPI works best for predicting scoreRatio. Using the new training data we learn an extended non-linear unified model that incorporates the additional parameters, i.e., DB, VM, and BP size. The resulting model has a correlation coefficient $R^2$ = 0.7869 and an average prediction error of 9%. This is a reasonable correlation coefficient and APE.

To learn a unified model for TPC-H that is able to predict performance degradation for any values of CPI, DB, VM, and BP size we use the same training data for the database running under RemusDB as in Section 3.8.2. In addition to those 40 experiments, we run 8 additional experiments with an unprotected VM using 2 DB sizes, 2 VM sizes, and 2 BP sizes and calculate the scoreRatio for each RemusDB training experiment. Using the new training data we learn an extended non-linear unified model that incorporates additional parameters, i.e., DB, VM, and BP size. The resulting model has a correlation coefficient $R^2 = 0.8763$ and an average prediction error of 6%.

## 3.9 Summary

In this chapter, we presented RemusDB, a system for providing simple transparent DBMS high availability at the virtual machine layer. RemusDB provides active-standby HA and relies on VM checkpointing to propagate state changes from the primary server to the backup server. It can make any DBMS highly available with little or no code changes and it imposes little performance overhead. We presented a detailed experimental evaluation that shows RemusDB's performance for various types of database workloads running under Postgres and MySQL. By using RemusDB's optimizations, we are able to provide very fast failover in $\leq$ 3 seconds while incurring very low performance overhead during normal operation. We also presented modeling techniques that can help a DBA build simple but effective models to predict the network bandwidth usage and performance degradation for RemusDB deployments. The unified model that we presented has an average prediction error of 5% and 9% for predicting network bandwidth usage and performance degradation, respectively, for TPC-C workload. For TPC-H workload, the unified model has an average prediction error of 18% and 6% for predicting network bandwidth usage and performance degradation, respectively. These results show that by using our experiment driven modeling approach, a DBA can build fairly accurate models for predicting network bandwidth usage and performance degradation for replication based HA systems like RemusDB.

Having investigated high availability, we now turn our attention to elastic scale-out, and present two approaches for achieving elasticity for a DBMS. We discuss one approach in Chapter 4 and the other in Chapter 5.

# Chapter 4

# Chimera: Elastic Scale-out and Load Balancing Through Data Sharing

In this chapter, we present design and implementation of a system that provides elastic scale-out and load balancing for database systems through data sharing. A detailed discussion about different techniques for scaling database systems can be found in Chapter 2 (Section 2.4).

## 4.1 Introduction

### 4.1.1 Different Database Architectures

A number of different architectures for parallel database systems have been explored in the evolution of current commercial database systems. The "big picture" categorization is the division into *shared nothing* and *data sharing*. We described shared nothing and data sharing systems in detail in Chapter 2, but to remind the reader, we begin by clearly defining these terms below. A more complete description of the relevant technical issues around database architectures is given in Section 4.2.

#### Shared Nothing DBMS

A shared nothing database server, whether a single-node system or a member of a multi-node cluster (recall that a *node* in this context is a server), has exclusive access to the data that it manages. Data for a node is usually stored on a disk that is locally attached to this node. A database is usually divided into mutually exclusive partitions and distributed among the nodes of a cluster, where each node is responsible for only its own partition(s). Only the node responsible for a partition can cache data from this partition. This node can hence provide concurrency control and recovery for the data that it manages without the need to coordinate with any other node, since this data can never be in the cache of any other node. This simplicity is a great advantage and in many settings leads to higher performance and scalability because there is no need for coordination among nodes. However, defining a good data partitioning is hard, partitions are usually statically defined, and repartitioning (which is typically necessary for elastic scale-out and load balancing) requires a reorganization of the entire database, and may require moving data between nodes.

### Data Sharing DBMS

In a data sharing DBMS, more than one node of a cluster can cache the same data (hence the name "data sharing") from the database stored on disk. All nodes in the cluster require access to the storage devices, which may be provided by using storage area networks (SANs) that enable shared access to disks. A data sharing DBMS cluster can respond to changing system load by deploying additional nodes for "hot" data to scale beyond single node performance, i.e., it can easily scale-out. As the working set and the hot spots in the data change, work can be moved between nodes for load balancing without the need to repartition the data. On the other hand, a data sharing system is more complex to implement, since it requires distributed concurrency control and recovery (CC&R) and distributed cache coherence, which impose additional complexity and load on the system.

The current database market is split between data sharing systems and shared nothing systems. In the data sharing camp are such long-lived commercial products as Oracle [90], and IBM mainframe DB2 [25]. In the shared nothing camp are the more recently implemented database systems such as IBM DB2/UDB [94] and Microsoft SQL Server [106].

In the TPC benchmarking wars, shared nothing systems usually win the price/performance battles, while data sharing systems have, with some frequency, won on peak performance. An interesting aspect is that when set up to run the benchmark, data sharing systems are frequently careful about partitioning **access** to the data to reduce the burden of CC&R and cache management.

Thus, to summarize, shared nothing database systems are easier to implement because they are less complex than data sharing systems [139]. For example, shared nothing systems do not face the problems of distributed locking and complex failure scenarios for recovery. Shared nothing systems can also be more scalable than data sharing systems since there is less need for coordination among the nodes. On the other hand, data sharing provides benefits such as *increased responsiveness to load imbalances*, which is highly desirable for dynamically changing workloads such as those in the cloud computing environments.

We would like a system with the flexibility of data sharing for load balancing and scale-out, but with the simplicity of shared nothing, as shown in Figure 4.1. An ideal

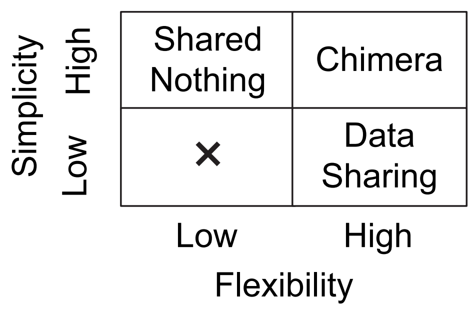


Figure 4.1: Chimera: Best of Both Worlds

solution would combine the advantages of data sharing with shared nothing without any of its disadvantages, e.g., without the added complexity that arises from the use of a distributed lock manager, global buffer manager, and complex logging and recovery mechanisms. In this chapter, we present the design and implementation of Chimera [100], a system that aims to achieve this objective by adding a data sharing "extension" to a shared nothing DBMS. Chimera is built using off-the-shelf components, providing effective scalability and load balancing with less than 2% overhead during normal operation.

### 4.1.2 Overview of Chimera

We enable sharing of databases among all the nodes of a cluster. Each node, acting as a **remote** node, can access a database hosted at any other node, the **local** node in the cluster, if the local node chooses to share that database. We start with a shared nothing cluster of low-cost desktop machines that can each host a stand-alone shared nothing DBMS, with one or more databases stored on their respective local disks. We carefully extend the DBMS with data sharing capability while minimizing the amount of new code and limiting the frequency of execution of this code. The techniques we use are applicable, in principle, to any traditional shared nothing database system. Our contribution is in (1) the design of a database architecture that is a hybrid of shared nothing and data sharing, (2) the implementation of Chimera – a system based on the hybrid architecture, and (3) the experimental confirmation of the validity of the approach.

"Full blown" data sharing can lead to both complexity and performance overhead. Thus, our goal is to provide carefully chosen data sharing that provides load balancing and scale-out benefits to a selected set of applications that are nonetheless of great practical importance since they occur frequently in modern data centers. Chimera provides the following:

1. **Load balancing at table granularity.** Chimera can offload the execution cost of database functionality in units of tables. A remote node can become the database server for accesses to one, several, or all tables. The node hosting the data simply serves pages belonging to the shared table(s) from its disk.

Our experiments confirm that this is an effective load balancing approach, not possible with a shared nothing system.

2. **Scale-out for read-mostly workloads.** Read-mostly workloads represent a wide range of Internet based workloads (e.g., many of the services hosted at Microsoft, Yahoo, or Google) and thus are important. Our experiments confirm effective scale-out for these kinds of workloads.

3. **Close to shared nothing simplicity.** A key design decision in Chimera is that only a single node can update a database at any time. The node doing the updating may change to enable load balancing. This decision means that the updating node has exclusive access to both the data and the transaction log. Thus, while working on tables, including those delegated to it by other nodes, a node executes as in a shared nothing system. Only in transitioning ownership need it be aware of data sharing, taking active steps to invalidate cached data at other nodes.

Our architectural goal is to minimize the number of system components impacted by our extensions. Most parts of the shared nothing database system are unaffected by the addition of data sharing. Chimera relies on three main software components: (1) the data is stored in a shared file system such as Common Internet File System (CIFS) [63] or Network File System (NFS) [129], so all nodes can access it, (2) a distributed lock manager such as Boxwood [92] or Chubby [28] provides ownership control, and (3) code added in the shared nothing DBMS coordinates data access and sharing among the nodes.

To enable sharing of database files at the file-system level, we store these files in a shared file system. This requires a very modest change to the DBMS and allows sharing of all data and metadata as a single package. Each node can receive, optimize, and compile queries using the metadata, as if the database were local. We present more details about file-system access in Chimera in Section 4.3.1.

Locking in Chimera goes on at two levels. We have global locks for ownership control, and local locks for transaction isolation. We do not modify the local locking mechanism at each DBMS node. We complement the normal locking protocol at each DBMS node with our global ownership locks, implemented by a generic distributed locking service as described above. Before any node in our system can gain access to a shared table, it first acquires a global read/write lock, after which it proceeds with the local locking protocol for transaction isolation, as it normally would without our extension. Local locks have lifetimes which are tied to transactions and follow standard locking protocols such as the two-phase locking (2PL) protocol. On the other hand, each node in the system can hold global ownership locks for extended periods of time, spanning multiple transactions. Furthermore, these locks are leased to each node, and are released only if the lease expires or there is a conflicting lock request. We present more details about our locking protocol in Section 4.3.2.

In order to ensure a globally consistent buffer cache, we have implemented a "selective cache invalidation" mechanism within the DBMS (Section 4.3.3). Another key design decision that greatly simplifies the data sharing mechanism is that we allow only a single writer of a shared database at any given time, even if each writer is modifying a different table. This allows us to avoid the complexities of logging and recovery protocols typical of a data sharing system. We can use the existing, unmodified logging and recovery protocol of a shared nothing DBMS. More details about logging and recovery in Chimera are presented in Section 4.3.4.

Another contribution of our work is that we present experiments demonstrating that Chimera enables load balancing for general workloads and scale-out for read-mostly workloads. Given the benefits of our approach and the simplicity of our extensions to "shared nothing" systems, this may gradually erase the hard boundary between data sharing and shared nothing.

The rest of this chapter is organized as follows. Section 4.2 describes data sharing and shared nothing database architectures. It details the complexities of the data sharing systems found in the database and systems literature. In Section 4.3, we provide an overview of our approach and talk about system components impacted when providing data sharing. Our implementation is described in some detail in Section 4.4, showing how our goals shaped our implementation approach. We report experimental results in Section 4.5 that demonstrate the value of the approach. Related work is discussed in Section 4.6. The final section summarizes the chapter.

## 4.2 DBMS Architectures

### 4.2.1 The Two Architectures

As we already mentioned in Chapter 2, a shared nothing database system (SN) deals with an inherently simpler situation than does a data sharing system (DS). A SN database system provides concurrency control and recovery where it has exclusive ownership of, and complete information about, the data it manages. A DS system must coordinate access to data that can change in flight, and provide recovery for the data that it manages even as multiple nodes are updating the same data and committing transactions.

Some of the issues in database systems apply also to distributed or cluster file systems, e.g., [13, 45, 55, 143] to cite a few early examples. Each metadata operation initiated by the file system is typically treated as an atomic operation requiring locking and logging techniques analogous to database kernels. However, distributed file systems (DFS) do not typically allow arbitrary client transactions.

The complexity of the task that DS systems face makes it inevitable that they are more complex, with longer execution paths. DS systems must provide distributed concurrency control for access to the data and distributed cache coherence as well. More complex and costly handling of system crashes is also typical. To provide recovery, a node in a DS system cannot simply write to a private log. Rather, it either needs a shared log or must ensure that its log be correctly merged with other logs. In the next two subsections, we delve more deeply into the challenges that DS systems face in accessing "shared data" from multiple nodes in a cluster of such nodes.

### 4.2.2 Concurrency Control

When data is shared by nodes of a cluster and lock based concurrency control is used, the locks on shared data must be visible to all nodes that can access this data. These are called distributed (or global) locks and the lock manager involved is usually called a distributed lock manager (DLM). Any node accessing data must request an appropriate distributed lock on the data from the DLM, which is costly. The DEC Rdb [89] experience was that distributed locks required almost 50% of the code path required for executing an I/O, and also introduced significant latency.

Achieving good performance in Rdb required a relentless effort to reduce the number of distributed lock requests. These locks arise more frequently than one might expect. Because a DS system has a distributed cache, costly distributed locks, not cheap latches, are required to protect pages and keep the cache coherent. In Rdb, locks are held by software processes, with each transaction assigned to a single process. Rdb 7.0 introduced lock retention across transactions to avoid extra trips to the DLM to re-acquire locks for the next transaction. The expectation was that often (and especially with effective application partitioning) the lock would be needed again soon by another transaction executing in the same process. A similar technique, called leases [59], is used in distributed file/storage systems [28, 45, 143, 131].

When a node of a DS cluster crashes and locks are not persistent, its distributed locks are lost. A very complex recovery process is needed to sort things out (see below). Making the locks persistent avoids this, but introduces extra cost. Writing the locks to disk results in very poor performance. IBM mainframe DB2, when executing on a cluster that includes a "Sysplex", specialized hardware that makes distributed locks stable, improves performance though at a significant cost. Another way to make locks persistent is to build the DLM as a replicated state machine [81], an approach found in some distributed systems [28, 92, 143]. This requires no special hardware support and the communication and disk bandwidth overhead may be acceptable.

### 4.2.3 Recovery

Data sharing systems can use a variety of techniques to ensure that locks persist, as indicated above, but they all add to the code path during normal execution. When distributed locks are volatile, execution cost is reduced but a node crash loses locks. This “partial” failure undermines a claimed advantage for data sharing clusters, i.e., that single node failures do not necessarily make data inaccessible if multiple nodes can access each disk. (Such “shared disk” hardware is sometimes confused with data sharing database systems because DS systems have frequently been built using shared disks.)

To cope with lost locks, Rdb instituted a “database freeze” [123]. This makes data briefly unavailable. All transactions are aborted, both on crashed nodes and on still up nodes. Transaction undo is guaranteed because sufficient locks are taken during forward execution so that no additional locks (which may block) are needed during undo. Undo returns the system to a state where no transactions are active and it is known that no locks are held. At that point, access to data can resume (and very quickly at the nodes that did not crash) while the crashed node itself is recovered. This is both complicated, and briefly turns a partial failure into at least a brief complete interruption of service. Some transaction aborts might be avoided were Rdb to track, outside of the DLM, the locks held by transactions. But it does not.

Another complication is how logging is handled. Two different ways to approach this problem have been pursued.

1. Provide a shared log with all nodes posting log records to the shared log. Coordinating high performance shared logging is subtle. One cannot afford to access this log at every update as this involves interprocess communication. Rdb’s shared log manager hands out log blocks to processes, which pass them back when they are filled. At a crash, extra effort is needed to determine the exact end of the log since the log tail typically had a number of noncontiguous blocks.

2. Provide separate logs for each node. This changes the problem to how to merge the logs during recovery. With separate logs, the usual log sequence number (LSN) ordering no longer is guaranteed to match the time-sequence of updates. This means that the notion of LSNs and how they are assigned to log records and pages needs to be modified [87, 105, 143] so that, at recovery time, the separate logs can be effectively merged. Finally, to make it possible for a node to recover without involving other nodes in its recovery, one needs to make sure that recovery for any single data item is confined to a single log. When the node caching a data item changes, an up-to-date copy of the data must be on stable storage (disk) before the data is cached at the new node. This is coordinated via the DLM and often requires a disk write to flush dirty data from the old node, adding to the performance overhead of logging.

## 4.3 Simplified Data Sharing

As we note in the previous section, providing full-fledged data sharing is hard owing to the complexity of CC&R and cache management. However, we note

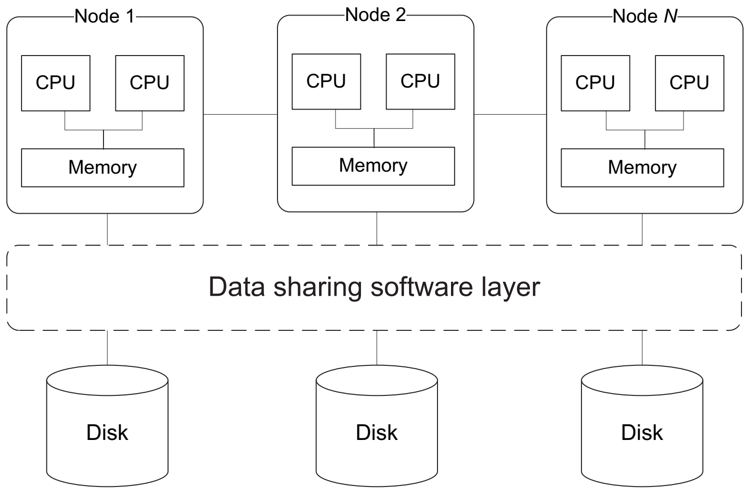


Figure 4.2: Shared Nothing Architecture with Data Sharing Software Layer

that both shared nothing and data sharing systems can be implemented using the same shared nothing hardware architecture. Only the system software needs to be different. A key challenge of a data sharing system is to maintain data integrity without sacrificing performance. Careful design is needed to keep the overhead low. Although software-only techniques for implementing shared disk systems have been proposed (e.g., [84]), specialized hardware has frequently been used to implement data sharing systems efficiently. This leads to specialized designs and increased costs since this hardware is not commodity and is priced accordingly.

In this work, we take a stand-alone SN database architecture running on commodity hardware and change only the software to implement a system that exhibits many of the characteristics of a data sharing system. We illustrate the resulting architecture for such an approach in Figure 4.2. The added *data sharing software layer*, which is carefully partitioned from the remainder of the system, distinguishes this architecture from a shared nothing architecture. We present the design of this layer later in this section, while the implementation details are presented in Section 4.4.

We exploit three fundamental simplifications in designing our solution.

1. All metadata and associated structures of the database are accessible by each node of a cluster as if it owned the database.

2. Only one node at a time can update a database, while multiple nodes can share read access.

3. Nodes have ownership locks on tables that are separate from transactional locking in order to share data access.

Sharing at table granularity limits the number of ownership locks and the frequency of lock requests. However, it limits data sharing to the scenario that while one node may be updating a table, other nodes can be sharing read access to the other tables of a database. This is definitely not as powerful as a full-fledged DS system, but it is quite useful, nevertheless. Moreover, this sharing is enabled at a very modest implementation cost. Importantly, load balancing by doing a scale-out of data from a "local node" to a "remote node" can be accomplished without moving the data from the local node's disk to the remote node's disk, and can easily be done on a time scale of minutes, not hours. The following sections provide an overview of different components required to build our data sharing system. We show how one can use off-the-shelf components and make minor changes to the DBMS to build a hybrid system such as Chimera.

### 4.3.1 File-system Access

For data sharing, the database files need to be accessible from multiple nodes. Data sharing systems traditionally solve this problem by using SANs, or shared disks. We posit that software-only techniques, for example, distributed or network file systems, can be used for storing data and can provide adequate performance for a data sharing system. We can use any network or distributed file system (see [63, 129, 143] for some representative examples) to share access to the database files from multiple nodes. Allowing a DBMS to use a DFS or NFS to store database files requires simple changes to the DBMS. In Section 4.4 we present details of these changes for one particular DBMS. Similar changes can be made to any existing shared nothing DBMS.

By sharing the database files (using a DFS or NFS), we encapsulate all data and metadata into a single package (design consideration (1)). When we provide access to the data, we simultaneously provide access to the metadata. This permits the remote (sharing) node to receive queries, compile, and optimize them as if the database were local. The local node for the database acts as a disk controller for the remote nodes, reading its disk to provide database pages requested by the remote nodes. The result of this activity is invisible to the local node database system. Each remote node caches data as required by the queries it executes.

### 4.3.2 Distributed Locking

We need some form of global locking with a data sharing system in order to synchronize access to common data. It is very important to keep the overhead of locking low. In the past, this has led to a highly complex distributed lock manager usually paired with specialized hardware for maximum performance. Multi-version concurrency control (MVCC) is an alternative to locking based concurrency control and is typically used to implement snapshot isolation (SI). SI has been shown to work well in a data sharing system such as Oracle RAC [110], and would obviate the need for a distributed lock manager. In this work, we assume that locking based concurrency control is used for data sharing, which is often the case. Distributed locking does not scale well, but it is important to note that we do not need a full-fledged distributed lock manager for our data sharing solution. We simply need a lock manager that is able to handle lock requests on remote resources (e.g., in our case, a remote table) in addition to local resources. Such a “global” lock manager dealing with ownership locking is far less complex than a full-fledged distributed lock manager more typical of data sharing systems.

Given the widespread deployment of Internet scale services, several highly scalable distributed coordination services have been proposed [28, 69, 92]. Chubby [28] offers a coarse-grained locking service that is able to provide strong synchronization guarantees for distributed applications. Chubby is used by Google File System [53] and Bigtable [31] to coordinate between master servers and their clients as well as to store metadata. ZooKeeper [69] offers a simple and efficient wait-free coordination service for large scale distributed applications. ZooKeeper is not a lock manager but the basic coordination primitives that it provides can be used to implement a distributed lock manager. Boxwood [92] implements a fault-tolerant distributed locking service that consists of a single master server and multiple slave servers. It implements leases which are cached at each lock client. A lease on a lock is released only if a conflicting request for the same lock is issued by another lock client. Failures of lock servers and/or clients are detected and safety is guaranteed by the use of a recovery procedure.

Our system exploits a global lock manager adopted from Boxwood [92]. While distributed locking at a fine granularity for resources and lock owners has proven to be complex and costly, simplified global locking at a coarse granularity has been successfully exploited [28, 69, 92], as mentioned above. Locking in our system goes on at two levels, globally for ownership sharing of tables among nodes and locally for transactional isolation within each node. This global/local distinction and its advantages were highlighted in [88]. We call the duration of ownership an *ownership period*. It is important to note that once an ownership period is activated, no changes to the shared nothing system are required to coordinate access among transaction executions within the period.

Locking granularity for ownership sharing among nodes is at the table level. This is a practical limitation designed to reduce the number of locks and hence locking overhead. We assume that reader and writer nodes (not transactions) hold ownership table locks for durations involving multiple transactions. Recall that we are using data sharing as a way to do load balancing and scale-out in a low cost manner. Our data sharing is not intended for continuous tightly coupled execution, but rather to inexpensively respond to load changes.

### 4.3.3 Distributed Buffer Management

Since data from the database can be cached at multiple nodes simultaneously, we need a distributed buffer manager to maintain consistency in the face of updates to the database. A surprisingly modest extension to the existing buffer manager in any shared nothing database system today can provide the capability of "selective cache invalidation" which is sufficient to maintain global buffer consistency.

We limit the impact on readers of a writer updating a table via selective cache invalidation. Readers cannot access pages of a table while it is being updated by the writer. But pages of such a table can remain cached. Each writer, at the point when it releases its exclusive lock on a table does two things:

1. Flushes updated pages to disk so that readers, when re-accessing changed pages, find the latest page versions.
2. Sends invalidation notices to all reader nodes, notifying them of the changed pages. Readers invalidate these pages by removing them from their caches, requiring them to re-read these pages from disk when they are accessed again.

### 4.3.4 Logging and Recovery

Updates to the database can be executed from any node in the cluster so distributed logging and recovery mechanisms are needed for a data sharing system as detailed in Section 4.2. However, by making careful design decisions we can work with existing unmodified shared nothing logging and recovery protocols.

By providing exclusive access on the database for updates (design consideration (2)), we avoid dealing with multiple logs or with complex and costly interactions on a shared log. An updating node treats the log as if it were a local log. Hence, there are no logging changes required in the shared nothing system. Ownership of the log changes hands when exclusive access to the database for updates changes hands. When ownership of the log is transferred, the pages changed by the node giving up control are all "checkpointed". This kind of protocol is not essential, but it simplifies the implementation. This restriction also simplifies how we deal with system crashes. (1) Only one node needs to be involved in recovery. (2) Reader nodes can, as long as the local node remains up, continue to execute queries on their read-only tables, unaffected by an updater crash. There is no ambiguity about which data is unavailable: it is the tables locked by the updater.

