Elasca: Workload-Aware Elastic Scalability for Partition Based Database Systems

by

Taha Rafiq

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2013

Abstract

Providing the ability to increase or decrease allocated resources on demand as the transactional load varies is essential for database management systems (DBMS) deployed on today's computing platforms, such as the cloud. The need to maintain consistency of the database, at very large scales, while providing high performance and reliability makes elasticity particularly challenging. In this thesis, we exploit data partitioning as a way to provide elastic DBMS scalability. We assert that the flexibility provided by a partitioned, shared-nothing parallel DBMS can be used to implement elasticity. Our idea is to start with a small number of servers that manage all the partitions, and to elastically scale out by dynamically adding new servers and redistributing database partitions among these servers as the load varies. Implementing this approach requires (a) efficient mechanisms for addition/removal of servers and migration of partitions, and (b) policies to efficiently determine the optimal placement of partitions on the given servers as well as plans for partition migration.

This thesis presents Elasca, a system that implements both these features in an existing shared-nothing DBMS (namely VoltDB) to provide automatic elastic