## 4.4 Implementation

In this section we present the implementation of Chimera using the techniques presented in Section 4.3. Our implementation is an extension to an existing shared nothing DBMS, Microsoft SQL Server. We use a cluster of PC-based commodity machines each running a stand-alone instance of SQL Server hosting one or more databases on local storage. Chimera allows a **remote** SQL Server instance to access a database hosted on a **local** instance. We share the database files among the nodes using CIFS – a network file sharing protocol [63]. CIFS is a commonly

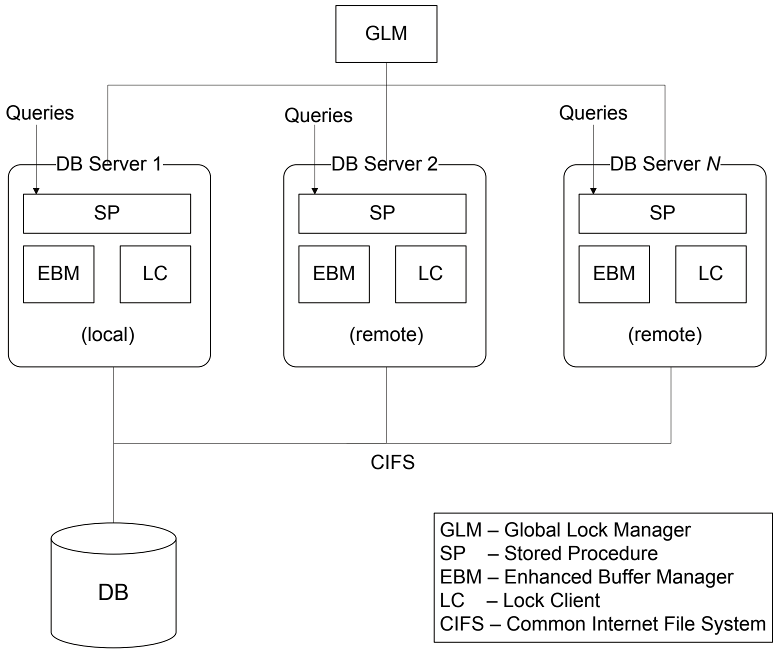


Figure 4.3: Chimera System Architecture

used protocol for file sharing over the network, any other similar protocol like NFS may be used as well. This enables multiple nodes to access the same database concurrently over the network, without physical data movement or repartitioning. Since this is not the default behavior, a minor change in the server allows the same database to be accessed remotely by multiple SQL Server instances. This setting instantly gives us a read-only data sharing cluster. However, in order to enable write data sharing, we need locks to synchronize access to common data. In addition, we also need to implement mechanisms to maintain global buffer consistency. Chimera achieves this by implementing an enhanced buffer manager (EBM) and a global lock manager (GLM) with corresponding local lock clients (LC). A stored procedure (SP) coordinates locking and buffer management while executing user queries against shared tables. A high-level architecture of an $N$-node data sharing cluster running Chimera is shown in Figure 4.3.

### 4.4.1 Stored Procedure

We implement most of the required changes in a user defined stored procedure that can be invoked like a standard stored procedure. An instance of this stored procedure is installed at all nodes of the cluster and accepts queries that need to be executed against one of the shared tables. Based on whether it is a read or an update query, the procedure does appropriate locking and buffer management to execute the query. As part of metadata, we maintain the information about other nodes in the system and the databases (or tables) that they host in a SQL table stored locally on each node. Maintaining this information can pose a scalability

**Algorithm 1**: Read Sequence

```
Acquire S lock on the abstract resource "ServerName.DBName.TableName".
if lock granted then
    execute the read query.
else
    retry lock request.
Release S lock.
```

challenge for a fairly large cluster. However, this problem has been studied in the domain of distributed file systems and we believe that the same techniques can be applied here. For the scale of systems that we deal with (a few tens of nodes), any of these techniques are sufficient.

### 4.4.2 Enhanced Buffer Manager

We extend the existing buffer manager in the database engine to implement a cross-node *selective cache invalidation* scheme necessary to maintain global buffer consistency. After an update period, dirty pages are forced to disk at the updating node. The updating node also captures a list of dirty pages and broadcasts an invalidation message to all the readers instructing them to evict these pages from their buffer pool. Only after these pages are evicted at all nodes will the updating node release the write locks, allowing other readers of the same table to proceed. After an update, readers are required to re-read the updated pages from disk. Note that the cache invalidation granularity is at the page-level which is finer than our table-level sharing granularity. This means that after an update, only the pages that got dirtied will be evicted. All other pages belonging to the shared table remain in-cache and immediately accessible.

### 4.4.3 Global Lock Manager

Chimera uses a global lock manager with local lock clients integrated with each database server instance. Our extended locking scheme is adopted from Boxwood [92] and complements the usual shared nothing locking protocol at each node when the data sharing capability is used. Instead of locking a physical entity (e.g., a data or an index page), the Boxwood lock server grants or releases a lock in shared (S) or exclusive (X) mode on an abstract resource represented by an arbitrary string. The use of Boxwood lock mechanism is ideal for Chimera as it does not require any changes to the existing lock manager. Also note that any other distributed coordination service such as Chubby [28] or ZooKeeper [69] would work equally well.

**Algorithm 2**: Write Sequence

```
Acquire X lock on the abstract resource "ServerName.DBName".
if lock granted then
    Acquire X lock on the abstract resource
    "ServerName.DBName.TableName".
    if lock granted then
        execute write request
    else
        retry X lock.
else
    retry X lock.
Flush updated pages to disk.
Do selective cache invalidation.
Release both X locks.
```

**Locking Protocol**

The locking protocol that we use is as follows: Assuming that a local server has enabled sharing on a table named `TableName` in a database named `DBName`, a reader always acquires a global shared (S) lock on an abstract resource created as "`ServerName.DBName.TableName`", where `ServerName` is the name of the local server hosting the database. This naming scheme for abstract resources ensures that shared table names are unique among all nodes of the cluster. On the other hand, writers of a shared table acquire global exclusive (X) locks on resources "`ServerName.DBName`" and "`ServerName.DBName.TableName`" to block other writers of the database and other readers of the table, respectively. This two level locking scheme for writes prevents multiple writers of a database. Readers and writers of the same table always block each other. Furthermore, a lock is cached at the local lock client after it is released until the GLM revokes it, which can happen when there is a conflicting lock request for the same resource.

The pseudocode for the read and write sequence implemented in the stored procedure is summarized in Algorithm 1 and Algorithm 2, respectively.

### 4.4.4 Recovery Issues

Section 4.2 describes the complexity of recovery in a data sharing system. The combination of our implementation technique and our restrictions on concurrency permits us to avoid that complexity. Our ownership locking protocol permits only a single writer at a time. That writer has access to the unique log associated with the database for the duration of its update period. The single writer simply uses LSNs associated with that log and the unmodified recovery protocol of the shared nothing DBMS. This is the "common shared log" approach implemented in Chimera in a simple and low-overhead way.

## 4.5 Experimental Evaluation

This section describes our experimental goals, the setup of physical machines used for experiments, and a discussion of the experimental results in detail.

### 4.5.1 Experimental Goals

We focus our attention on a small number of issues for which we want to provide quantitative results.

- Scalability: A key advantage of Chimera is that work can be spread among a set of nodes during periods of high load by doing elastic scale-out. We want to understand the limits of scalability of Chimera and the impact on shared nothing performance.
- Overhead: Chimera can introduce overhead arising from several sources: (1) The cost of executing database functionality from a remote node is likely to be more than that of local execution of the same functionality. (2) The latency of execution depends on the effectiveness of the cache. At startup, a cold cache results in significant overhead due to cache misses, while we expect steady state behavior, with warmed up caches, to be acceptable. (3) The code changes we introduce into the common case execution path of a shared nothing DBMS adds cost when used.
- Update Cost: Writes introduce a penalty in our system as they impact caching behavior. We wish to quantify this write penalty and also understand the read/write mix for which we can provide acceptable performance.

### 4.5.2 Experimental Setup

We use a 16 node cluster to conduct our experiments. Each node in the cluster has identical hardware and software configurations. Each node has two AMD Opteron CPUs running at 2.0GHz and 8GB RAM. The nodes run Windows Server 2008 Enterprise with SP2, and Microsoft SQL Server 2008 (SP1) with our modifications.

We use the TPC-H benchmark [146] for our experimental evaluation. We use a TPC-H database with scale factor 1. The total size of the database on disk including indexes is approximately 2GB. We configure SQL Server's buffer pool size to 2GB so that when the cache is warm the entire database can fit in main memory.

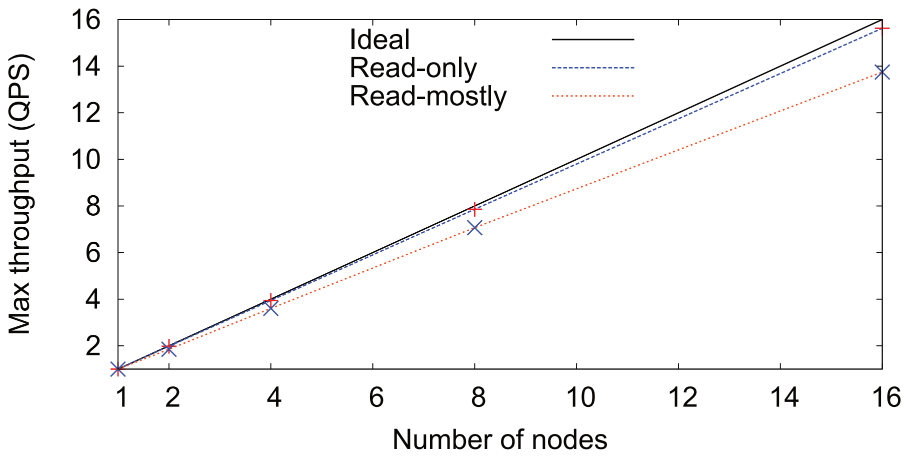


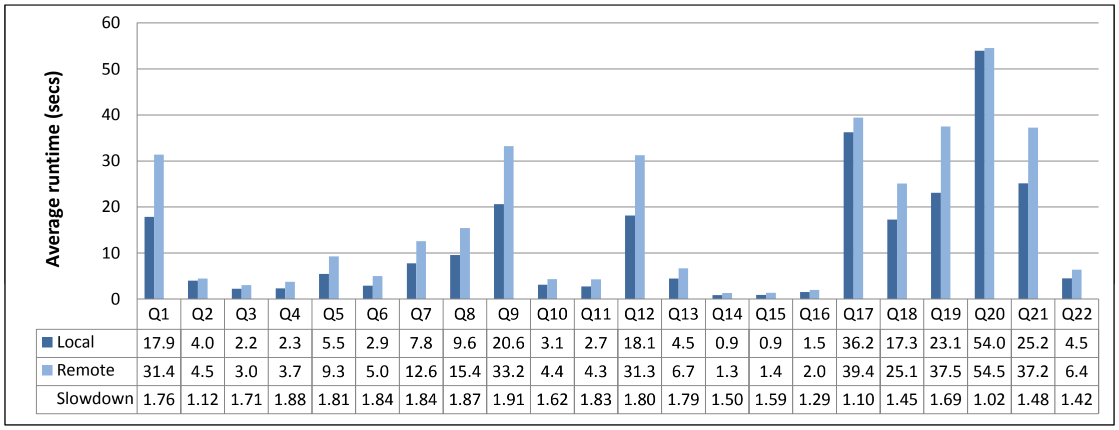
Figure 4.4: Scalability with Increasing Number of Nodes

### 4.5.3 Scalability

In the first experiment, we measure how performance scales with an increasing number of nodes in a cluster having only a single local node, i.e., a single node hosting the database. We run concurrent TPC-H streams first using only a single node (the local node), and then incrementally add more (remote) nodes up to a maximum of 16 nodes. The number of concurrent TPC-H streams is also increased with the number of nodes adding increasingly more load to the cluster. We compare the peak throughput in each case. We run this experiment with a read-only workload, using only the TPC-H query streams and not the update streams. We also run in a read-mostly case, in which the same TPC-H query streams are used but now the *lineitem* table is updated by a single writer every 15 seconds. The results for the read-only and read-mostly workloads are presented in Figure 4.4. These results have been normalized with respect to the maximum throughput achieved using a single node.

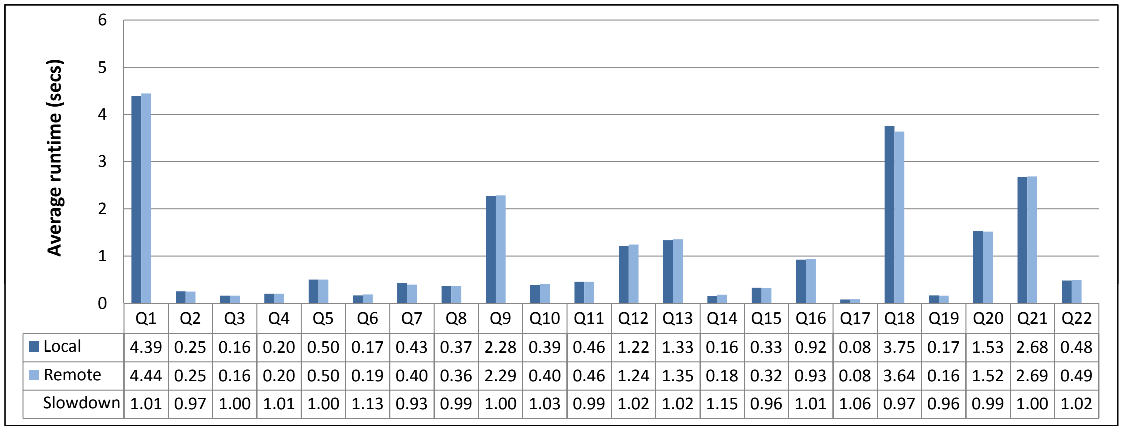
Focusing on the read-only case, with two nodes we achieve a maximum throughput of about 1.95 queries per second (QPS), which is nearly twice as much as the single node case. For four nodes, the performance peaks at about 3.81 QPS, which is about twice as that of the two nodes case. We see a similar trend moving forward, with 8 and 16 node clusters achieving a maximum throughput of 7.72 and 14.35 QPS, respectively. In the read-mostly case, we have the additional overhead of locking and selective cache invalidation, which causes the throughput curve to drop below the read-only curve. For both workloads, Figure 4.4 shows *almost linear* scalability. There is a slight loss of performance with higher number of nodes in both cases due to increased contention, but this is expected.

Nodes can be added or removed to/from the cluster in an on-demand and seamless fashion allowing the database to scale-out and scale-in as needed. This makes Chimera suitable for cloud computing environments by allowing scale-out for load balancing on a much finer time scale than what is possible with a pure shared nothing DBMS cluster.



| | Q1 | Q2 | Q3 | Q4 | Q5 | Q6 | Q7 | Q8 | Q9 | Q10 | Q11 | Q12 | Q13 | Q14 | Q15 | Q16 | Q17 | Q18 | Q19 | Q20 | Q21 | Q22 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ■ Local | 17.9 | 4.0 | 2.2 | 2.3 | 5.5 | 2.9 | 7.8 | 9.6 | 20.6 | 3.1 | 2.7 | 18.1 | 4.5 | 0.9 | 0.9 | 1.5 | 36.2 | 17.3 | 23.1 | 54.0 | 25.2 | 4.5 |
| ■ Remote | 31.4 | 4.5 | 3.0 | 3.7 | 9.3 | 5.0 | 12.6 | 15.4 | 33.2 | 4.4 | 4.3 | 31.3 | 6.7 | 1.3 | 1.4 | 2.0 | 39.4 | 25.1 | 37.5 | 54.5 | 37.2 | 6.4 |
| Slowdown | 1.76 | 1.12 | 1.71 | 1.88 | 1.81 | 1.84 | 1.84 | 1.87 | 1.91 | 1.62 | 1.83 | 1.80 | 1.79 | 1.50 | 1.59 | 1.29 | 1.10 | 1.45 | 1.69 | 1.02 | 1.48 | 1.42 |

Figure 4.5: Remote Execution Overhead: Start Up (Cold Cache)



| | Q1 | Q2 | Q3 | Q4 | Q5 | Q6 | Q7 | Q8 | Q9 | Q10 | Q11 | Q12 | Q13 | Q14 | Q15 | Q16 | Q17 | Q18 | Q19 | Q20 | Q21 | Q22 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ■ Local | 4.39 | 0.25 | 0.16 | 0.20 | 0.50 | 0.17 | 0.43 | 0.37 | 2.28 | 0.39 | 0.46 | 1.22 | 1.33 | 0.16 | 0.33 | 0.92 | 0.08 | 3.75 | 0.17 | 1.53 | 2.68 | 0.48 |
| ■ Remote | 4.44 | 0.25 | 0.16 | 0.20 | 0.50 | 0.19 | 0.40 | 0.36 | 2.29 | 0.40 | 0.46 | 1.24 | 1.35 | 0.18 | 0.32 | 0.93 | 0.08 | 3.64 | 0.16 | 1.52 | 2.69 | 0.49 |
| Slowdown | 1.01 | 0.97 | 1.00 | 1.01 | 1.00 | 1.13 | 0.93 | 0.99 | 1.00 | 1.03 | 0.99 | 1.02 | 1.02 | 1.15 | 0.96 | 1.01 | 1.06 | 0.97 | 0.96 | 0.99 | 1.00 | 1.02 |

Figure 4.6: Remote Execution Overhead: Steady State (Warm Cache)

### 4.5.4 Remote Execution Overhead

Our next experiment measures the overhead of remote execution, i.e., the overhead of executing a query from a remote node against a shared database hosted on a local node. We run each of the 22 TPC-H queries 5 times, measure the query execution time, and then take the average. To get startup costs, we flush the SQL Server buffer cache between runs. Figure 4.5 presents the times to run TPC-H queries locally and remotely, as well as the slowdown, i.e., the ratio of the remote to the local running time. Queries experience an overhead ranging from moderate (about 2%) to relatively high (about 91%) when run with a cold database cache. On average it takes about 1.6 times longer to run a query on a remote node as compared to a local node. The worst case occurs for Q9 where the remote execution time is 1.91 times that on the local node. We note that all the high overhead queries access a significant portion of the TPC-H database. For example, Q9 is an aggregation query over more than half the tables of the TPC-H database including the *lineitem* and *orders* table, the two largest tables in the database. This causes the remote node to request a large number of pages across the network from the local node resulting in high overhead at startup.

We believe that most of the overhead observed in query runtimes presented in Figure 4.5 can be attributed to the network latency of shipping database pages to the remote system. Next, we conduct an experiment where we repeat the same experimental runs except that we do not flush the database buffer cache between runs, i.e., the system is in a steady state. Results are presented in Figure 4.6. In this case queries have about the same runtime on both local and remote nodes with an average overhead of less than 1%. We note that this is the common case for such a data sharing system when executing a read-mostly workload. It is very likely that the active pages of even a fairly large table will entirely fit in memory on the remote node. Thus, the high overhead presented in Figure 4.5 should only be incurred at startup, while subsequent execution times will have small overhead similar to the results presented in Figure 4.6. In subsequent experiments, unless stated otherwise, the system is in a steady state with a warm cache.

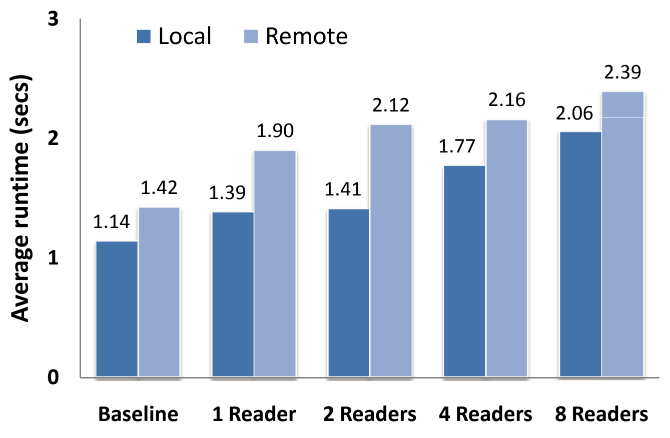
### 4.5.5 Prototype Overhead

Next, we present an experiment where we measure the overhead of running Chimera. The goal is to show how vanilla SQL Server performs as compared to Chimera. Table 4.1 shows the runtime and slowdown factor when we run the 22 TPC-H queries on the local node with and without Chimera. Each value presented in Table 4.1 is an average over 5 runs.

In most of the cases the query runtimes with and without Chimera are about the same. In some cases (e.g., Q8, Q9), query runtime with Chimera is higher than without, indicating a small overhead. In some other cases (e.g., Q2, Q13),

| **TPC-H Query** | **Runtime Without Chimera (ms)** | **Runtime With Chimera (ms)** | **Slowdown Factor** |
|---|---|---|---|
| **Q1** | 4477 | 4389 | 0.98 |
| **Q2** | 279 | 253 | 0.91 |
| **Q3** | 182 | 163 | 0.90 |
| **Q4** | 199 | 202 | 1.02 |
| **Q5** | 504 | 502 | 1.00 |
| **Q6** | 166 | 165 | 0.99 |
| **Q7** | 356 | 429 | 1.21 |
| **Q8** | 329 | 367 | 1.12 |
| **Q9** | 2257 | 2275 | 1.01 |
| **Q10** | 381 | 390 | 1.02 |
| **Q11** | 470 | 459 | 0.98 |
| **Q12** | 1209 | 1217 | 1.01 |
| **Q13** | 1350 | 1333 | 0.99 |
| **Q14** | 155 | 157 | 1.01 |
| **Q15** | 315 | 331 | 1.05 |
| **Q16** | 922 | 923 | 1.00 |
| **Q17** | 76 | 79 | 1.04 |
| **Q18** | 3763 | 3753 | 1.00 |
| **Q19** | 160 | 168 | 1.05 |
| **Q20** | 1524 | 1533 | 1.01 |
| **Q21** | 2673 | 2677 | 1.00 |
| **Q22** | 460 | 481 | 1.05 |

Table 4.1: Overhead of Prototype Execution



| Slowdown (local) | 1.21 | 1.24 | 1.55 | 1.81 |
|---|---|---|---|---|
| (remote) | 1.34 | 1.49 | 1.52 | 1.68 |

Figure 4.7: Cost of Updates

queries seem to be running faster with Chimera. However, most of these differences are very small and fall within the range of experimentation error. Averaged over all runs, the overhead is less than 2%. The conclusion that we draw from these numbers is that Chimera adds a negligible overhead to query runtimes. Moreover, this extra overhead is incurred only when our data sharing capability is exercised.

### 4.5.6 Cost of Updates

Even though Chimera has been designed to provide elastic scale-out for read-mostly workloads, it is important to characterize how costly updates will be when readers are present in the system. In this experiment, we perform a simple update on the *lineitem* table on the local node and its runtime is measured to serve as a baseline. We consider four different scenarios where 1, 2, 4, or 8 remote nodes are reading the shared TPC-H database (including the *lineitem* table) in an infinite loop at the same time that the update query is executed on the local node. The runtime of the update query on the local node in different cases is presented in Figure 4.7. Each value reported is an average over 5 runs. Slowdown in this case is calculated as the ratio of the runtime in the multiple reader case to the baseline runtime.

In the local case, our baseline update query takes an average of 1.14 seconds. With one reader in the system, the same query takes an average of 1.39 seconds, which is about 1.21 times the baseline runtime. With two readers, the runtime is about 1.24 times the baseline. We see a similar trend for 4 and 8 readers, with slowdown factors of 1.55 and 1.81, respectively. At the updating node, each additional read node adds to the overhead of acquiring X locks, as required by our locking protocol. The global lock server has to wait for each reader node of the *lineitem* table to release the S lock on the shared resource before it can grant the

X lock to the writer node. This adds some extra overhead with increasing number of nodes, as shown in the local case in Figure 4.7.

In the next experiment we repeat the previous experiment except that the update is performed on one of the remote nodes – this is our new baseline. The results, presented as the remote case in Figure 4.7, show a similar pattern as the local case. The difference in this case is that the average time it takes to complete the update is slightly higher in all scenarios as compared to the local case. This increase is a result of the additional cost of accessing data from the local node because the update is executed on a remote node.

The results presented in this section show that the cost for acquiring X locks in our system increases sub-linearly with the cluster size. This experiment illustrates that there is a moderate cost to execute updates with conflicting reads, which is acceptable for our target read-mostly workloads.

### 4.5.7 Cost of Reads with Updates

In our next experiment, we quantify the cost of reads while at least one of the nodes (the local node in our case) is updating the database at varying intervals of 60, 30, 15, and 5 seconds. We repeatedly run one of the TPC-H read queries for a fixed duration of 300 seconds and measure average response time, the number of queries completed, and the number of queries executed per second (QPS). Results are presented in Table 4.2 for TPC-H query 6, 13, 20, and 21. Query 6, 20, and 21 conflict with the update query (i.e., read the same table) while query 13 performs a non-conflicting read. We have chosen this set to present a mix of queries with short to relatively long runtimes and with conflicting/non-conflicting reads to show how various update frequencies affect each group. Results with other TPC-H queries (not presented here) follow the same general trend.

The results for Q6 show that if updates are $\geq$ 60s apart, we are able to achieve performance very close to the state when updates are entirely absent. As we decrease the update interval from 60s down to 5s, we see a gradual decrease in performance resulting in a slightly higher average runtime (or low QPS), which is to be expected. However, note that the performance is adequate even at high update frequencies (i.e., every 5s). We see a similar trend for Q20 and Q21. Note that the overhead in these results includes the overhead of locking and selective cache invalidation. For Q13, the performance remains constant at all update frequencies because the query performs a non-conflicting read.

This experiment demonstrates that it is possible to provide adequate performance for scenarios where reads run for relatively long intervals between writes, which is the case in our target read-mostly workload.

| | **Update Freq. (sec)** | **Queries Completed** | **Avg Runtime (ms)** | **Steady State Avg (ms)** | **QPS** |
|---|---|---|---|---|---|
| **Q6** | 60 | 1455 | 206 | 187 | 4.85 |
| | 30 | 1363 | 215 | | 4.54 |
| | 15 | 1269 | 234 | | 4.23 |
| | 5 | 1130 | 264 | | 3.77 |
| **Q13** | 60 | 220 | 1376 | 1353 | 0.73 |
| | 30 | 220 | 1375 | | 0.73 |
| | 15 | 216 | 1388 | | 0.72 |
| | 5 | 218 | 1376 | | 0.73 |
| **Q20** | 60 | 185 | 1616 | 1518 | 0.62 |
| | 30 | 180 | 1654 | | 0.60 |
| | 15 | 148 | 1991 | | 0.49 |
| | 5 | 146 | 2039 | | 0.49 |
| **Q21** | 60 | 107 | 2783 | 2687 | 0.36 |
| | 30 | 104 | 2853 | | 0.35 |
| | 15 | 100 | 3011 | | 0.33 |
| | 5 | 83 | 3597 | | 0.28 |

Table 4.2: Cost of Reads with Updates

## 4.6 Related Work

There is much work in the literature that is aimed at facilitating data sharing, much like the goals of our system. Chimera has some similarity to work in the areas of distributed file systems (DFS), global lock servers, replicated databases, cloud databases that use data sharing, and scalable cloud stores.

Parts of Chimera are reminiscent of distributed file systems (DFS), e.g., [131, 143]. Our lock service is similar in spirit to modern file system lock managers (e.g., [28, 69]). Furthermore, the caching of pages in the buffer pool in Chimera shares aspects with the management of data in a DFS file cache. However, the details of the design are considerably different due to the complexities imposed by the database semantics compared to the simpler file and directory abstractions supported by a DFS.

Our approach to load balancing is superficially similar to replication used for the same purpose by some database vendors [90, 94, 106]. A replicated database can be used effectively to balance load by sending read requests to one of the different replicas, making it similar in spirit to our system. However, write requests have to be propagated to all replicas and this does not really benefit load balancing. Chimera is complementary to replicated databases in that it can be used in conjunction with a replicated database to balance load. In particular, Chimera can spread load to all server nodes in a cluster, even those that do not hold a replica. This is in contrast to systems that use replication alone because requests can only be handled by the nodes that hold a replica.

Similar to our work, there have been some recent efforts to implement relational databases in the cloud using data sharing techniques [22, 79]. The technique presented in [22] makes data accessible to multiple nodes by storing it in cloud storage. Low-level protocols are provided to read and write data from/to the cloud storage. A detailed discussion and quantitative comparison of techniques that use replication, partitioning, or data sharing to implement cloud databases is presented in [79]. However, as opposed to Chimera, consistency is sacrificed in these systems in favor of scalability and availability, and general purpose ACID transactions are not provided.

Finally, providing scalability for large data sets is a property of cloud infrastructures like Bigtable [31] and PNUTS [35]. One important difference here is that these infrastructures, while supporting atomic operations, do not support transactions or database functionality. Chimera fully supports transactions and provides database functionality.

## 4.7 Summary

Chimera demonstrates the design and implementation of data sharing as an extension to a shared nothing system. The goal of Chimera is to get the best of both worlds – data sharing flexibility with shared nothing simplicity. Chimera uses a hybrid database architecture to enable sharing at the granularity of tables, and has been designed as a simple extension to a shared nothing database system. It uses low-cost, coarse-grained global locks to control how data is shared among nodes and selective cache invalidation for global buffer consistency. Chimera demonstrates the ability to move database functionality from a local node storing the database to a remote node for elastic scale-out without moving tables between the disks of these nodes. This permits load balancing in a matter of minutes not hours. Using a TPC-H database, we show that Chimera is able to serve read-mostly workloads with acceptable performance overhead while providing nearly linear scalability with increasing number of nodes.

The second approach in this thesis for elastic scale-out exploits data partitioning as opposed to data sharing, and is presented in the next chapter.

# Chapter 5

# Elastic Scale-out for Partition-Based Database Systems

As noted previously in this thesis, there is currently a lot of interest in elastically scaling database systems, and in this chapter we argue that a good way to solve this problem is to start with scalable (but not elastic) parallel database systems. The approach that we present here relies on data partitioning and is different from the approach presented in Chapter 4, which enables elasticity by using data sharing.

Traditional scalable database systems enable multiple *nodes*(i.e., servers) to manage a database, but the set of nodes is static. These systems distribute the database among the nodes by relying on *replication* [11, 14, 74, 153] or *partitioning* [36, 93]. Replication comes with a cost due to data duplication and maintaining consistency among the replicas. Partitioning is also non-trivial since it requires users to define how to partition the database, and it makes dealing with multi-partition transactions problematic. However, partitioning is gaining popularity as a way to scale database systems (e.g., [39, 40, 132, 133]), and in this chapter we argue that partitioned, shared nothing database systems are a good starting point for DBMS elasticity.

Our basic approach is to start with a small number of nodes that manage the database partitions, and to add nodes as the database workload increases. These new nodes could be (spare) physical servers that are dynamically provisioned from a local cluster. Alternatively, in cloud computing environments such as Amazon's EC2 [7], the new nodes could be dynamically provisioned virtual machines. When adding new nodes, the database partitions are redistributed among the nodes so that the workload gets shared among a larger set of nodes. Conversely, when the workload intensity decreases, the number of nodes can be reduced and partitions can be moved to a smaller set of nodes. Redistributing partitions under this approach can be costly and difficult to manage because of its disruptive effect on transaction processing [137, 148]. Thus, this approach requires efficient mechanisms for partition redistribution.

We have implemented this elastic scale-out and scale-in approach in VoltDB [152], including the required efficient partition redistribution mechanism [99]. VoltDB is a parallel shared nothing partition-based database system. Like other shared nothing database systems [94, 106, 107, 141], VoltDB divides the database into disjoint partitions based on some partitioning key and distributes these partitions to the nodes of a cluster. In this chapter, we describe the changes that we made to VoltDB to enable us to dynamically add new nodes to the cluster and redistribute partitions, and we experimentally demonstrate the effectiveness of our elastic scale-out and scale-in mechanisms under varying load.

Our approach to database elasticity gives rise to a number of research problems in the area of database provisioning and manageability. These research challenges need to be addressed in order to implement a database management system that can scale elastically in response to a time varying workload. For example, an elastically scalable database system should be able to decide when to scale-out/in. Timing of these scaling decisions is critical. Responding to a load spike too early or too late will result in an over- or under-provisioned system, respectively, which in turns leads to either lost customers or wasting of costly computing resources [16]. Also, once the DBMS decides to scale out/in it also needs to decide which partitions to move, to which nodes, during a scaling operation. In this chapter, we focus on the latter problem which we call the *partition placement problem*. We show that partition placement can have a significant impact on performance in an elastically scalable database system. We then go on to present a *workload-aware* partition placement strategy that relies on mathematical optimization to find an optimal partition placement depending on the workload and whether the workload is known in advance or not. More specifically, we present a problem formulation for the partition placement problem as a mixed-integer program (MIP) in (1) an *offline* setting where the entire workload and how it varies over time is known in advance, and (2) an *online* setting where we do not know the future load. We use a general purpose solver (IBM ILOG CPLEX [70]) to find a solution to the offline and online optimization problems. We present a comparison of the two approaches through actual experiments, and show that both of these approaches outperform a simple partition placement strategy. Our partition placement algorithms can be used with any partition-based elastic database system. In this thesis, we use these algorithms with VoltDB extended with our elasticity mechanisms, but we can also use them with any system that provides partition-based elastic scale-out.

The rest of this chapter is organized as follows. In Section 5.1, we present previous work in the areas of scalable database systems and partition placement. In Section 5.2, we present an overview of VoltDB. Section 5.3 presents the changes that we made to VoltDB to make it elastically scalable. Section 5.4 then focuses on the specific problem of partition placement with elastic scale-out. We present experimental results that show the effectiveness of our implementation for time varying workloads in Section 5.5, as well as the experiments that show the advantage of our partition placement strategy. Section 5.6 summarizes the chapter.

## 5.1 Related Work

Improving scalability of database systems has been a focus of research for over two decades. Interested readers are referred to Chapter 2 (Section 2.4) for a more detailed survey of previous techniques used to build scalable database systems. Most of the existing techniques for scaling database systems either rely on data replication [11, 14, 23, 33, 60, 74, 112, 153] or data partitioning [26, 36, 46, 93, 124]. Our focus in this work is to use a partition-based, shared-nothing database system as a starting point to build an elastically scalable database system. Previous research does not deal with using partitioning for scale-out, mainly because of the high cost of repartitioning and the cost of physically moving data around, which has a negative impact on the performance of ongoing transaction processing [137, 148]. Furthermore, none of the existing techniques explicitly deal with elastic scale-out. In this chapter, we present efficient mechanisms for moving database partitions for elastic scale-out that also minimize the impact on normal database processing.

Similar to our work, the Relational Cloud project at MIT [38] aims to implement a scalable, relational database service for the cloud. As part of this project, researchers have presented workload-aware techniques for consolidation of database workloads [40] with an aim to minimize system resources and improve system utilization while providing the desired level of performance. However, their focus is not on elastic scale-out for time varying workloads. Other research related to the same project [39] presents techniques for defining optimal database partitioning and/or replication in a workload-aware fashion that improves database scalability. Our focus in this work is not coming up with new partitioning or replication schemes. Our approach can work with any existing partitioning scheme to implement elastic scale-out for relational database systems.

ElasTraS [42] is an elastically scalable, fault-tolerant, transactional DBMS for the cloud, and is closest to our work. Like VoltDB, ElasTraS uses database partitioning and performs scaling at the granularity of a partition. However, ElasTraS relies on a shared storage tier which allows for a simpler mechanism for data migration. VoltDB stores the data completely in-memory on each host, requiring a more elaborate mechanism for data migration (Section 5.3). Furthermore, ElasTraS relies on schema-level database partitioning and limits update transactions to a single partition. VoltDB, on the other hand, uses table level partitioning and allows multi-partition transactions at the cost of reduced performance for only these transactions. Aside from these differences, our work is very similar to ElasTraS and the research problem of partition placement presented in this chapter is applicable to both of these systems.

There is an increasing body of work in the area of elastically scalable data stores, especially for cloud computing environments. Key-value stores [31, 62, 80] can be scaled elastically, similar to our solution. At a superficial level, such systems bear some resemblance to relational database systems in that they also store data as rows and columns. However, these systems provide a very simple interface that allows clients to read or write a *value* for a given *key* from the store. SQL is not supported, hence the name "NoSQL" given to these systems. In addition, these systems typically support only single-row atomic updates, not general application-defined transactions like those that are supported by relational database systems. Some key-value stores provide only eventual consistency guarantees. These limitations make it easier for these systems to scale. In contrast, our system implements elastic scalability in an ACID compliant DBMS (namely, VoltDB) that provides full SQL functionality.

The problem of partition placement (or more generally data placement) and re-distribution for load balancing has been studied extensively in the past both in the context of database systems [15, 26, 36, 67, 93, 127] and file servers [51, 155]. This problem is also very similar to the problem of consolidating multiple database workloads with an aim to minimize the number of physical servers used and maximize load balance [40].

The possibility of the database cluster growing and shrinking dynamically adds another dimension to the problem of partition placement. Previous research either does not deal with the data re-distribution problem for load balancing, or if it does, it works with a fixed number of nodes. Clearly, this is not sufficient for the case where nodes can be added or removed dynamically for elastic scaling. Our goal in this work is to find an optimal partition placement where the number of physical servers used can change dynamically, with an aim to minimize the cost of running the system and/or the data migration cost. This problem has not been addressed before.

The general partition placement problem has been shown to be NP-complete [127]. Various heuristic techniques are typically used to find a solution. More generally, this problem bears close resemblance with the zero/one multiple knapsack problem, and so similar techniques, such as genetic algorithms, can be used for finding a solution [77]. In this work, we use mathematical programming to solve this problem. Similar to our work, in [21, 138] authors adopt a mathematical programming approach to find an optimal allocation of servers to applications (partitions in our case) in a virtualized data center environment. Their goal is to use this approach for server consolidation to minimize the cost of operating a data center. An integer programming approach to assign servers to applications has been presented in [126], again with a goal of reducing server costs through consolidation. In this work, we formulate the partition placement problem as a mixed-integer program (MIP), with a goal of minimizing server costs through elastic scale-out/in, and aim to find an exact solution instead of relying on heuristics. As such, our goal is not to present a new heuristic for solving this problem. Instead, we use a general purpose solver to find a solution.

## 5.2 Overview of VoltDB

VoltDB [152] is a in-memory database system derived from H-Store [73]. VoltDB has a shared nothing architecture and is designed to run on a multi-node cluster. It divides the database into disjoint partitions and makes each node responsible for a subset of the partitions. Only a node that stores a partition can directly access the data in that partition, and such nodes are therefore sometimes called the "owners" of that partition. This shared nothing partitioning has been shown to provide good scalability while simplifying the DBMS implementation.

VoltDB has been designed to provide very high throughput and fault tolerance for transactional workloads. It does so by making the following design choices: (1) all data is stored in main memory, which avoids slow disk operations, (2) all operations (transactions) that need to be performed on the database are pre-defined in a set of stored procedures that execute on the server, i.e., ad hoc transactions are not allowed, (3) transactions are executed on each database partition serially, with no concurrent transactions within a partition, thereby eliminating the need for concurrency control, and (4) partitions are replicated for durability and fault tolerance.

A VoltDB cluster comprises one or more nodes connected together by a local network, each running an instance of the VoltDB server process. Each node stores one or more database partitions in main memory. The VoltDB server at each node is implemented as a multi-threaded process. For each partition hosted on a VoltDB node, a separate thread within the server process manages that partition and is responsible for executing transactions against that partition. The best practice for achieving high performance with VoltDB is to keep the number of partitions on a node slightly smaller than the number of CPU cores on that node. This way, each thread managing a partition will always be executing on its own CPU core, and there will still be some cores for the operating system and other administrative tasks. Thus, threads never contend with other threads for cores. Furthermore, these threads never wait for disk I/O since the database is in memory, and never wait for locks since transactions are executed serially. This means that the CPU cores can be running at almost full utilization doing useful transaction processing work, which leads to maximum performance.

Clients connect to any node in the VoltDB cluster and issue transactions by calling pre-defined stored procedures. Each stored procedure is executed as a database transaction. VoltDB supports two types of transactions:

1. **Single-partition Transactions:** These transactions, as the name implies, are the ones involving only a single database partition and are therefore very fast.

2. **Multi-partition Transactions:** These transactions require data from more than one database partition and are therefore more expensive to execute. Multi-partition transactions can be completely avoided if the database is cleanly partitionable, which is the case for the kinds of transactional workloads that VoltDB targets, and is our focus in this thesis.

In order to tolerate node failures, VoltDB replicates partitions on more than one node. The replication factor can be specified as part of the initial configuration of the VoltDB cluster. A VoltDB cluster configured with $k$-safety will have $k+1$ instances of every partition, and can tolerate at most $k$ node failures. Transactions are executed on the different replicas of a partition in the same serial order, and VoltDB waits for all copies of a transaction to finish before acknowledging its success to the client.

## 5.3 Enabling Elastic Scale-out with VoltDB

In order to implement elastic scale-out, we need to provide mechanisms to: (1) dynamically add new nodes to the cluster, and (2) move database partitions from one node to another. One of the reasons for choosing VoltDB for elastic scale-out is that it already provides much of the needed functionality. This simplifies the implementation of our scale-out mechanism.

### 5.3.1 Growing the Size of the Cluster

In large cluster deployments, node failures are common. Therefore, VoltDB implements an efficient way of identifying failed (down) nodes and a *node rejoin* mechanism which introduces a node back into the cluster after it recovers from failure.

In our implementation, we violate the VoltDB best practices and assign more partitions to a node than the number of CPU cores on that node. This means that VoltDB threads will contend for CPU cores, but this is acceptable in a lightly loaded system. As the load on the system increases and we need to scale out, we introduce new nodes into the cluster and move some partitions to these new nodes. When the system scales out to its limit, there is one VoltDB thread per core. We made the design decision to build our system with the notion of a maximum scalability limit since this notion greatly simplifies the scale-out mechanism. The simplification stems from the fact that VoltDB can be made aware in advance of the number of nodes that can potentially participate in the cluster, and the mapping of partitions to these nodes. Knowing this information in advance enables us to use VoltDB's failure handling mechanism for managing scale-out as we describe next.

To enable the number of nodes in the cluster to grow dynamically we introduce a new type of node called the *scale-out* node. A scale-out node is a node that initially has no database partitions mapped to it, and that looks to VoltDB like a "failed" node. We initialize the cluster with a fixed number of additional scale-out nodes. These nodes begin as inactive (i.e., they do not store data or process transactions), so the cluster will ignore them with little overhead. Note that at this point, a scale-out node is merely a data structure with no physical counterpart. When we need to actually introduce such a node into the cluster, we can bring up an actual physical server (or a virtual machine) that corresponds to a scale-out node. To dynamically grow the size of the cluster, we use a slightly modified version of VoltDB's node rejoin mechanism. At the completion of rejoin, the scale-out node will have transitioned from the "failed" state to the "active" state, and will have database partitions mapped to it. Effectively, we have made VoltDB think that this node was part of the cluster all along, but was just failed, and is now back up. However, at this point the node still has no data for any of the partitions. The partitions need to be moved from one of the active nodes, which is the second step of scale-out.

### 5.3.2 Moving Database Partitions Between Nodes

Introducing a new node into the cluster requires copying data from one of the existing nodes to the new node. This data copying step is also required when recovering a failed node. For this purpose, VoltDB implements a *recovery* mechanism that runs after a node has rejoined the cluster. The recovery mechanism works at the database partition level, and copies the data from a *source* partition to a *destination* partition in a consistent manner. Note that source partitions reside on one of the active nodes, while destination partitions reside on the node that has rejoined the cluster. Once all the database partitions mapped to the rejoining node have been recovered, this node can start accepting new transactions.

For scale-out, we made small changes to the recovery mechanism. Once the scale-out node has rejoined the cluster, it will run recovery on all the database partitions mapped to it to populate them. In the original recovery mechanism, after the data is copied from the source node to the destination node, both nodes are active and accept transactions. In contrast, once recovery is finished when scaling out, the source partitions are shutdown – reflecting the change of "ownership" for scaled-out partitions from the source node to the scale-out node.

The converse of the operations described above are used when scaling in because the load on the system has become lower: partitions are moved away from scale-out nodes and back to the original nodes, and when all the partitions on a scale-out node are moved back to the original nodes, that node is shutdown and returned to the pool of (inactive) scale-out nodes.

Similar to our approach, Microsoft has implemented mechanisms to provide scale-out and scale-in in Windows Azure SQL Database (formerly SQL Azure) through *federations*. A federation contains several members, each of which is an instance of Windows Azure SQL Database. As with our scale-out and scale-in mechanisms, federations can be SPLIT or MERGED to provide elastic scalability [68]. Given this similarity, the research challenges that arise in both systems (described next) are the same.

Having presented the scale-out mechanism that we implemented in VoltDB, we now turn our attention to the research problem of partition placement in an elastically scalable DBMS.

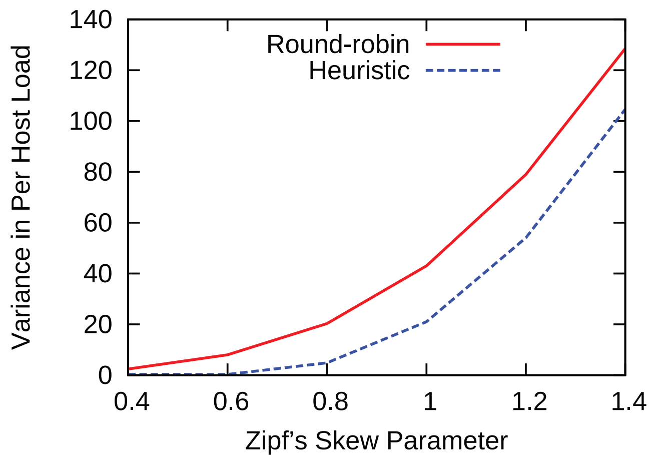


Figure 5.1: Round-robin vs. Greedy Heuristic

## 5.4 Partition Placement in an Elastically Scalable DBMS

In our approach presented above, we enable elasticity by moving database partitions for scale-out. The key idea is that a database is stored as a collection of partitions, and when the database server becomes overloaded, it transfers the "ownership" of some of these partitions to another node that is dynamically added to the cluster. Similarly, for scale-in, an underloaded node will transfer the ownership of its partitions to some other node in the cluster. The loads placed on each partition might not be the same, as is implicitly assumed by VoltDB and other similar systems. It is possible that certain partitions might be "hotter" than other partitions, which may result in unbalanced load among the nodes of the cluster. Nodes hosting one or more hot partitions will always be more loaded than other nodes. With non-uniform partition loads, the initial placement of partitions and the decision of which partitions to move, and to which nodes, during a scaling operation become non-trivial.

Figure 5.1 shows the result of a simulation that compares two data placement strategies in terms of overall load balance when database partitions are not uniformly loaded. In our simulation, we use 40 partitions and the loads on these partitions are distributed according to a Zipf distribution. The x-axis in Figure 5.1 shows the value of the Zipf's skew parameter (the higher this parameter, the more skewed the distribution), and the y-axis shows the variance in per host load. The higher the variance, the more unbalanced the system. We compare VoltDB's default placement strategy (i.e., round-robin) with a simple greedy heuristic that places the highest loaded partitions on the least loaded hosts. As shown in Figure 5.1, for moderate skew (0.4) both strategies provide equally well balanced systems. As the skew in partition load grows, the default (round-robin) strategy starts to perform worse than the simple heuristic. We see the greatest opportunity for optimization when Zipf's skew coefficient is around 1. As the skew increases beyond 1, both strategies start to perform worse. This is because with higher skew values, a few partitions always have the most load, so the overall variance in per host load will be very high regardless of the placement strategy used.

As shown by the simulation above, placement of database partitions matters when the load is non-uniform. Moreover, there is room for improvement over a simple strategy such as round-robin. An important research challenge is to devise a technique that is able to provide close to optimal allocation of partitions to nodes in the presence of skew in partition load, while maximizing system throughput and minimizing the cost of running the system.

### 5.4.1 Problem Definition

Given a set of $P$ database partitions, the load on each partition in transactions per second, and a set of $N$ nodes, where each node has a certain CPU and memory capacity, we want to find the optimal number of nodes required $M$, where $M \leq N$, and a mapping of partitions to these nodes. Optimality can be defined in terms of balancing load among all nodes, maximizing overall throughput, and/or minimizing the cost of running the system by minimizing $M$ for a given workload.

The partition placement problem can be formulated using mathematical programming (e.g., integer programming) and solved using mathematical optimization techniques. We present two different optimizers:

1. **Offline Optimizer**: In an offline scenario, we want to compute a partition placement, in advance, for a period of time, $\{1, ..., T\}$, which we call the *planning period*. We assume that the future load on each database partition is known in advance. We can use a workload predictor to acquire the workload profile or we can extract it from the logs of an already executed workload. This optimizer is most useful for the case where we have a periodic workload that is executed on a regular basis, and so the workload is known in advance. There are many practical data center workloads that are predictable due to their periodic nature [16, 52]. The output of the offline optimizer is an optimal *schedule* comprising the number of nodes required and a mapping of partitions to nodes for each time interval $t \in \{1, ..., T\}$.

2. **Online Optimizer**: In the online case, we do not know the entire workload in advance. We only know the instantaneous load on each partition and the system state (e.g., current CPU utilization on each node) for a given time interval $t$. Given this information, we want to make an instantaneous decision of whether a scaling operation is required, and if so, we want to find the optimal partition to node mapping for the the time interval $t + 1$.

We now present a problem formulation for partition placement with elasticity for the offline and online optimizers and show how we can use a general purpose solver for finding an optimal solution. The main contributions of this work are: (1) formulating the offline optimization problem, (2) finding a solution to this problem, and (3) showing experimental results that quantify the benefits of our approach. The online optimizer was developed separately, and we present it here for comparison purposes only.

### 5.4.2 Problem Formulation for the Offline Optimizer

As noted above, there are various scenarios where we know the workload in advance, e.g., periodic workloads, and we want to find an optimal placement of partitions for, say, a daily, weekly, or monthly planning period. In this section, we present a formulation for the problem of partition placement where we know the load on each partition for the entire planning period. Our goal is to find an allocation of partitions to nodes that minimizes the number of nodes used and the migration costs for the entire planning period. We formulate this problem as a mixed-integer program (MIP) and show how we can use a general purpose solver to find a solution to this problem.

#### Notation

Let $P$ denote the total number of unique partitions in a database, that are to be placed on $N$ nodes in a cluster, where each partition is replicated $k$ times. We want to find an optimal placement of partitions on nodes for a planning period with $T$ time intervals. Let $A^t$ denote the assignment matrix of partitions to nodes in time interval $t$, where the elements of this matrix are $A^t_{ij} \in \{0, 1\}$, where $i = 1, 2, ..., P$, $j = 1, 2, ..., N$. We have $T$ such matrices for $t = 1, 2, ..., T$. A value $A^t_{ij} = 1$ means that partition $i$ is assigned to node $j$ in the time interval $t$. Let $X^t_j$ denote the assignment matrix for the nodes. A value of $X^t_j = 1$ means that the $j$-th node is used in the time interval $t$. A node is used if there are any partitions allocated to it in a given time interval $t$. Let $C_j$ denote the number of CPU cores of the $j$-th node, each of which can handle a maximum CPU load of $L_j$ (defined in the range (0,1]). Let $S_j$ denote the cost of running the $j$-th node and $M_j$ denote the memory capacity of the $j$-th node. Let $l^t_i$ denote CPU load generated by the $i$-th partition in the time interval $t$, and $m^t_i$ denote the memory required by the $i$-th partition. $I$ denotes the fixed cost of migrating a partition. And finally, let $k$ denote the replication factor.

For our formulation, we assume that all the $N$ nodes in the cluster are homogeneous, i.e., they have the same number of CPU cores $C$, same memory capacity $M$, and same running cost $S$. We also assume that the memory required for each partition $m^t_i$ is fixed, and that whenever a partition is migrated from one node to another we pay a migration cost equal to $I$. Lastly, the value of $T$ for a given planning period is workload dependent. For example, for a planning period of one hour, for a volatile workload, we may want to recalculate the optimal allocation every minute, i.e., set $T = 60$. On the other hand, for a less volatile workload, we may want to calculate an optimal allocation every five minutes by setting $T = 12$ for the same planning period of one hour. Choosing an optimal value of $T$ for a given workload is outside the scope of this work.

## Objective Function

The goal of the optimizer is to find a set of assignment matrices $A = \{A^0, ..., A^T\}$ that minimizes the overall cost of nodes for the entire planning period while minimizing the migration cost as well. Using the above notation, such an objective function can be expressed as:

$$\begin{aligned} \textit{minimize} \quad & \sum_{t=1}^{T} \left( \sum_{j=1}^{N} (X_j^t * S) \right) + \\ & \sum_{t=2}^{T} \left( \sum_{i=1}^{P} (|A_{ij}^t - A_{ij}^{t-1}| * m_i^t * I) \right) \end{aligned} \tag{5.1}$$

The first part of the objective function minimizes the cost of running the system by minimizing the number of nodes used at any time interval $t$. The second part minimizes the migration cost by minimizing the number of partitions migrated.

By using the above objective function, we can emphasize either the cost of running a node or the migration cost. If we set the cost of migration, $I$, low relative to the cost of running a node, $S$, this objective function will output a fully elastic schedule, where the number of nodes used at any time interval $t$ will always be minimal. On the other hand, if we set the migration cost high relative to the cost of running a node, the optimizer will favor schedules that require less movement at the expense of some extra nodes. The DBA chooses the values for $S$ and $I$ depending on the workload and the particular server environment, taking into account issues such as the difficulty of provisioning a new server, expected contention for network bandwidth, and the need for stability in partition placement.

## Constraints

We want to minimize the objective function given above subject to the following constraints:

1. **Replication:** Since partitions are replicated for durability and fault tolerance, every replica of a partition must be assigned to exactly one server at any time interval $t$, and the servers must be different. As the decision variables $A_{ij}^t$ are binary, the sum of every row must be equal to $k$:

$$\sum_{j=1}^{N} A_{ij}^t = k \quad \text{for } i = 1, \dots, P \tag{5.2}$$

2. **CPU Capacity:** The CPU load of all the partitions assigned to a node $j$ at any time interval $t$ should not exceed the maximum CPU capacity of that node $L * C$:

$$\sum_{i=1}^{P} A_{ij}^{t} * l_i^t \leq L * C * X_j^t \text{ for } j = 1, \ldots, N \text{ and } t = 1, \ldots, T \tag{5.3}$$

3. **Memory Capacity:** The memory required for all the partitions assigned to a node $j$ at any time interval $t$ should not exceed the maximum memory capacity of that node $M$:

$$\sum_{i=1}^{P} A_{ij}^{t} * m_i^t < M \quad \text{for } j = 1, \ldots, N \tag{5.4}$$

Note that all of the above constraints are *linear* constraints but our objective function is *non-linear* due to the presence of absolute values. This means that the above formulation is a mixed-integer non-linear program (MINLP). A MINLP formulation is generally more computationally expensive to solve than a mixed-integer linear program (MILP). In order to remove an absolute value $|x|$ from an objective function, a new decision variable $y$ can be introduced, replacing all instances of $|x|$ in the objective function. In addition, two constraints of the form $x - y \leq 0, -x - y \leq 0$ are introduced. We used this technique to make the problem formulation linear.

#### Finding a Solution

As stated above, the decision variables $A_{ij}^t$ and $X_j^t$ are binary. Since now the objective function and constraints are linear, this formulation is a mixed-integer linear program (MILP), which can be solved by using any general purpose solver. For this work, we use IBM ILOG CPLEX Optimization Studio [70]. The above problem formulation was implemented in the Optimization Programming Language (OPL), and solved using the CPLEX optimizer integrated in the Optimization Studio. The ability to solve this optimization problem using a general purpose solver instead of relying on a custom built solver is an added advantage of our formulation.

### 5.4.3 Problem Formulation for the Online Optimizer

The goal of the online optimizer is to find an optimal partition placement at any given time by taking into account only the instantaneous characteristics of the workload. In contrast to the offline optimizer, where we are planning in advance, the time to find a solution is critical for an online optimizer since it will be used in live partition migration scenarios.

The online optimizer is also designed with slightly different optimization objectives. First, similar to the offline optimizer, the online optimizer aims to minimize migration costs by avoiding solutions that involve excessive partition migration. The second objective of the online optimizer, in contrast to the offline optimizer, is to prefer solutions that divide system load evenly across the nodes of a cluster, i.e., to do load balancing. This is done with an aim to maximize overall resource utilization in the cluster and to avoid any possible overload of a particular node, which can negatively impact the throughput of the entire cluster.

### Notation

We use notation similar to the offline optimizer, with a few additions. First, we do not use the node assignment vector $X_j^t$ in the online optimizer. Instead, we represent the number of *live* nodes at time $t$ using a variable $N^t$ which is calculated as:

$$N^t = \sum_j signum(\sum_i A_{ij}^t) \tag{5.5}$$

The *signum* function has a value of 1 if its argument is greater than 0, and it has a value of 0 if its argument is equal to 0. Second, $O_j$ denotes the optimal per core CPU load allowed on the $j$-th node. This is the load at which that core gives the best performance.

### Objective Function

Given the current state of the system at time $t$, the objective of the online optimizer is to find an optimal assignment matrix $A^{t+1}$. The number of hosts in this placement is represented by $N^{t+1}$, and it can be calculated in a manner similar to Equation (5.5) by using $A^{t+1}$ as input to the *signum* function. The objective function is defined as:

$$\begin{aligned} minimize \qquad & \sum_j^N \sum_i^P (|A_{ij}^{t+1} - A_{ij}^t| * m_i^t) \quad + \\ \epsilon * \quad & \frac{\sum_{j=1}^{N^{t+1}} |\sum_{i=1}^P A_{ij}^{t+1} * l_i^t - O_j * C|}{N^{t+1}} \end{aligned} \tag{5.6}$$

The primary goal of the objective function is to minimize the movement of data during migration. The secondary goal is to balance the load evenly across nodes and to ensure that processing resources are utilized efficiently. The objective function is a sum of two separate functions that correspond to the primary and secondary goals of optimization. The first function represents the total movement of partition data during the migration process. The second function represents the average absolute deviation of the load generated by partitions assigned to a node from the optimal load for a node, multiplied by a very small value $\epsilon$.

The objective function is designed to ensure that the primary goal of optimization is satisfied first. Among the solutions that satisfy the primary goal, the second function selects the solution that has the minimum average absolute deviation from the optimal load $O_j$. As the value of $\epsilon$ is small, the second function will not interfere with the ordering of solutions produced by the first function. If there are multiple solutions with the same amount of data movement, then the second function will select the one that minimizes average absolute deviation from the optimal load.

The above problem formulation for the online optimizer is non-linear because of (1) the presence of absolute values in the objective function and (2) the variable $N^{t+1}$ in the denominator. In order to remove the absolute values from the objective function, the method mentioned in Section 5.4.2 was used. We can remove the variable $N^{t+1}$ by fixing $N^{t+1}$ to be a constant. This means that the solver will only look for solutions that have exactly $N^{t+1}$ nodes. This is not particularly inconvenient as it is easy to run the solver multiple times with different values of $N^{t+1}$ and select the solution with the minimum value of the objective function. Since we know the instantaneous load on each partition, we mathematically calculate the lower bound on the number of nodes $N^{t+1}$ required to handle the total load and run the solver to obtain a solution that minimizes data movement and load balance for this fixed $N^{t+1}$. We set a time out of 15 seconds for the solver. If a solution is not found within that time, we run the solver again, increasing the lower bound to $N^{t+1} = N^{t+1} + 1$. We repeat until a solution is found.

## Constraints

The constraints for the online optimizer are the same as the offline one, with minor modifications outlined below.

1. **Number of Active Nodes:** First, we add constraints to ensure that exactly $N^{t+1}$ nodes are used in the returned solution. This is not required in the offline optimizer as it determines the minimum number of nodes required as part of the objective function using $X_j^t$. For the online optimizer, there must be exactly $N^{t+1}$ active nodes in the returned solution.

$$\sum_{i=1}^{P} A_{ij}^{t+1} > 0 \quad \text{for } j = 1, \ldots, N^{t+1} \tag{5.7}$$

$$\sum_{i=1}^{P} A_{ij}^{t+1} = 0 \quad \text{for } j = N^{t+2}, \ldots, N \tag{5.8}$$

2. **CPU Capacity:** Second, the right hand side of the constraint for CPU capacity (Equation 5.3) is modified to remove $X_j^t$:

$$\sum_{i=1}^{P} A_{ij}^{t+1} * l_i^t < L * C \quad \text{for } j = 1, \ldots, N \tag{5.9}$$

## 5.5 Experimental Evaluation

### 5.5.1 Experimental Setup

#### System Configuration

All the nodes used in our experiments have identical hardware and software configuration. Each node is an IBM Blade Center server with two 2.4 GHz dual core CPUs and 8GB of physical memory. All nodes run OpenSUSE 11.3 (64-bit) with the Linux kernel version 2.6.34-12. The version of VoltDB that we use is 1.3.

#### Benchmark

For these experiments, we use a modified, cleanly partitionable version of the TPC-C benchmark [145]. VoltDB is intended for transactional applications in which the database fits in the total memory of all nodes in the cluster. A TPC-C database grows continuously as transactions run. Thus, without modification, a TPC-C workload is not suitable for VoltDB since the database will eventually grow beyond the memory limit. Because VoltDB can execute transactions very quickly, the memory limit will be reached very quickly – in less than 5 minutes in our server environment. Thus, to produce a transactional test application suited to VoltDB's memory constraint, we made several changes to TPC-C to avoid database growth. First, we removed the *History* table, which keeps track of payment history and thus continues to grow. Second, in the original TPC-C specifications, the *delivery* transaction removes rows from the *NewOrder* table. We modified the delivery transaction to remove corresponding rows from the *Orders* and *OrderLine* tables at the same time, to prevent these tables from growing. Finally, to make TPC-C cleanly partitionable by warehouse, we eliminated *new order* transactions that access remote warehouses.

### 5.5.2 Demonstrating Elastic Scale-out and Scale-in

In this section, we present experiments to illustrate that the mechanism that we implemented in VoltDB can be used to efficiently grow and shrink the size of the VoltDB cluster elastically, depending on the load.

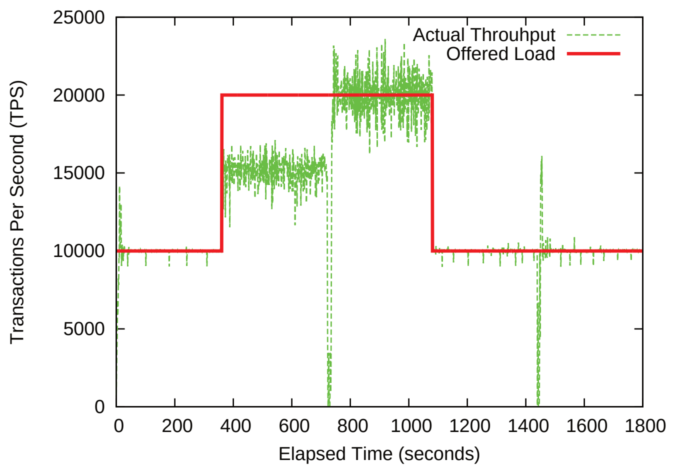


Figure 5.2: Cluster Throughput with Varying Load

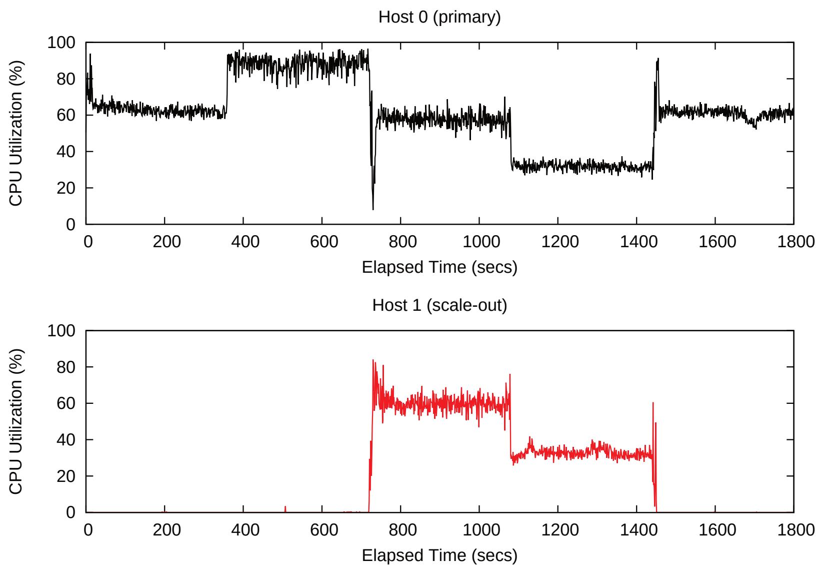


Figure 5.3: Per Host CPU Utilization with Varying Load

The first experiment that we present here was run with a total of three physical hosts. We have a total of 8 unique database partitions, and we run this experiment without $k$-safety. One of the three hosts is used to run the TPC-C benchmark clients and the other two are used as VoltDB servers. We start the VoltDB cluster with just one physical host that stores all eight partitions. We call this the *primary* host. We have one spare *scale-out* host which will be initially inactive. Clients connect to the VoltDB cluster (which is just one primary host) and submit TPC-C transactions. Clients in this experiment are *open loop* clients that submit the requests at a fixed rate, which we call the *offered load.* Offered load is varied during the experiment to vary the load on the VoltDB cluster. We run the test for a total of 30 minutes and measure the throughput of the cluster in terms of transactions per second (TPS). Our goal is to demonstrate that we can handle increases and decreases in the offered load by adding or removing hosts to the VoltDB cluster.

Figure 5.2 shows the offered load and the actual throughput of the cluster. Figure 5.3 shows the CPU utilization of each host (primary and scale-out) for the duration of the experiment. Initially, there is just one host, which is offered a fixed load of 10,000 requests/sec. The single host is able to service this load successfully, with about 60% CPU utilization (Figure 5.3). At about 400 seconds into the test, we increased the offered load to 20,000 requests/sec. A single host is not able to handle this load, and can provide only 15,000 TPS, as shown by the green line in Figure 5.2. We also see that the primary host (Host 0) is highly loaded now with about 90% CPU utilization (Figure 5.3). At about 750 seconds into the test, we do a scale-out operation by adding another host (Host 1) to the VoltDB cluster. Note that before this point, Host 1 was idle thus its CPU utilization was at 0%. During the scale-out operation, we move 4 of the 8 partitions from the primary host (Host 0) to the scale-out host (Host 1). The amount of data moved was about 1 GB. After the scale-out operation, the cluster is once again able to successfully service the offered load, now 20,000 TPS. We reduce the offered load back to 10,000 requests/second at around 1100 seconds. Now both hosts are very lightly loaded at around 30% CPU utilization. We perform a scale-in operation at around 1450 seconds, once again leaving only one node in the cluster that stores all partitions. The green line in Figure 5.2 shows that our scale-out and scale-in mechanisms have a minimal transient effect on throughput. When doing a scale-out or scale-in there is a small drop in throughput (as expected) but the drop lasts only for a very short time.

The second experiment, which is similar to the first, was run with a total of five physical hosts. Like before, one of the five hosts is used to run the TPC-C benchmark clients but now we have four hosts in the VoltDB cluster instead of two. We start the VoltDB cluster with just one physical host, the *primary* host. When the offered load is increased, the number of hosts in the VoltDB cluster is also increased, up to a total of four hosts (including the primary host). This approach ensures that the cluster is always adequately provisioned to handle the offered load, thus avoiding any overload conditions. Similarly, when the offered load is decreased, we decrease the number of hosts in the VoltDB cluster to avoid under-utilization

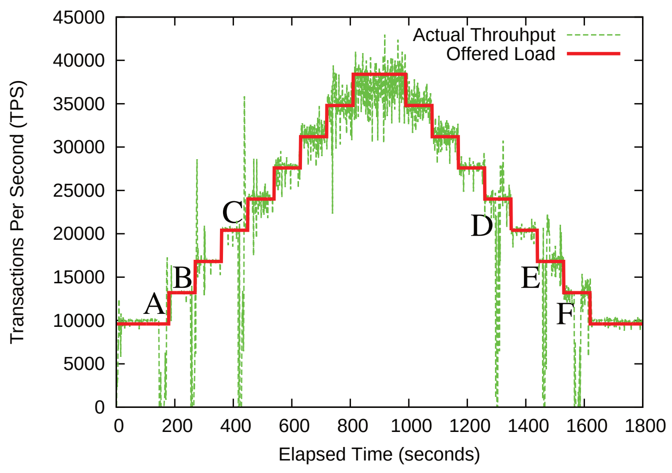


Figure 5.4: Cluster Throughput with Varying Load and Multiple Scale-out Operations

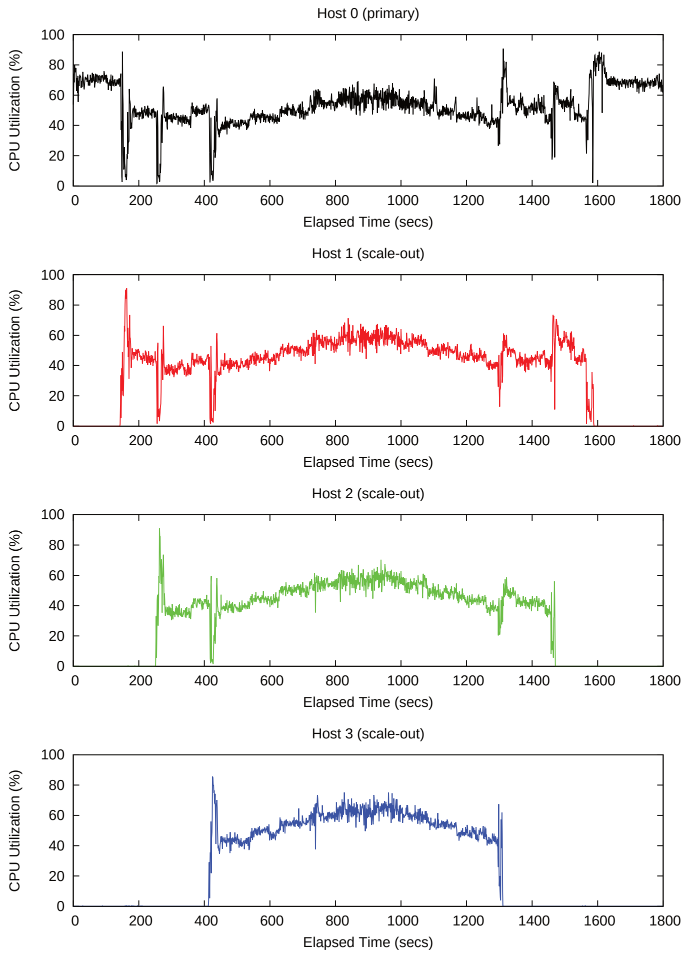


Figure 5.5: Per Host CPU Utilization with Varying Load and Multiple Scale-out Operations

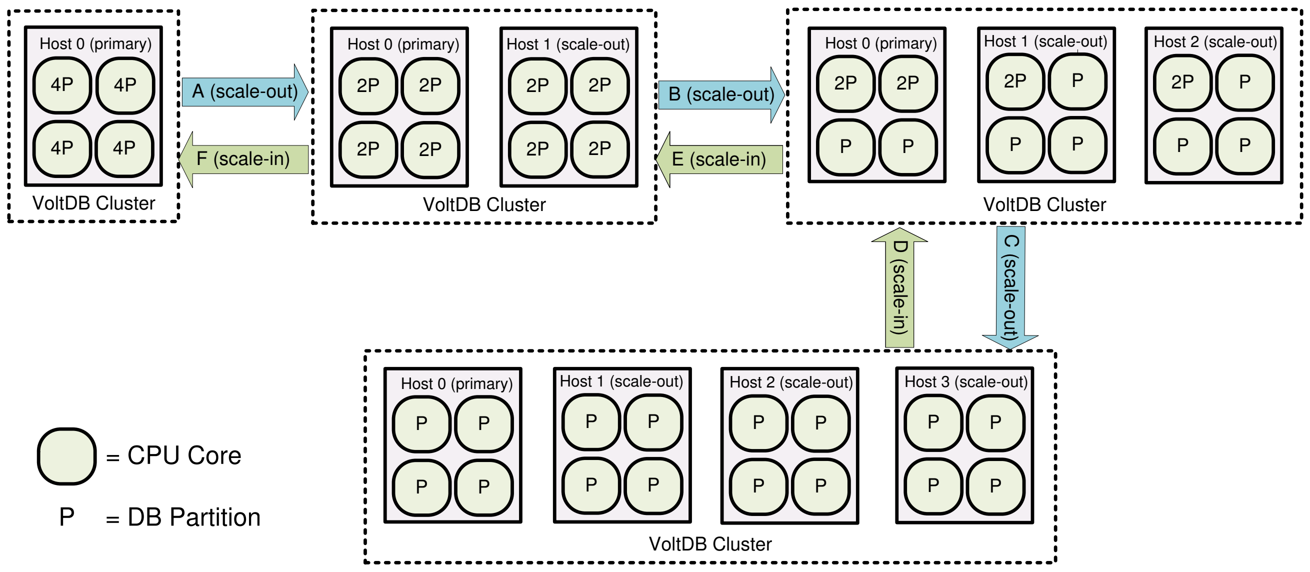


Figure 5.6: Scale-out Illustration

of resources. In total, we perform three scale-out operations at points A, B, and C in Figure 5.4, and we perform three scale-in operations at points D, E, and F. In addition to showing the offered load and scale-out/scale-in points, Figure 5.4 shows the throughput during this experiment. Figure 5.5 shows the CPU utilization of the different hosts over time. The experiment starts with 16 TPC-C partitions on one host, and through scale-out we get to four hosts, and then back to one host through scale-in. Figure 5.6 shows the mapping of partitions to hosts at different points in the experiment, along with the expected mapping of partitions to cores within a host. (The mapping of partitions to cores is managed by the operating system and not by VoltDB.) The total amount of data migrated during the scale-out operations is approximately 3GB.

Note that a single host in our system can handle up to 10,000 TPS. With a total of four hosts, the system is able to effectively handle approximately 40,000 TPS at peak load. As the offered load starts to subside around 1,000 seconds into the test, the cluster gradually becomes over-provisioned, so it is still able to handle the offered load even after the scale-in operations at points D, E, and F in Figure 5.4. Figure 5.5 shows that none of the hosts in the cluster is ever overloaded. In this experiment, we add and remove hosts at fixed pre-defined points that we know will match the pre-defined offered load. In a more realistic setting, a DBMS that is able to scale elastically would implement a controller that would automatically add or remove hosts from the cluster based on one of our partition placement algorithms (offline or online).

These experiments show that the changes we implemented in VoltDB provide an effective scale-out and scale-in mechanism that enables us to effectively handle varying load by dynamically growing and shrinking the size of the VoltDB cluster.

| **Parameter** | **Value** |
|---|---|
| Time Intervals ($T$) | 40 |
| Number of Partitions ($P$) | 16 |
| Replication Factor ($k$) | 1 |
| Cores Per Node ($C$) | 4 |
| Maximum Load Per Core ($L$) | 95% |
| Server Cost ($S$) | 1000 |
| Migration Cost ($I$) | 100 |

Table 5.1: Optimizer Parameters

| **Abbreviation** | **Distribution Name** | **Parameters** |
|---|---|---|
| UNIF | Uniform Distribution | N/A |
| TCATG | Temporally Skewed Categorical Distribution | $p = 0.4, 0.3, 0.2, 0.1$ |
| TNORM | Temporally Skewed Normal Distribution | $\mu = 4.0$ |
| TZIPF | Temporally Skewed Zipfian Distribution | $s = 0.5$ |

Table 5.2: Load Distributions

### 5.5.3 Effectiveness of Optimal Partition Placement

Having established the effectiveness of the scale-out mechanism, we now present experimental results that show the advantage of using our partition placement strategy. The goal of these experiments is to evaluate the effectiveness of our optimizers in achieving their optimization goals. We evaluate the optimal placement strategy calculated using our offline and online optimizers against a simple placement strategy of *greedy first fit*. Greedy first fit starts with a node, allocates partitions to it until its maximum capacity is reached, moves on to the next node, and so on, until there are no more partitions to be allocated. For our baseline, we use a fully scaled out system, i.e., one that uses the maximum number of nodes and the partition to node allocation is static (no elasticity). For example, given 16 partitions, the fully scaled out allocation will require 4 nodes with 4 cores each, statically allocating one partition per core. In terms of overall performance, we do not expect to do any better than such a fully scaled out system.

For these experiments, we use a total of 16 TPC-C partitions, distributed over a maximum of 4 nodes. The test was run for a total of 2 hours. For both offline and online optimizers, we recalculate the optimal placement every 3 minutes, i.e., $T = 40$. Table 5.1 provides a summary of parameters chosen for the offline optimizer. As opposed to the TPC-C benchmark specifications, we assume that the load on each database partition is not uniform. Our goal is to present a comparison of different partition placement strategies under different load distributions. A list of load distributions used in our experiments and their corresponding parameters is presented in Table 5.2. Note that the distributions are "temporally skewed" which means that the distribution of the skew varies with time. This is different from varying the skew. For example, for the normal distribution, we shift the mean of the distribution with time. The standard deviation or any other characteristics of the distribution do not change. We simply "slide" the distribution over all the partitions to ensure that all partitions experience high and low load. Also note that these distributions dictate how the total load at any given time is distributed to each partition. The overall load is varied according to a sine wave pattern with a frequency 4, as shown in Figure 5.7. We set the minimum load to be 10,000 TPS and the maximum load to be 52,000 TPS.

We want to evaluate the effectiveness of different placement strategies, with different load distributions in achieving the following goals: (1) providing good overall performance, (2) minimizing the amount of computing resources used, and (3) minimizing the amount of data movement. We use the following abbreviations to refer to the different placement strategies in our experiments: SCO – fully scaled out case, OFFLINE – offline optimizer, ONLINE – online optimizer, and GRD – greedy first fit.

#### Overall Throughput

Figure 5.8 presents the overall throughput achieved for each case. On the x-axis we have different load distributions, and on the y-axis we plot overall throughput in terms of transactions per second. As expected, for all load distributions, the fully scaled out case (SCO) with static partition allocation achieves the best performance. This is because SCO uses the maximum computing resources and does not involve any partition migration and thus no data movement. Also note that, for all placement strategies, throughput drops when going from a non-skewed load (uniform distribution) to a highly skewed load (Zipf distribution), which is to be expected. The throughput in case of SCO is better than the offline strategy (OFFLINE) by 12%, 6%, 2%, and 4% for UNIF, TCATG, TNORM, and TZIPF, respectively. The offline strategy (OFFLINE) is better than the online strategy (ONLINE) in terms of throughput by 3%, 5%, 8%, and 7% for UNIF, TCATG, TNORM, and TZIPF, respectively. The offline strategy (OFFLINE) is better than the simple greedy first fit strategy (GRD) by 22%, 20%, 20%, and 15% for UNIF, TCATG, TNORM, and TZIPF, respectively.

This experiment shows that in terms of overall performance, both the offline and online partition placement strategies perform fairly close to the fully scaled out case, and significantly better than the greedy first fit strategy. It also shows that by having exact knowledge about the entire future workload, we can gain some performance benefits by calculating the placement strategy for the entire planning period by using the offline optimizer, as compared to the case when we cannot predict the future load (online optimizer).

#### Computing Resources Consumed

We now evaluate the effectiveness of our placement strategies in minimizing the cost of running the system, i.e., minimizing the number of nodes used. One node running

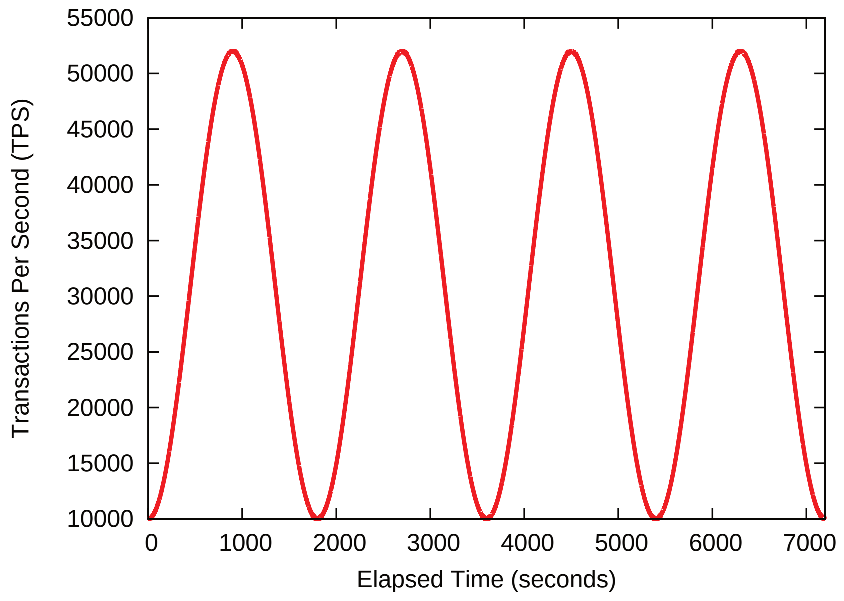


Figure 5.7: Overall Offered Load on the System

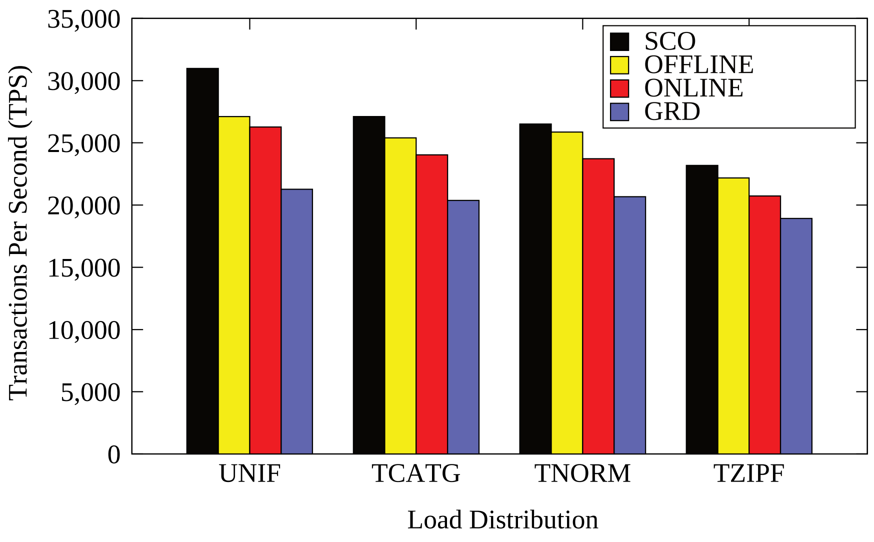


Figure 5.8: Overall Throughput

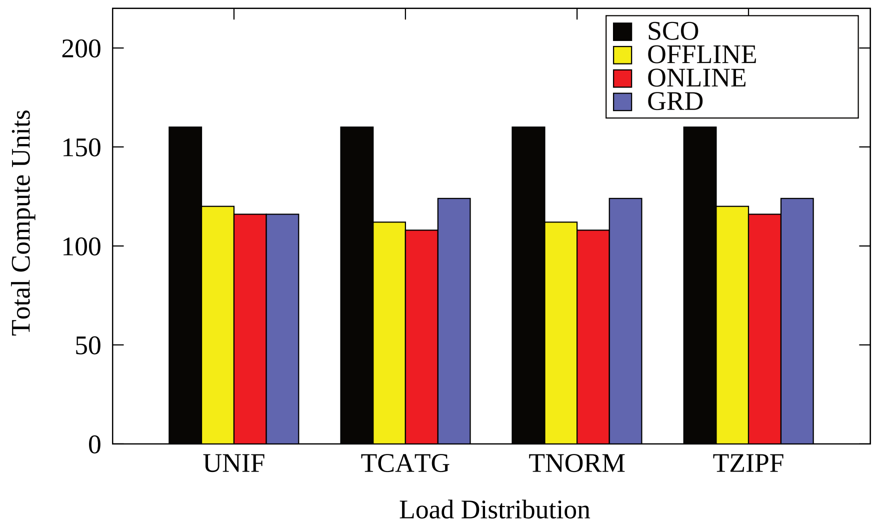


Figure 5.9: Compute Units Used

for one time interval is defined as a *compute unit*. Figure 5.9 shows the compute units summed over all time intervals, for different strategies and load distributions. The y-axis shows the total compute units used in each case. As expected, SCO uses the maximum compute units in all cases because it uses a static partition to node allocation with maximum number of nodes. All the other strategies (OFFLINE, ONLINE, and GRD), use a fairly similar amount of computing resources. One point to note from Figure 5.9 is that OFFLINE achieves slightly better throughput than ONLINE (as shown in Figure 5.8) at the expense of a few extra compute units.

This experiment shows that both OFFLINE and ONLINE achieve their goal of minimizing total computing resources used as compared to SCO. By taking advantage of elasticity, we can ensure that the performance goals are met while minimizing computing resources.

### Data Movement

Lastly, Figures 5.10 and 5.11 present the total number of partitions migrated and the total amount of data moved between the nodes of the VoltDB cluster as a result of these migrations. These figures do not include the fully scaled out case (SCO) because there are no partition migrations and hence no data movement in that case. Focusing on the results of Figure 5.11, we note that for all load distributions, OFFLINE performs the least amount of data movement, an average of 63% and 286% less data movement than ONLINE and GRD, respectively.

From the results of these experiments, we conclude that both the offline and online optimizers are able to meet their optimization objectives in terms of maximizing performance and minimizing server and migration cost over a simple placement strategy. If we know the entire workload in advance, we can use the offline opti-

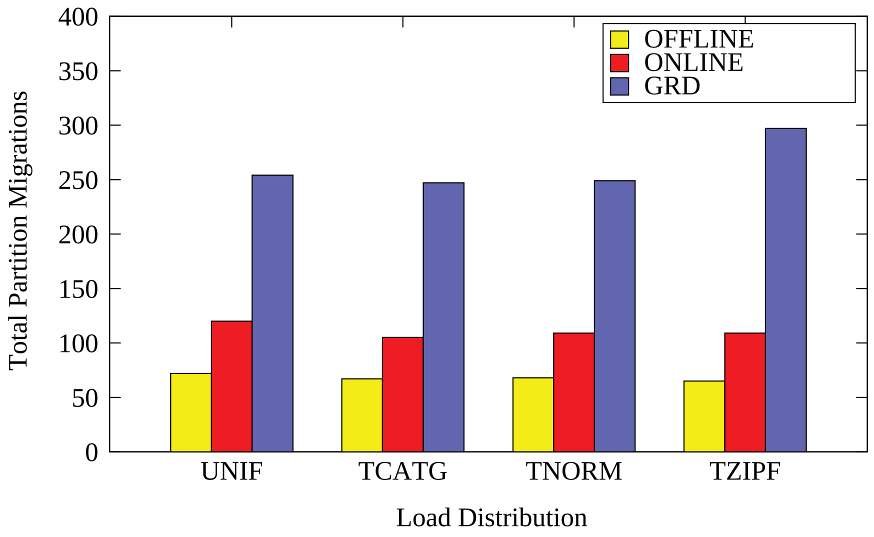


Figure 5.10: Number of Partitions Migrated

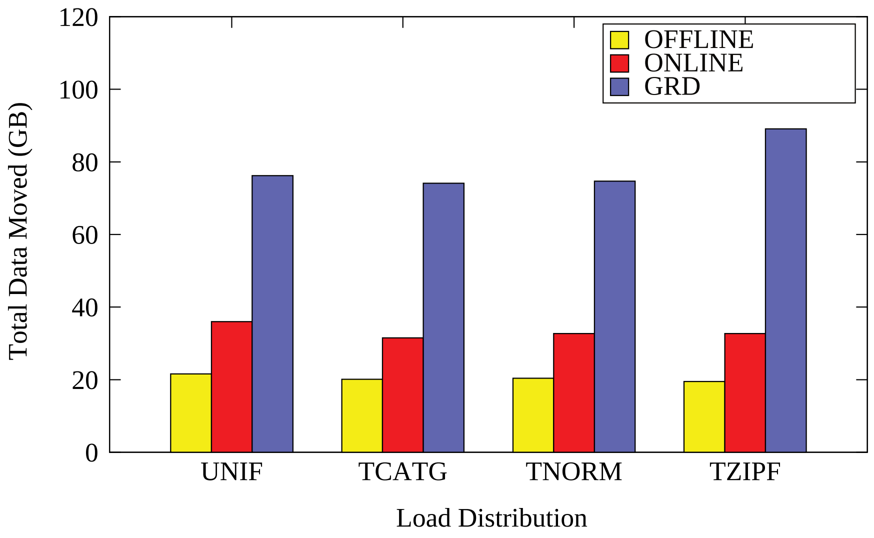


Figure 5.11: Total Data Moved

mizer to gain slight performance benefits over the online optimizer, and can save significant migration cost, at the expense of some extra computing resources. Except for the differences in data movement, both the offline and online optimizers perform adequately. As shown in the next section, the low complexity of the online optimizer and its ability to find an optimal placement in a very short amount of time will warrant its use in most practical situations.

In this section, we presented overall results only. More detailed results under uniform and normal load distributions, showing per host CPU utilization, offered load vs. overall throughput, and number of nodes used, can be found in Appendix A.

### 5.5.4 Scalability of the Optimizers

The goal of this section is to evaluate the scalability of the offline and online optimizers with respect to the problem size. Ideally, the optimizer should always return the *optimal* solution within a very short amount of time. For the online optimizer, the time to find a solution is critical for it to be applicable in live partition migration scenarios. On the other hand, as the name implies, we expect the offline optimizer to be useful for scenarios where we have plenty of time to find a partition placement schedule for a long-running, periodic workload. Nevertheless, the offline optimizer should still be able to return the optimal (or close to the optimal) solution in a reasonable amount of time.

We evaluate the online optimizer by varying the number of partitions ($P$) from 16 to 512 (results not presented here). For $P \leq 256$, the optimizer always returns the optimal solution within a few seconds. In the worst case, for $P = 512$, the online optimizer returns the optimal solution within 20 seconds. This shows that even for large problem sizes, the online optimizer returns a solution fairly quickly, and is thus adequate for live partition migration scenarios.

In contrast to the online optimizer, the problem formulation for the offline optimizer is much more complex in terms of the number of decision variables. This additional complexity arises from the fact that the offline optimizer has to take into account another dimension (i.e., time) for each decision variable. Consequently, the number of decision variables increase exponentially with the problem size. For relatively small problem sizes ($T = 15$ and $P \leq 16$), we are able to obtain the *optimal* solution in less than a second. For larger problem sizes ($P \geq 32$), we are not able to obtain the optimal solution in a reasonable amount of time. The CPLEX solver allows us to specify a time limit for the solution, and reports the best solution obtained within this time limit plus a bound on how far this solution is from the optimal solution (details in the next paragraph). For large problem sizes, we take advantage of this feature of CPLEX and report the solution it obtains in a fixed amount of time equal to 8 hours. We also report how far the reported solution is from the optimal solution.

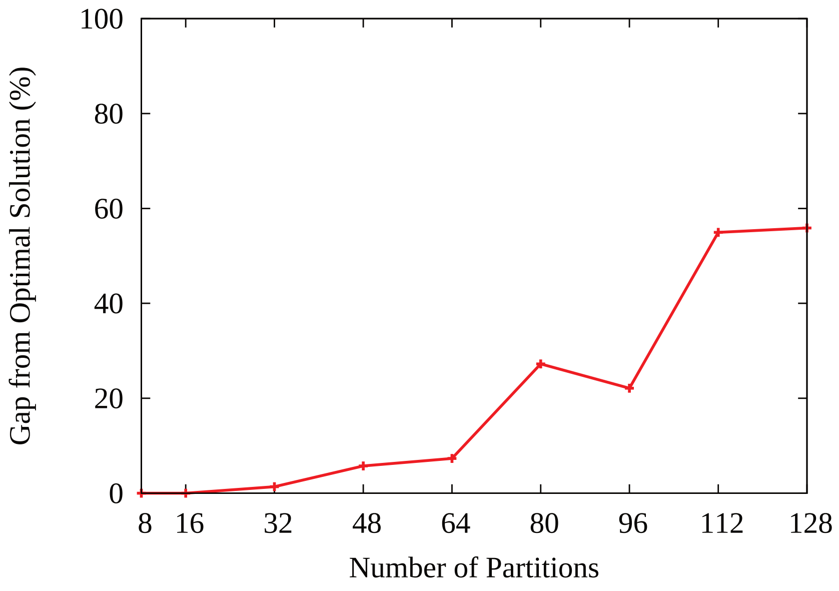


Figure 5.12: Scalability of Offline Optimizer

In Figure 5.12, we present results that show how the offline problem formulation scales with the problem size. We fix the planning period to 15 time intervals, i.e., $T = 15$. The load on partitions is distributed according to a temporally skewed normal distribution. In Figure 5.12, we vary the number of partitions from 8 to 128 on the x-axis. The gap between the actual optimal and the solution found by our optimizer is plotted on the y-axis. The CPLEX optimizer first solves a linear programming (LP) relaxation of the mixed-integer program, where decision variables can take real values. The value of the objective function found with LP relaxation is the *optimal*. The reported gap is the difference (in percentage) between the optimal value of the objective function and the best solution found by the optimizer so far that satisfies the integer constraints. Thus, it is guaranteed that the solution to the mixed-integer program will be no worst than "gap %" from the actual optimal solution (of the LP relaxation). From Figure 5.12, we can see that the offline optimizer always finds a solution that is within 30% of the optimal for relatively large problem sizes ($P \leq 96$). For even larger problem sizes, the solution found is within 60% of the optimal.

Note that the above results were obtained after we made several modifications to the original problem formulation for the offline optimizer to improve scalability. One such improvement has to do with how general purpose solvers, like CPLEX, deal with symmetry in the problem formulation. Many of these solvers use traditional *branch and bound algorithms* to explore the search space. It is well known that these algorithms perform poorly on problems that have a lot of symmetry [27, Chapter 3]. As mentioned earlier, we assume that all the nodes in our environment are homogeneous. This assumption introduces a lot of symmetry in our problem formulation. In order to break that symmetry, we added some additional constraints to the offline problem formulation that improved its scalability. For example, we included a constraint that specifies that host *j+1* can only be used if host $j$ is already used. Even with these changes, admittedly, the offline optimizer is limited in its scalability to very large problem sizes. Exploring advanced techniques available in the mathematical optimization literature to improve the scalability of the offline optimizer is an interesting direction for future work.

## 5.6 Summary

A self-managing database system should be able to automatically and elastically grow and shrink the computing resources that it uses in response to variations in load. Database systems today do not support this kind of elastic scale-out and scale-in due to the challenges of maintaining consistency of the database while elastically scaling. In this chapter, we presented a solution for database scale-out using a shared nothing partition-based parallel database system, namely VoltDB. We presented the required changes that need to be incorporated in a system like VoltDB to make it elastically scalable. More specifically, we present mechanisms to dynamically add more nodes to an existing VoltDB cluster and to move partitions from existing nodes to the new nodes in an efficient manner.

In the second half of the chapter, we focus on the research problem of partition placement with elastic scale-out. We presented a mixed-integer linear program (MILP) formulation of the problem, showed how to use a general purpose solver to find a solution, and presented experimental results that showed that our optimizer is able to provide performance benefits and cost savings over a simple placement strategy. We believe that a solution implemented using the techniques presented in this chapter can provide effective elastic scaling for partitioned database systems.

# Chapter 6

# Conclusion

Cloud computing is rapidly changing the computing landscape by fundamentally revolutionizing the way computing is viewed and delivered to the end user. Computing is now treated as a "utility" that can be provisioned on-demand, in a cost-effective manner. Cloud computing presents a paradigm shift in how we provision and manage computing resources as well as how we develop, deploy, and use software applications. We need to devise novel system architectures that can meet the computing demands of the next generation of applications. Existing applications might need to be altered or might need to be redesigned altogether from the ground up for deploying in the cloud. In this thesis, we consider database management system as one such application, and explore how these systems can be adapted for the cloud computing platforms of today.

Database management systems are an important class of applications that are frequently deployed in the cloud. In order to provide services to end users, most applications need to store and retrieve persistent data, sometimes in large quantities. Moreover, this data storage and retrieval needs to be done in a scalable, reliable, and efficient manner. Relational database systems are a very popular choice for storing data for such applications. Relational database systems provide ACID transactions, which guarantee data consistency, and allow users to access and manipulate their data using SQL, which is popular and simplifies application development. However, the data consistency guarantees provided by relational database systems significantly limit the high availability and scalability of these systems, which presents a major challenge in deploying them in the cloud.

This thesis focuses on exploiting existing cloud technologies (e.g., virtualization), and the entire cloud computing ecosystem in general, to provide transparent high availability and elastic scalability for database systems. To that extent, we have presented three different systems that achieve these goals.

In Chapter 3, we presented a system called RemusDB. The goal of RemusDB is to provide simple and cheap high availability for database systems in the cloud. RemusDB is a reliable, cost-effective, active/standby high availability solution implemented at the virtualization layer. RemusDB offers high availability as a service provided by the underlying virtualization infrastructure to any database system running over it, which fits well with the service oriented computing model of the cloud. It requires little or no modification to the database system and imposes minimal performance overhead.

The second system that we presented, in Chapter 4, is called Chimera – a hybrid of a shared nothing and data sharing system. The goals of Chimera are to provide elastic scale-out and flexibility in load balancing for time varying workloads in dynamic environments such as the cloud. By combining the scalability of a shared nothing system and the flexibility of a data sharing system, Chimera is able to effectively achieve its goal for read-mostly workloads. We believe that this work is an important step towards erasing the hard boundaries between the shared nothing and data sharing system architectures, thus paving the way for novel architectures that are more relevant in today's computing environments.

The third system, presented in Chapter 5, proposes the use of a partition-based parallel shared-nothing database system, namely VoltDB, as a starting point to provide elastic scale-out for database systems. We have implemented new mechanisms in VoltDB that allow us to dynamically and efficiently move database partitions to nodes (virtual or physical) that can be added or removed in response to time varying workloads. Furthermore, by using this platform we address the research problem of partition placement. We formulate the partition placement problem as a mathematical optimization problem, and we show how a general purpose solver can be used to find a solution. We present quantitative results that demonstrate the effectiveness of our partition placement strategy in achieving good performance while reducing both system and data migration costs. The elastic scale-out mechanism, coupled with the partition placement optimizer, presents an important step towards building a self-managing, elastically scalable database system suitable for cloud computing environments.

There are many avenues for future research in this general area of improving the usability and scalability of database systems deployed in the cloud. As noted above, we believe that cloud computing presents a paradigm shift in how we view computing. Many applications welcome this shift and can be readily deployed on the cloud, making full use of its capabilities including dynamic resource provisioning, fault tolerance, and elastic scalability. However, existing relational database systems are not ideally suited for deployment in the cloud. We need to make fundamental design changes to make them "cloud friendly". We have presented some of these changes for making existing database systems highly-available and elastically scalable. However, this thesis presents only a first step towards implementing a transactional database service in the cloud. We believe that our work opens up many opportunities for new and interesting research in the areas of database systems and cloud computing.

# References

[1] Ashraf Aboulnaga, Kenneth Salem, Ahmed A. Soror, Umar Farooq Minhas, Peter Kokosielis, and Sunil Kamath. Deploying database appliances in the cloud. *IEEE Data Engineering Bulletin*, 32(1), 2009. 11, 28

[2] Sanjay Agrawal, Vivek R. Narasayya, and Beverly Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *International Conference on Management of Data (SIGMOD)*, 2004. 24

[3] Mumtaz Ahmad, Ashraf Aboulnaga, Shivnath Babu, and Kamesh Munagala. Interaction-aware scheduling of report generation workloads. *Very Large Data Bases Journal (VLDBJ)*, 20(4), 2011. 58

[4] Mumtaz Ahmad, Songyun Duan, Ashraf Aboulnaga, and Shivnath Babu. Predicting completion times of batch query workloads using interaction-aware models and simulation. In *Conference on Extending Database Technology (EDBT)*, 2011. 58

[5] Gautam Altekar and Ion Stoica. ODR: Output-deterministic replay for multicore debugging. In *Symposium on Operating System Principles (SOSP)*, 2009. 32

[6] Mehmet Altinel, Christof Bornhövd, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, and Berthold Reinwald. Cache Tables: Paving the way for an adaptive database cache. In *International Conference on Very Large Data Bases (VLDB)*, 2003. 24, 25

[7] Amazon Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2/. 8, 10, 99

[8] Amazon Relational Database Service (RDS). http://aws.amazon.com/rds/. 4

[9] Khalil Amiri, Sanghyun Park, Renu Tewari, and Sriram Padmanabhan. Dbproxy: A dynamic data cache for web applications. In *International Conference on Data Engineering (ICDE)*, 2003. 24, 25

[10] Cristiana Amza, Alan L. Cox, and Willy Zwaenepoel. Conflict-aware scheduling for dynamic content applications. In *USENIX Symposium on Internet Technologies and Systems*, 2003. 21, 22

[11] Cristiana Amza, Alan L. Cox, and Willy Zwaenepoel. Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites. In *International Middleware Conference*, 2003. 21, 22, 99, 101

[12] Cristiana Amza, Alan L. Cox, and Willy Zwaenepoel. A comparative evaluation of transparent scaling techniques for dynamic content servers. In *International Conference on Data Engineering (ICDE)*, 2005. 21, 22, 24

[13] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. *Transactions on Computer Systems (TOCS)*, 14(1), 1996. 79

[14] Todd Anderson, Yuri Breitbart, Henry F. Korth, and Avishai Wool. Replication, consistency, and practicality: are these mutually exclusive? In *International Conference on Management of Data (SIGMOD)*, 1998. 14, 21, 22, 99, 101

[15] Peter M. G. Apers. Data allocation in distributed database systems. *Transactions on Database Systems (TODS)*, 13(3), 1988. 102

[16] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Communications of ACM (CACM)*, 53(4), 2010. 2, 100, 107

[17] Mary Baker and Mark Sullivan. The Recovery Box: Using fast recovery to provide high availability in the UNIX environment. In *USENIX Conference*, 1992. 37

[18] Paul T. Barham, Boris Dragovic, Keir Fraser, Steven Hand, Timothy L. Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Symposium on Operating System Principles (SOSP)*, 2003. 9, 19, 20, 28

[19] Philip A. Bernstein, Alan Fekete, Hongfei Guo, Raghu Ramakrishnan, and Pradeep Tamma. Relaxed-currency serializability for middle-tier caching and replication. In *International Conference on Management of Data (SIGMOD)*, 2006. 25

[20] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987. 13, 16, 21

[21] Martin Bichler, Thomas Setzer, and Benjamin Speitkamp. Capacity planning for virtualized servers. In *Workshop on Information Technologies and Systems (WITS)*, 2006. 102

[22] Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, and Tim Kraska. Building a database on S3. In *International Conference on Management of Data (SIGMOD)*, 2008. 97

[23] Yuri Breitbart, Raghavan Komondoor, Rajeev Rastogi, S. Seshadri, and Abraham Silberschatz. Update propagation protocols for replicated databases. In *International Conference on Management of Data (SIGMOD)*, 1999. 14, 21, 101

[24] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault-tolerance. In *Symposium on Operating System Principles (SOSP)*, 1995. 28, 32, 34

[25] Paolo Bruni, Roy Cornford, Rafael Garcia, Sabine Kaschta, and Ravi Kumar. *DB2 9 for z/OS Technical Overview*. IBM Redbooks, 2007. 14, 15, 25, 76

[26] Anna Brunstrom, Scott T. Leutenegger, and Rahul Simha. Experimental evaluation of dynamic data allocation strategies in a distributed database with changing workloads. In *Conference on Information and Knowledge Management (CIKM)*, 1995. 24, 101, 102

[27] Edmund K. Burke and Graham Kendall. *Search methodologies: introductory tutorials in optimization and decision support techniques*. Springer, 2005. 124

[28] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2006. 78, 80, 84, 87, 97

[29] Emmanuel Cecchet, George Candea, and Anastasia Ailamaki. Middleware-based database replication: The gaps between theory and practice. In *International Conference on Management of Data (SIGMOD)*, 2008. 22

[30] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. C-JDBC: Flexible database clustering middleware. In *USENIX Conference*, 2004. 22

[31] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *Transactions on Computer Systems (TOCS)*, 26, 2008. 3, 84, 97, 101

[32] Whei-Jen Chen, Masafumi Otsuki, Paul Descovich, Selvaprabhu Arumuggharaj, Toshihiko Kubo, and Yong Jun Bi. *High Availability and Disaster Recovery Options for DB2 on Linux, Unix, and Windows*. IBM Redbooks, 2009. 31

[33] Parvathi Chundi, Daniel J. Rosenkrantz, and S. S. Ravi. Deferred updates and data placement in distributed databases. In *International Conference on Data Engineering (ICDE)*, 1996. 14, 21, 101

[34] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2005. 16, 29, 36

[35] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. In *International Conference on Very Large Data Bases (VLDB)*, 2008. 97

[36] George P. Copeland, William Alexander, Ellen E. Boughter, and Tom W. Keller. Data placement in bubba. In *International Conference on Management of Data (SIGMOD)*, 1988. 23, 24, 99, 101, 102

[37] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High availability via asynchronous virtual machine replication. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2008. 4, 16, 18, 20, 21, 28, 30

[38] Carlo Curino, Evan Jones, Raluca Popa, Eugene Malviya, Nirmesh Wu, Sam Madden, Har Balakrishnan, and Nickolai Zeldovich. Relational Cloud: a database service for the cloud. In *Conference on Innovative Data Systems Research (CIDR)*, 2011. 101

[39] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment (PVLDB)*, 3(1-2), 2010. 99, 101

[40] Carlo Curino, Evan P.C. Jones, Samuel Madden, and Hari Balakrishnan. Workload-aware database monitoring and consolidation. In *International Conference on Management of Data (SIGMOD)*, 2011. 99, 101, 102

[41] Shaul Dar, Michael J. Franklin, Björn T. Jónsson, Divesh Srivastava, and Michael Tan. Semantic data caching and replacement. In *International Conference on Very Large Data Bases (VLDB)*, 1996. 24

[42] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. ElasTraS: an elastic transactional data store in the cloud. In *HotCloud*, 2009. 101

[43] Anindya Datta, Kaushik Dutta, Helen M. Thomas, Debra E. VanderMeer, Krithi Ramamritham, and Dan Fishman. A comparative study of alternative middle tier caching solutions to support dynamic web content acceleration. In *International Conference on Very Large Data Bases (VLDB)*, 2001. 25

[44] Better Business Protection Through Virtualization. http://www.dell.com/downloads/global/power/ps4q06-20070169-Ziff.pdf. 2

[45] Murthy Devarakonda, Bill Kish, and Ajay Mohindra. Recovery in the Calypso file system. *Transactions on Computer Systems (TOCS)*, 14(3), 1996. 79, 80

[46] David J. DeWitt and Jim Gray. Parallel database systems: The future of high performance database systems. *Commununications of the ACM (CACM)*, 35(6), 1992. 23, 101

[47] Distributed Replicated Block Device (DRBD). http://www.drbd.org/. 42, 47

[48] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2002. 32

[49] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. Execution replay of multiprocessor virtual machines. In *Virtual Execution Environments (VEE)*, 2008. 32

[50] everRun VM. http://www.marathontechnologies.com/high_availability_xen server.html. 16, 19

[51] Derrell V. Foster, Lawrence W. Dowdy, and James E. Ames IV. File assignment in a computer network. *Computer Networks*, 5, 1981. 102

[52] Prashant Gaharwar. Dynamic storage provisioning with SLO guarantees. Master's thesis, University of Waterloo, 2010. 107

[53] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Symposium on Operating Systems Principles (SOSP)*, October 2003. 84

[54] David K. Gifford. Weighted voting for replicated data. In *Symposium on Operating System Principles (SOSP)*, 1979. 32

[55] Andrew C. Goldstein. The design and implementation of a distributed file system. *Digital Technical Journal*, 1(5), 1987. 79

[56] Google App Engine. http://cloud.google.com/products/index.html. 8

[57] Google Cloud SQL. https://developers.google.com/cloud-sql/docs/introduction. 4

[58] Google Docs. https://docs.google.com/. 8

[59] Cary Gray and David Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Symposium on Operating Systems Principles (SOSP)*, 1989. 80

[60] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *International Conference on Management of Data (SIGMOD)*, 1996. 13, 14, 21, 32, 101

[61] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993. 32

[62] Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Symposium on Operating System Principles (SOSP)*, 2007. 3, 101

[63] Christopher R. Hertel. *Implementing CIFS: The Common Internet File System*, chapter Introduction. Prentice Hall, 2003. 78, 83, 85

[64] Charles R. Hicks and K. V. Turner. *Fundamental Concepts in the Design of Experiments*. Oxford University Press, 1999. 68

[65] JoAnne Holliday, Divyakant Agrawal, and Amr El Abbadi. The performance of database replication with group multicast. In *Fault-Tolerant Computing (FTCS)*, 1999. 22

[66] Hui-I Hsiao and David J. DeWitt. A performance study of three high available data replication strategies. *Distributed and Parallel Databases (DAPD)*, 1(1), 1993. 13

[67] Kien A. Hua and Chiang Lee. An adaptive data placement scheme for parallel database computer systems. In *International Conference on Very Large Data Bases (VLDB)*, 1990. 102

[68] George Huey. Scaling Out with SQL Azure Federation. http://msdn.microsoft.com/en-us/magazine/hh848258.aspx. 105

[69] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *USENIX Conference*, 2010. 84, 87, 97

[70] IBM ILOG CPLEX Optimization Studio. http://www-01.ibm.com/software/integration/optimization/cplex-optimization-studio/. 100, 110

[71] The Costs of Downtime: North American Medium Businesses 2006. http://www.infonetics.com/pr/2006/upna06.dwn.nr.asp. 2

[72] Java TPC-W implementation, PHARM group, University of Wisconsin, 1999. http://www.ece.wisc.edu/pharm/tpcw/. 43

[73] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-Store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment (PVLDB)*, 1(2), 2008. 102

[74] Bettina Kemme and Gustavo Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *International Conference on Very Large Data Bases (VLDB)*, 2000. 21, 22, 32, 99, 101

[75] Bettina Kemme and Gustavo Alonso. A new approach to developing and implementing eager database replication protocols. *Transactions on Database Systems (TODS)*, 25(3), 2000. 22

[76] Bettina Kemme, Fernando Pedone, Gustavo Alonso, André Schiper, and Matthias Wiesmann. Using optimistic atomic broadcast in transaction processing systems. *Transactions on Knowledge and Data Engineering (TKDE)*, 15(4), 2003. 22

[77] Sami Khuri, Thomas Bäck, and Jörg Heitkötter. The zero/one multiple knapsack problem and genetic algorithms. In *Symposium on Applied Computing (SAC)*, 1994. 102

[78] Darmadi Komo. *Microsoft SQL Server 2008 R2 High Availability Technologies White Paper*. Microsoft, 2010. 31

[79] Donald Kossmann, Tim Kraska, and Simon Loesing. An evaluation of alternative architectures for transaction processing in the cloud. In *International Conference on Management of Data (SIGMOD)*, 2010. 25, 97

[80] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *Operating Systems Review*, 44, 2010. 3, 101

[81] Leslie Lamport. The part-time parliament. *Transactions on Computer Systems (TOCS)*, 16(2), 1998. 80

[82] Per-Åke Larson, Jonathan Goldstein, and Jingren Zhou. MTCache: Transparent mid-tier database caching in SQL Server. In *International Conference on Data Engineering (ICDE)*, 2004. 24, 25

[83] Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter M. Chen, and Jason Flinn. Respec: Efficient online multiprocessor replay via speculation and external determinism. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010. 32

[84] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed virtual disks. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1996. 82

[85] Linux-HA Project, 1999. http://www.linux-ha.org/doc/. 46

[86] Diego R. Llanos. TPCC-UVa: An open-source TPC-C implementation for global performance measurement of computer systems. *SIGMOD Record*, 35(4), 2006. 43

[87] David Lomet. Recovery for shared disk systems using multiple redo logs. Technical Report 4, Digital Cambridge Research Lab, 1990. 81

[88] David Lomet. Private locking and distributed cache management. In *International Conference on Parallel and Distributed Information Systems (PDIS)*, 1994. 84

[89] David Lomet, Rick Anderson, T. K. Rengarajan, and Peter Spiro. How the Rdb/VMS data sharing system became fast. Technical Report 2, Digital Cambridge Research Lab, 1992. 80

[90] Kevin Loney. *Oracle Database 11g The Complete Reference*. McGraw-Hill, 2008. 76, 97

[91] Qiong Luo, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, Honguk Woo, Bruce G. Lindsay, and Jeffrey F. Naughton. Middle-tier database caching for e-business. In *International Conference on Management of Data (SIGMOD)*, 2002. 25

[92] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2004. 78, 80, 84, 87

[93] Manish Mehta and David J. DeWitt. Data placement in shared-nothing parallel database systems. *Very Large Data Bases Journal (VLDBJ)*, 6(1), 1997. 23, 99, 101, 102

[94] Roman B. Melnyk and Paul C. Zikopoulos. *DB2: The Complete Reference*. McGraw-Hill, 2001. 14, 15, 23, 76, 97, 100

[95] Memcached. http://memcached.org/. 24

[96] Microsoft Hyper-V. http://www.microsoft.com/windowsserver2008/en/us/hyperv-main.aspx. 16, 19

[97] Microsoft SQL Database. http://www.windowsazure.com/en-us/home/features/data-management/. 4

[98] Jesús M. Milán-Franco, Ricardo Jiménez-Peris, Marta Patiño-Martínez, and Bettina Kemme. Adaptive middleware for data replication. In *International Middleware Conference*, 2004. 22

[99] Umar Farooq Minhas, Rui Liu, Ashraf Aboulnaga, Kenneth Salem, Jonathan Ng, and Sean Robertson. Elastic scale-out for partition-based database systems. In *Workshop on Self Managing Database Systems (SMDB)*, 2012. 100

[100] Umar Farooq Minhas, David Lomet, and Chandramohan A. Thekkath. Chimera: data sharing flexibility, shared nothing simplicity. In *International Database Engineering and Applications Symposium (IDEAS)*, 2011. 77

[101] Umar Farooq Minhas, Shriram Rajagopalan, Brendan Cully, Ashraf Aboulnaga, Kenneth Salem, and Andrew Warfield. RemusDB: Transparent high availability for database systems. *Proceedings of the VLDB Endowment (PVLDB)*, 4(11), 2011. 28, 31

[102] Umar Farooq Minhas, Shriram Rajagopalan, Brendan Cully, Ashraf Aboulnaga, Kenneth Salem, and Andrew Warfield. RemusDB: Transparent high availability for database systems. *Very Large Data Bases Journal (VLDBJ)*, 2013. 28, 31

[103] Umar Farooq Minhas, Jitendra Yadav, Ashraf Aboulnaga, and Kenneth Salem. Database systems on virtual machines: How much do you lose? In *Workshop on Self Managing Database Systems (SMDB)*, 2008. 11

[104] 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. *Transactions on Database Systems (TODS)*, 17(1), 1992. 31, 44

[105] C. Mohan and Inderpal Narang. Efficient locking and caching of data in the multisystem shard disks transaction environment. In *Conference on Extending Database Technology (EDBT)*, 1992. 81

[106] Microsoft SQL Server 2008. http://www.microsoft.com/sqlserver/2008/. 14, 15, 23, 76, 97, 100

[107] MySQL Cluster. http://www.mysql.com/products/database/cluster/. 14, 15, 23, 100

[108] MySQL Cluster 7.0 and 7.1: Architecture and new features. A MySQL Technical White Paper by Oracle, 2010. 31

[109] Oracle. *Oracle Data Guard Concepts and Administration*, 11g release 1 edition, 2008. 31

[110] Oracle. *Oracle Real Application Clusters 11g Release 2*. Oracle, 2009. 14, 15, 25, 31, 83

[111] Oracle. *MySQL 5.0 Reference Manual*, 2010. Revision 23486, http://dev.mysql.com/doc/refman/5.0/en/. 46

[112] Esther Pacitti, Pascale Minet, and Eric Simon. Fast algorithms for maintaining replica consistency in lazy master replicated databases. In *International Conference on Very Large Data Bases (VLDB)*, 1999. 14, 21, 101

[113] Stratos Papadomanolakis and Anastassia Ailamaki. Autopart: Automating schema design for large scientific databases using data partitioning. In *Conference on Scientfic and Statistical Data Base Management (SSDB)*, 2004. 24

[114] Marta Patiño-Martínez, Ricardo Jiménez-Peris, Bettina Kemme, and Gustavo Alonso. MIDDLE-R: Consistent database replication at the middleware level. *Transactions on Computer Systems (TOCS)*, 23(4), 2005. 22

[115] David A. Patterson, Garth A. Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *International Conference on Management of Data (SIGMOD)*, 1988. 12

[116] Fernando Pedone and Svend Frølund. Pronto: High availability for standard off-the-shelf databases. *Journal of Parallel Distributed Computing (PDC)*, 68(2), 2008. 22

[117] Percona Tools TPC-C MySQL Benchmark, 2008. https://code.launchpad.net/ percona-dev/perconatools/tpcc-mysql. 43

[118] Francisco Perez-Sorrosal, Marta Patiño-Martínez, Ricardo Jiménez-Peris, and Bettina Kemme. Consistent and scalable cache replication for multi-tier J2EE applications. In *International Middleware Conference*, 2007. 25

[119] Christian Plattner and Gustavo Alonso. Ganymed: Scalable replication for transactional web applications. In *International Middleware Conference*, 2004. 22

[120] Christos A. Polyzois and Hector Garcia-Molina. Evaluation of remote backup algorithms for transaction processing systems. In *International Conference on Management of Data (SIGMOD)*, 1992. 32

[121] The Rackspace Open Cloud. http://www.rackspace.com/cloud/. 8

[122] Shriram Rajagopalan, Brendan Cully, Ryan O'Connor, and Andrew Warfield. SecondSite: Disaster tolerance as a service. In *Virtual Execution Environments (VEE)*, 2012. 42

[123] T. Rengarajan, Peter Spiro, and William Wright. High availability mechanisms of VAX DBMS software. *Digital Technical Journal*, 1(8), 1989. 81

[124] Pedro I. Rivera-Vega, Ravi Varadarajan, and Shamkant B. Navathe. Scheduling data redistribution in distributed databases. In *International Conference on Data Engineering (ICDE)*, 1990. 24, 101

[125] Uwe Röhm, Klemens Böhm, Hans-Jörg Schek, and Heiko Schuldt. FAS - A freshness-sensitive coordination middleware for a cluster of OLAP components. In *International Conference on Very Large Data Bases (VLDB)*, 2002. 22

[126] Jerry Rolia, Artur Andrzejak, and Martin Arlitt. Automating enterprise application placement in resource utilities. In *Self-Managing Distributed Systems*, volume 2867 of *Lecture Notes in Computer Science*. Springer, 2003. 102

[127] Domenico Saccà and Gio Wiederhold. Database partitioning in a cluster of processors. In *International Conference on Very Large Data Bases (VLDB)*, 1983. 102

[128] salesforce.com. https://www.salesforce.com/. 8

[129] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun network file system. In *UNIX Conference*, 1985. 78, 83

[130] Daniel J. Scales, Mike Nelson, and Ganesh Venkitachalam. The design and evaluation of a practical system for fault-tolerant virtual machines. Technical Report VMWare-RT-2010-001, VMWare, 2010. 18, 33

[131] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *Conference on File and Storage Technologies (FAST)*, 2002. 80, 97

[132] Database sharding whitepaper. http://www.dbshards.com/articles/database-sharding-whitepapers/. 99

[133] Database sharding at Netlog with MySQL and PHP. http://nl.netlog.com/go/developer/blog/blogid=3071854. 24, 99

[134] Muhammad Bilal Sheikh, Umar Farooq Minhas, Omar Zia Khan, Ashraf Aboulnaga, Pascal Poupart, and David J. Taylor. A bayesian approach to online performance modeling for database appliances using gaussian models. In *International Conference on Autonomic Computing (ICAC)*, 2011. 58

[135] Ahmed A. Soror, Umar Farooq Minhas, Ashraf Aboulnaga, Kenneth Salem, Peter Kokosielis, and Sunil Kamath. Automatic virtual machine configuration for database workloads. In *International Conference on Management of Data (SIGMOD)*, 2008. 11

[136] Ahmed A. Soror, Umar Farooq Minhas, Ashraf Aboulnaga, Kenneth Salem, Peter Kokosielis, and Sunil Kamath. Automatic virtual machine configuration for database workloads. *Transactions on Database Systems (TODS)*, 35(1), 2010. 11, 28

[137] Gokul Soundararajan, Cristiana Amza, and Ashvin Goel. Database replication policies for dynamic content applications. In *European Conference on Computer Systems (EuroSys)*, 2006. 22, 24, 99, 101

[138] Benjamin Speitkamp and Martin Bichler. A mathematical programming approach for server consolidation problems in virtualized data centers. *IEEE Transactions on Services Computing (TSC)*, 3(4), 2010. 102

[139] Michael Stonebraker. The case for shared nothing. *IEEE Data Engineering Bulletin*, 9(1), 1986. 15, 23, 76

[140] Rob Strom and Shaula Yemini. Optimistic recovery in distributed systems. *Transactions on Computer Systems (TOCS)*, 3(3), 1985. 30

[141] Teradata Database. http://www.teradata.com/t/products-and-services/database/teradata-13/. 23, 100

[142] The R Project, 1993. http://www.r-project.org/. 59

[143] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In *Symposium on Operating Systems Principles (SOSP)*, 1997. 79, 80, 81, 83, 97

[144] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *Transactions on Database Systems (TODS)*, 4(2), 1979. 32

[145] The TPC-C Benchmark, 1992. http://www.tpc.org/tpcc/. 42, 46, 113

[146] The TPC-H Benchmark, 1999. http://www.tpc.org/tpch/. 42, 43, 89

[147] The TPC-W Benchmark, 1999. http://www.tpc.org/tpcw/. 42, 50

[148] Patrick Valduriez. Parallel database systems: Open problems and new issues. *Distributed and Parallel Databases (DAPD)*, 1(2), 1993. 24, 99, 101

[149] VMware Infrastucture. http://www.vmware.com/products/vi/. 16, 19

[150] VMware. http://www.vmware.com/. 9

[151] VMware VMotion. http://www.vmware.com/files/pdf/VMware-VMotion-DS-EN.pdf. 16

[152] VoltDB. http://voltdb.com/. 5, 100, 102

[153] Matthias Wiesmann, André Schiper, Fernando Pedone, Bettina Kemme, and Gustavo Alonso. Database replication techniques: A three parameter classification. In *Symposium on Reliable Distributed Systems (SRDS)*, 2000. 21, 22, 99, 101

[154] Microsoft Windows Azure. http://www.windowsazure.com/. 8

[155] Joel Wolf. The placement optimization program: A practical solution to the disk file assignment problem. In *International Conference on Measurement and Modeling of Computer Systems*, 1989. 102

[156] Xen Blktap2 Driver. http://wiki.xensource.com/xenwiki/blktap2. 41

[157] Min Xu, Rastislav Bodik, and Mark D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. *Computing Architecture News*, 31(2), 2003. 32

[158] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy M. Lohman, Adam J. Storm, Christian Garcia-Arellano, and Scott Fadden. DB2 Design Advisor: Integrated automatic physical database design. In *International Conference on Very Large Data Bases (VLDB)*, 2004. 24

# Appendix A

# Detailed Results Showing the Effectiveness of the Elastic Scale-out Optimizer

In this appendix, we present detailed results for the experiments that have been presented in Section 5.5.3 with a goal to provide further insights into how our optimizers achieve their benefits. We present these detailed results for the case where the overall load is distributed according to a uniform (UNIF) and normal (TNORM) distribution, and for each of the following cases: (1) scaled out system (SCO), (2) offline optimizer (OFFLINE), (3) online optimizer (ONLINE), and (4) greedy heuristic (GRD). For each case, we present results that show how each approach affects load balancing by showing per host CPU utilization. We also show how the the actual throughput and the number of nodes used vary over time.

Among all the approaches, SCO achieves the best load balance and stable throughput, but this is because there is no elasticity, i.e., it uses the maximum resources (4 nodes) throughout the experiment. However, these results show that in each case, both the OFFLINE and ONLINE optimizers are able to (1) balance load among all nodes, (2) effectively handle the offered load, and (3) minimize the number of nodes used, when compared with SCO and GRD.

Note that the fluctuating patterns in the (instantaneous) CPU utilization and throughput graphs for OFFLINE, ONLINE, and GRD are because of scale-out and scale-in operations. We note that SCO, OFFLINE, ONLINE, and GRD, in that order, provide the most to least stability, both for CPU utilization and overall throughput. There is no data movement in case of SCO, and as shown in Section 5.5.3, both OFFLINE and ONLINE move significantly less data as compared to GRD and are therefore able to achieve better stability, and consequently better performance.

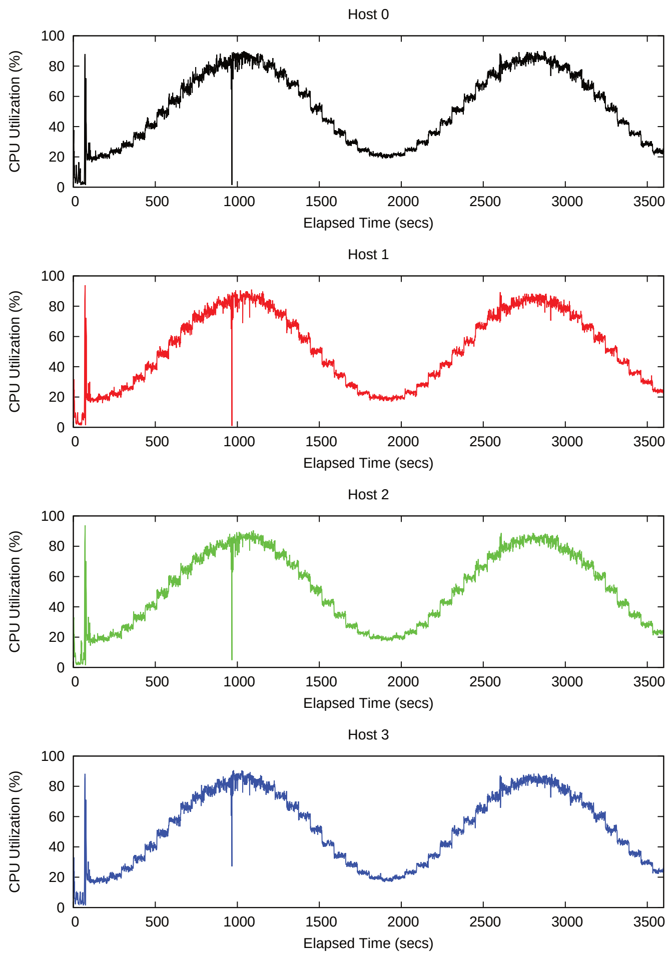


Figure A.1: Per Host CPU Utilization (SCO, UNIF)

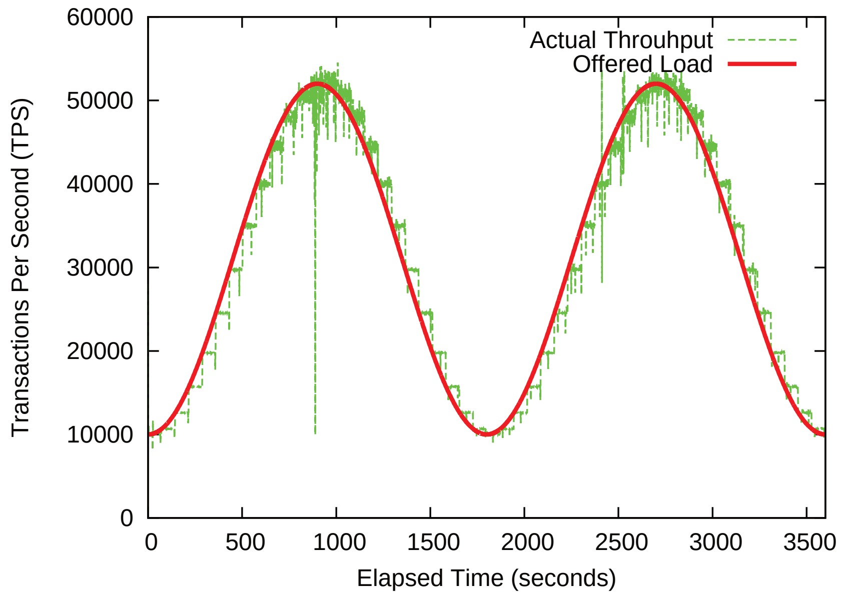


Figure A.2: Offered Load vs. Actual Throughput (SCO, UNIF)

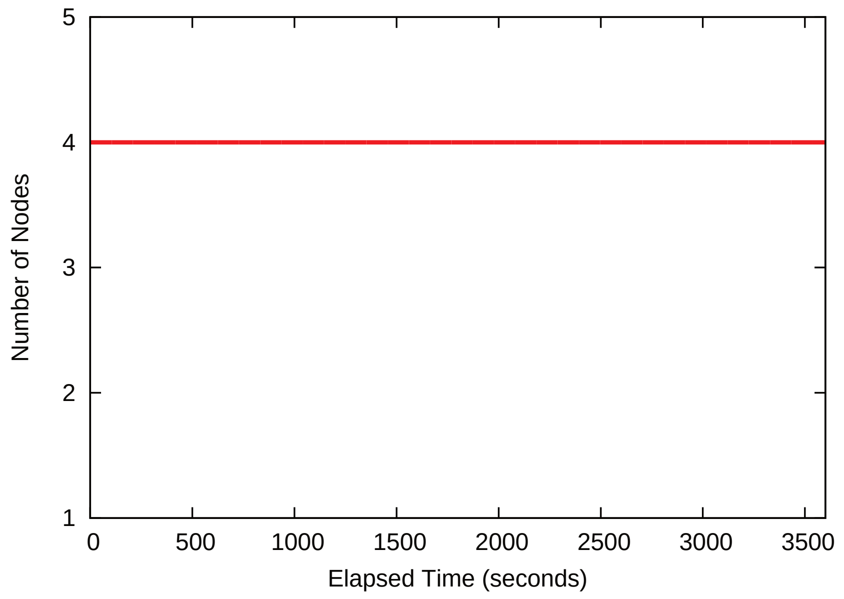


Figure A.3: Number of Nodes Used (SCO, UNIF)

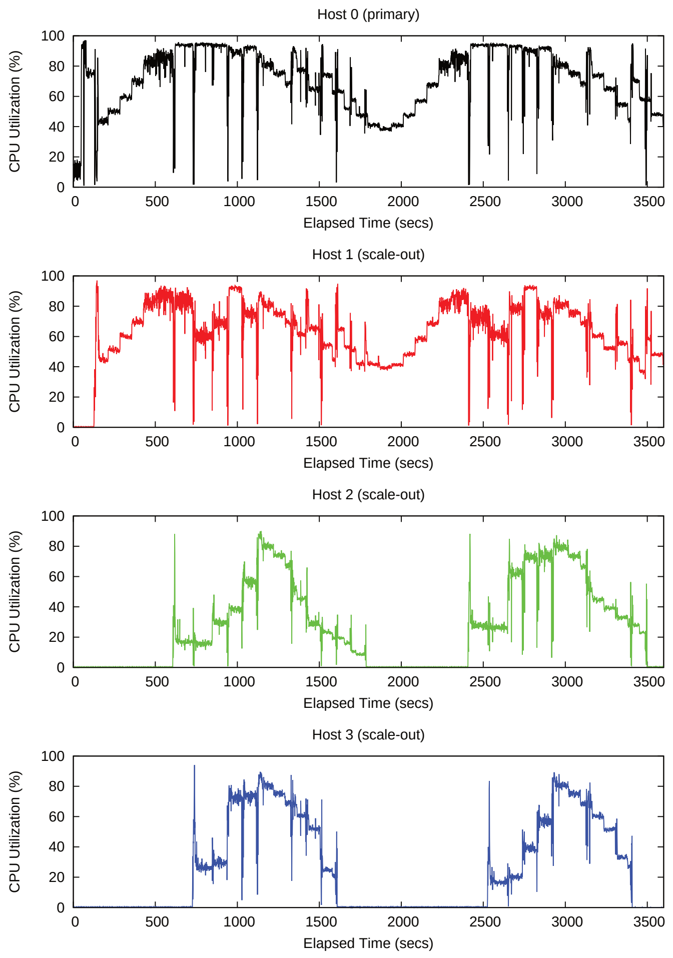


Figure A.4: Per Host CPU Utilization (OFFLINE, UNIF)

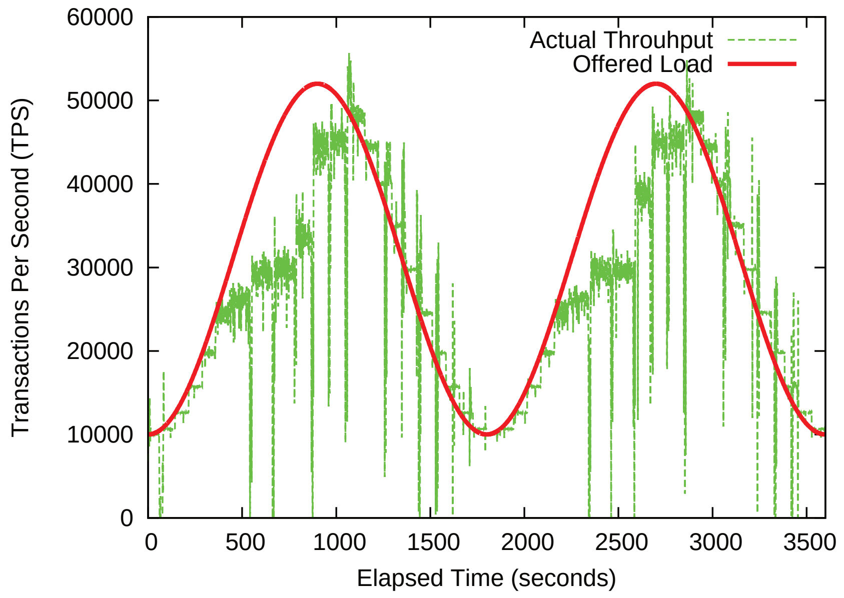


Figure A.5: Offered Load vs. Actual Throughput (OFFLINE, UNIF)

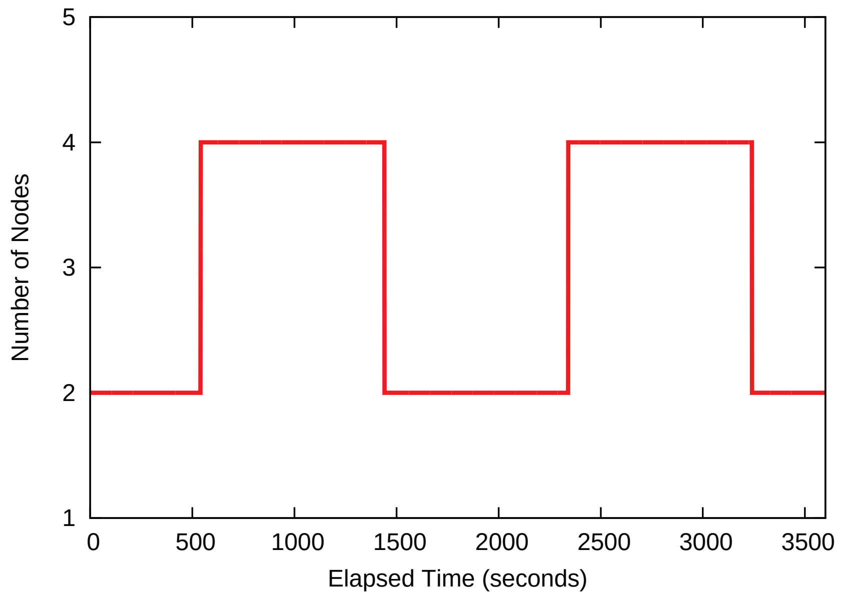


Figure A.6: Number of Nodes Used (OFFLINE, UNIF)

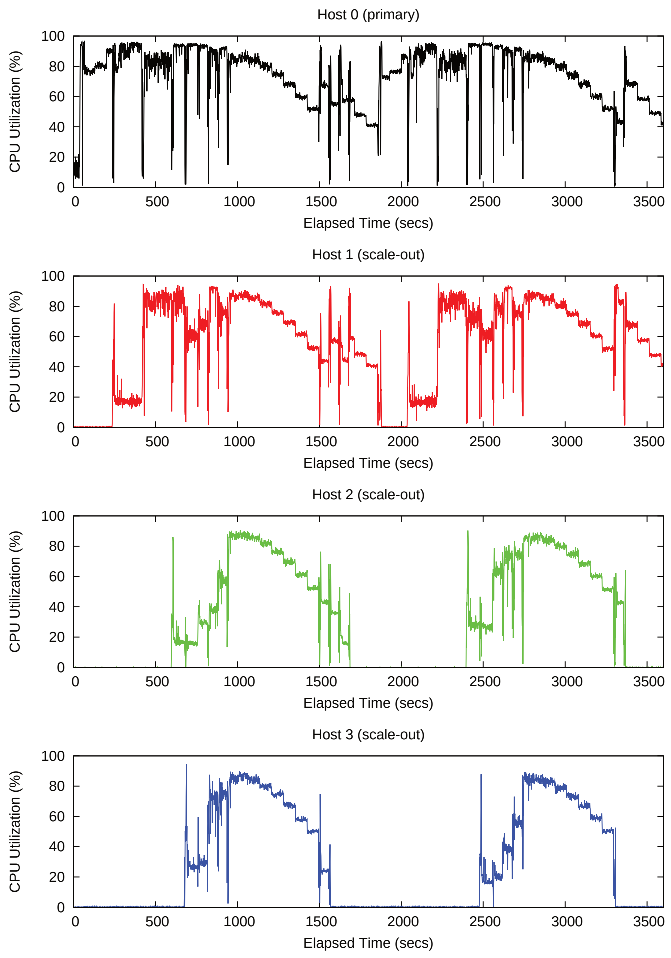


Figure A.7: Per Host CPU Utilization (ONLINE, UNIF)

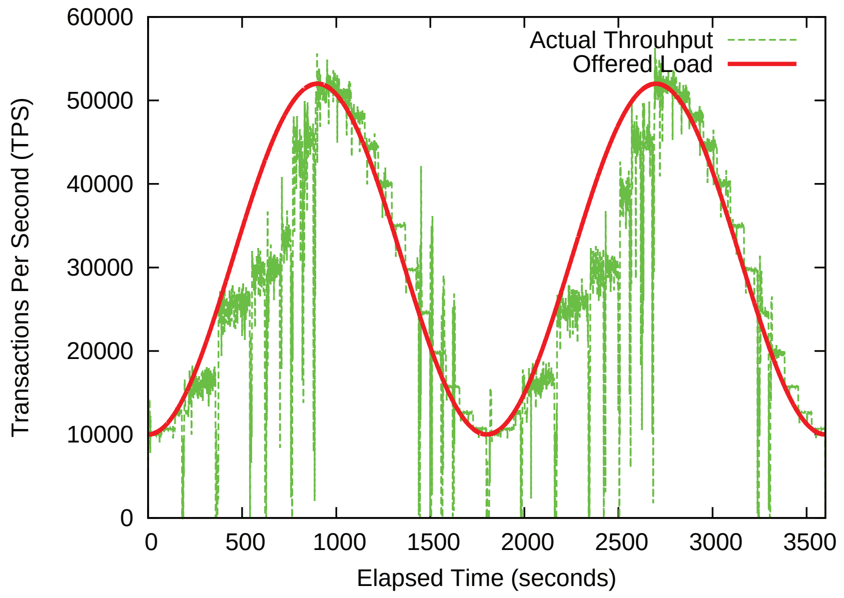


Figure A.8: Offered Load vs. Actual Throughput (ONLINE, UNIF)

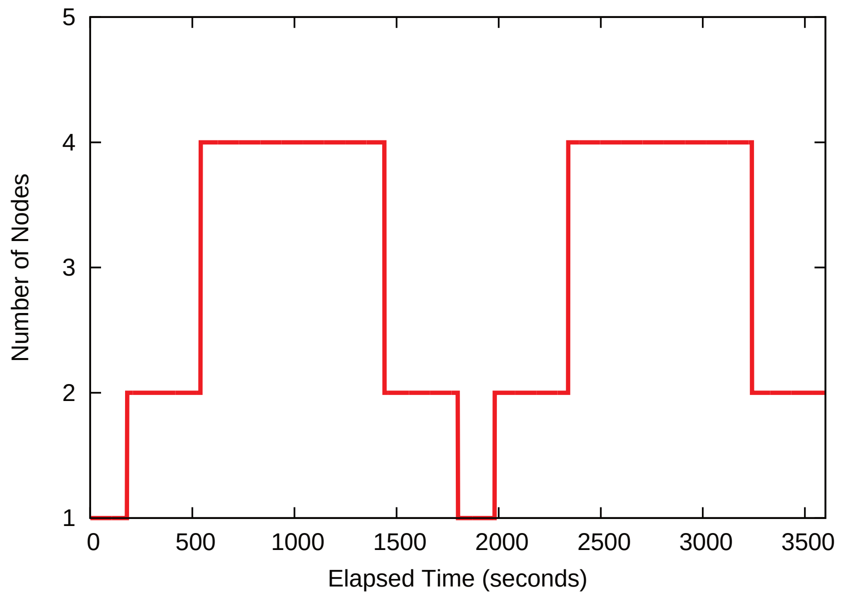


Figure A.9: Number of Nodes Used (ONLINE, UNIF)

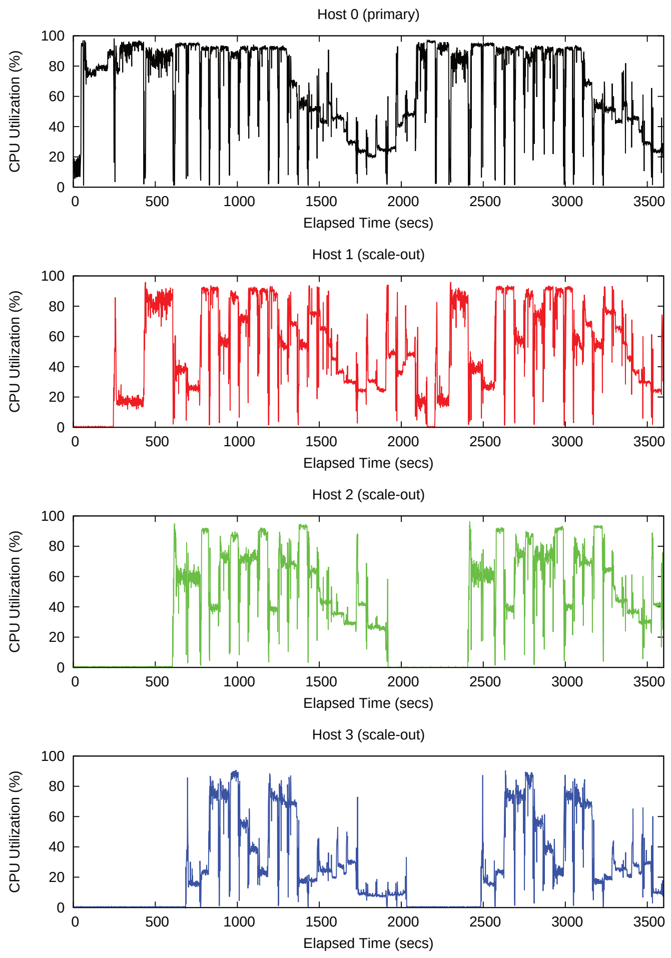


Figure A.10: Per Host CPU Utilization (GRD, UNIF)

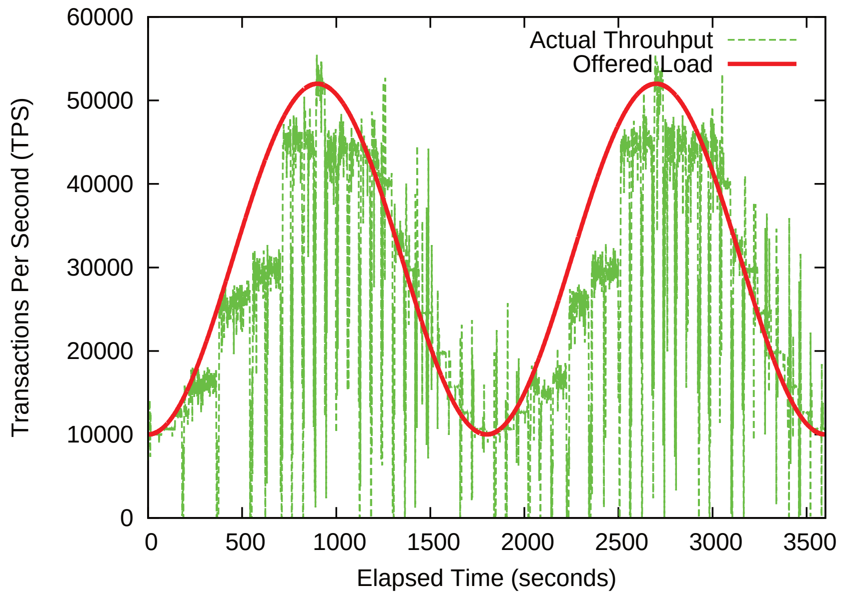


Figure A.11: Offered Load vs. Actual Throughput (GRD, UNIF)

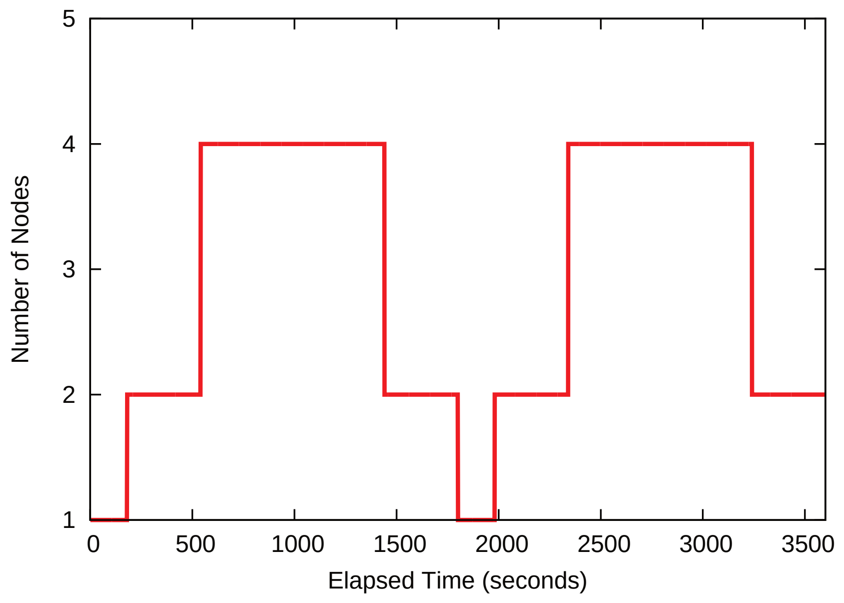


Figure A.12: Number of Nodes Used (GRD, UNIF)

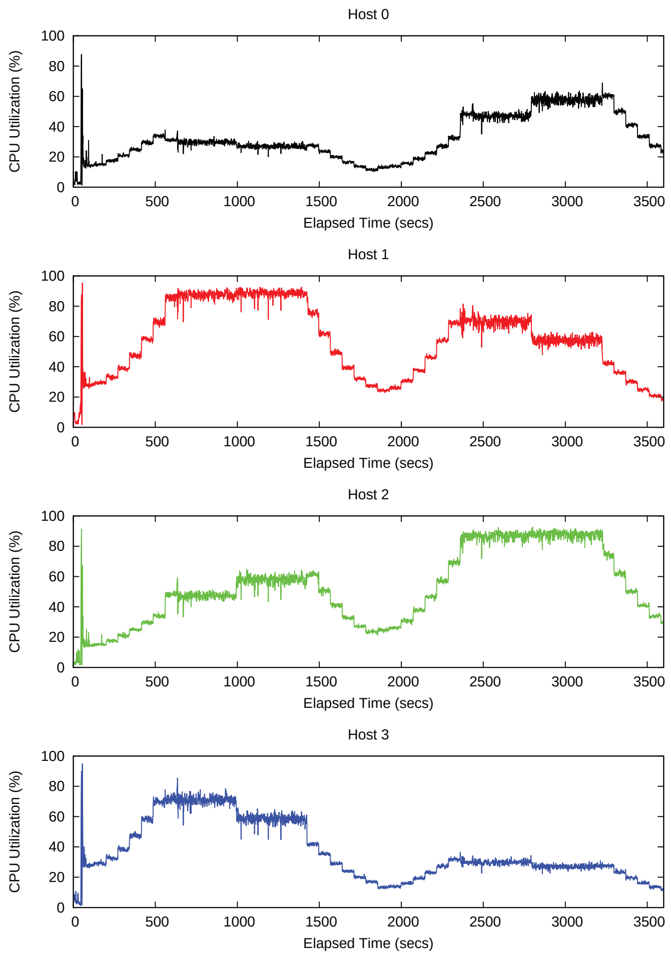


Figure A.13: Per Host CPU Utilization (SCO, TNORM)

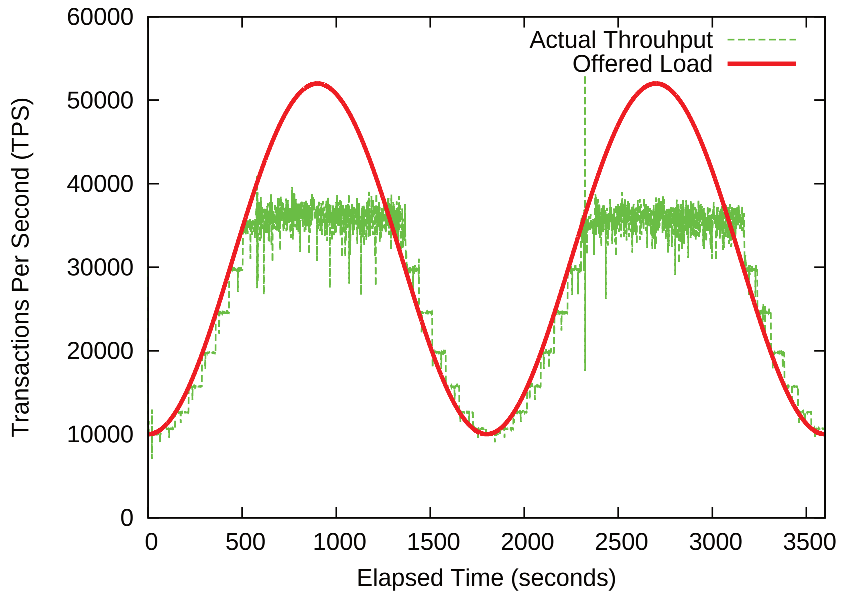


Figure A.14: Offered Load vs. Actual Throughput (SCO, TNORM)

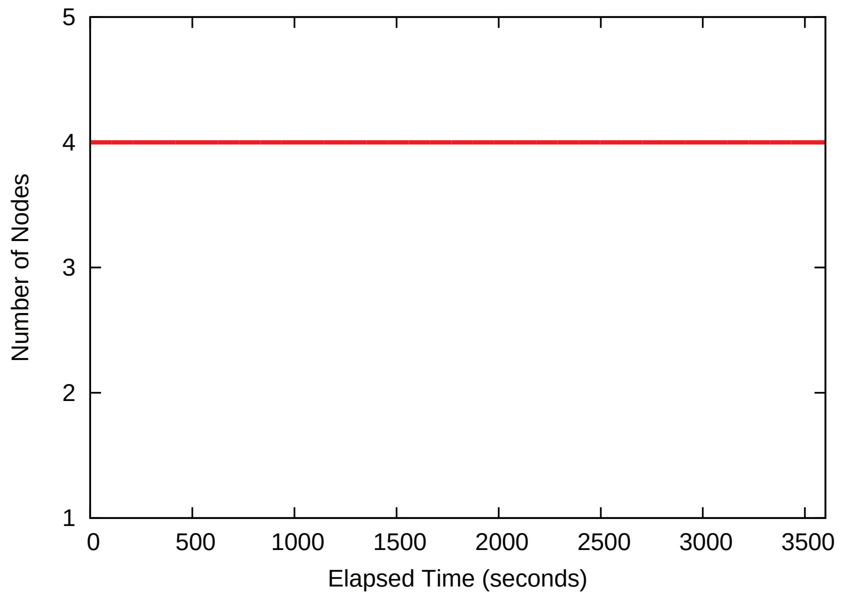


Figure A.15: Number of Nodes Used (SCO, TNORM)

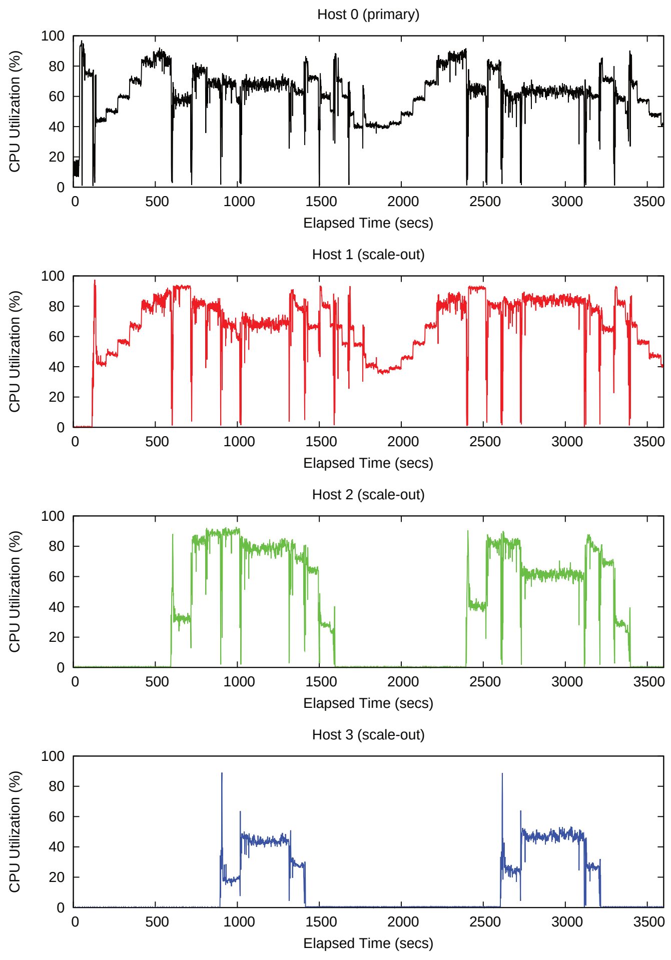


Figure A.16: Per Host CPU Utilization (OFFLINE, TNORM)

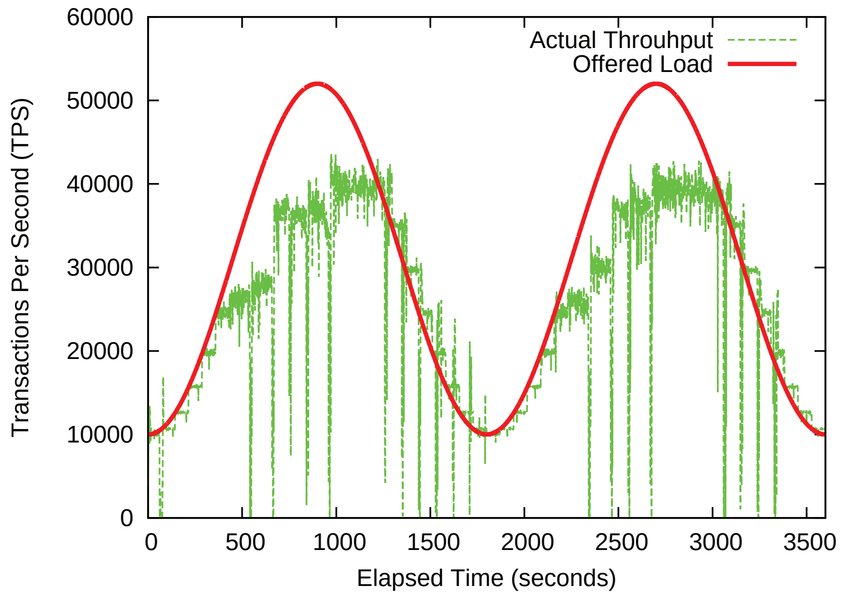


Figure A.17: Offered Load vs. Actual Throughput (OFFLINE, TNORM)

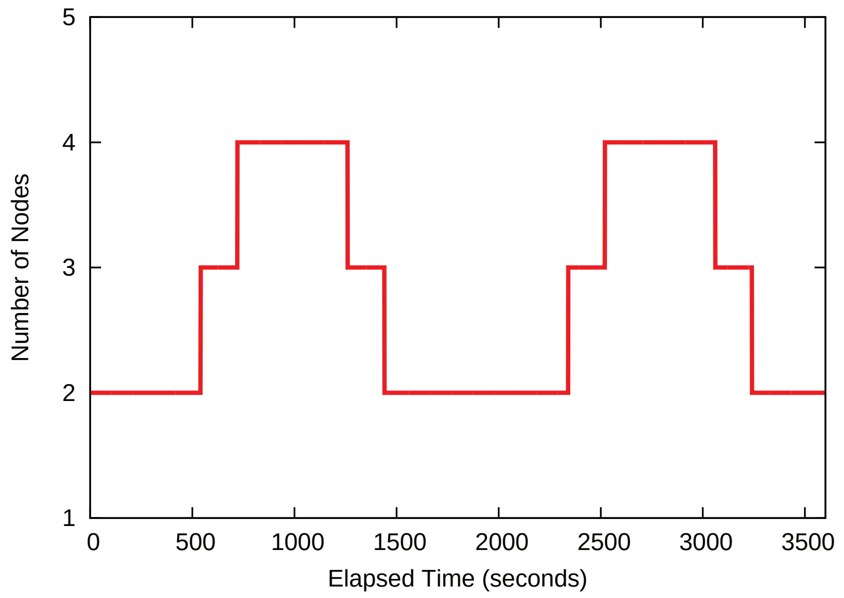


Figure A.18: Number of Nodes Used (OFFLINE, TNORM)

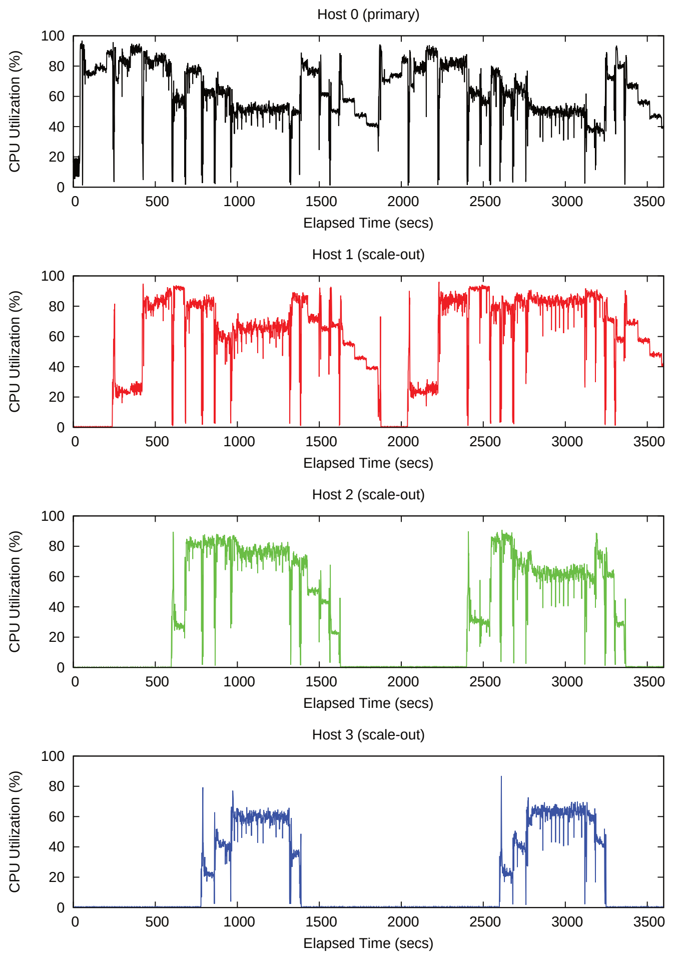


Figure A.19: Per Host CPU Utilization (ONLINE, TNORM)

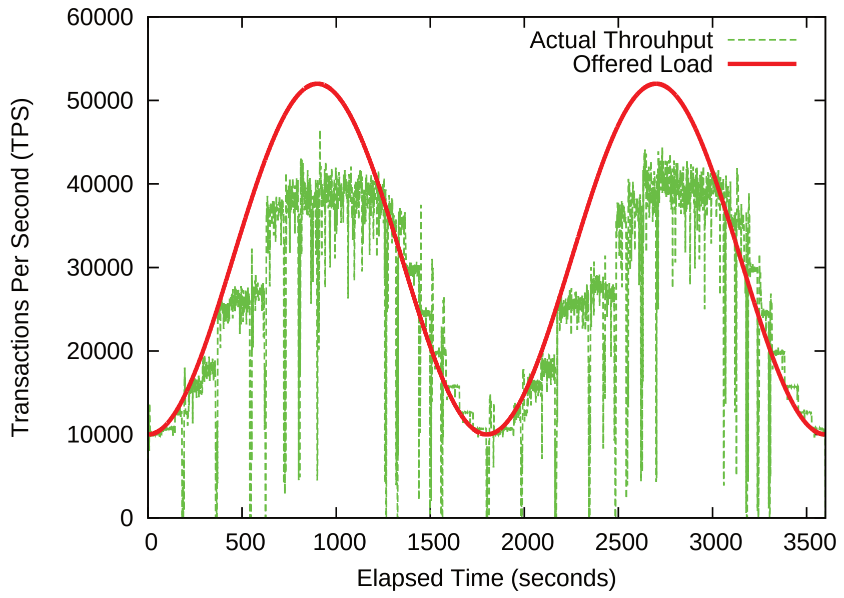


Figure A.20: Offered Load vs. Actual Throughput (ONLINE, TNORM)

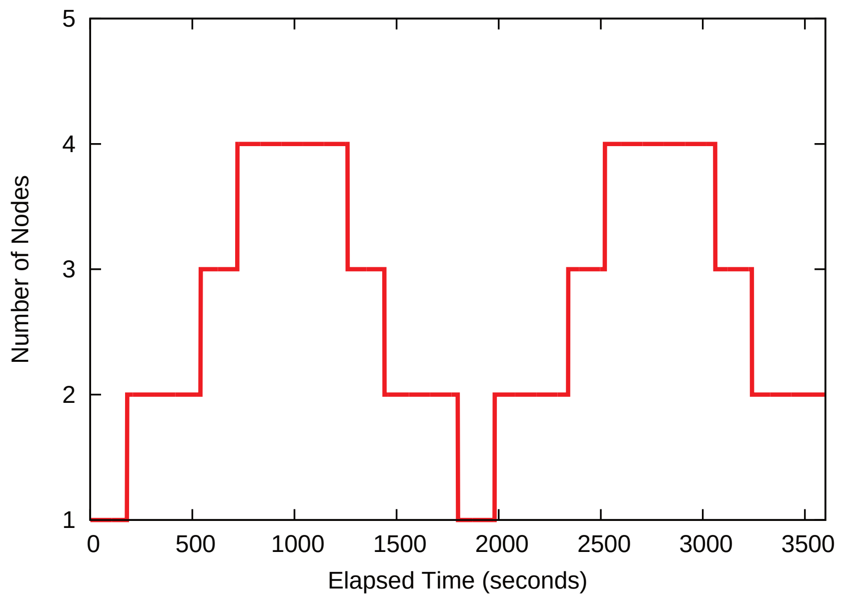


Figure A.21: Number of Nodes Used (ONLINE, TNORM)

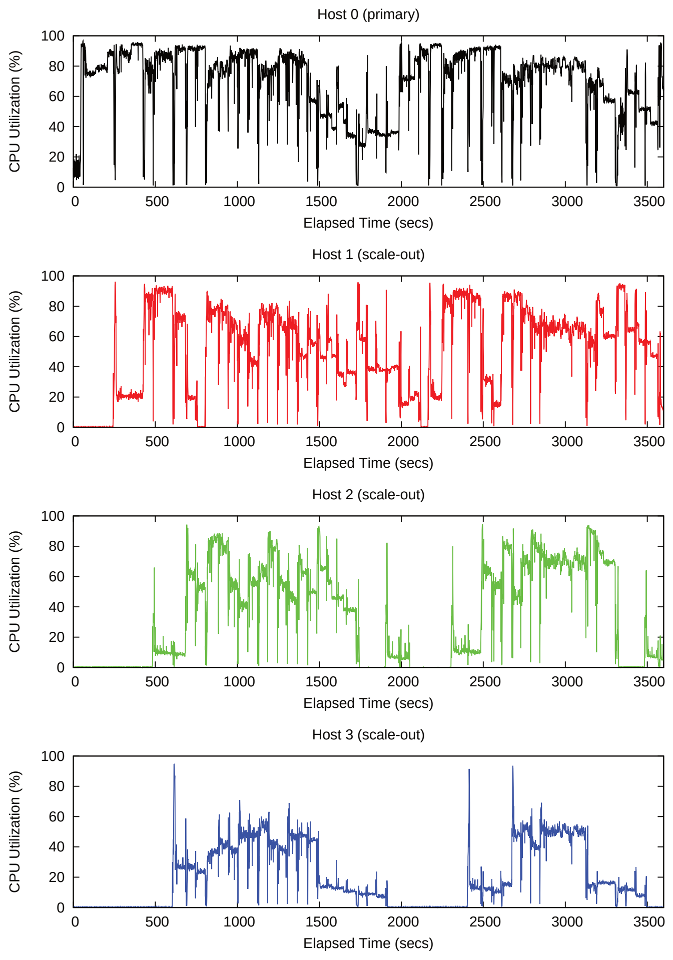


Figure A.22: Per Host CPU Utilization (GRD, TNORM)

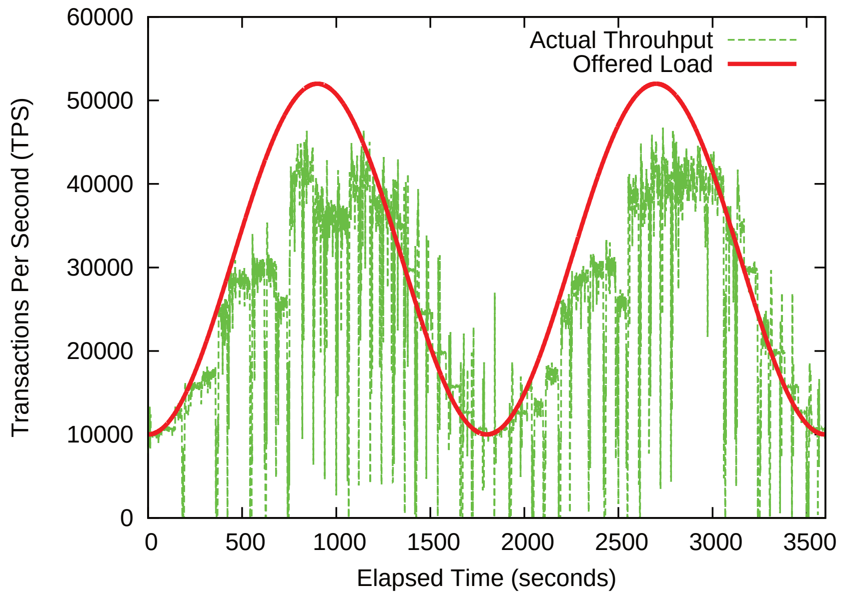


Figure A.23: Offered Load vs. Actual Throughput (GRD, TNORM)

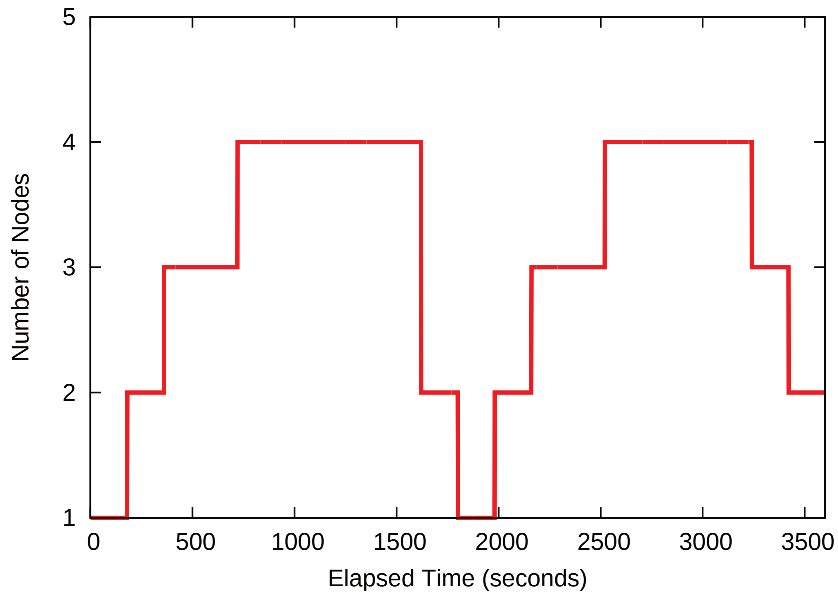


Figure A.24: Number of Nodes Used (GRD, TNORM)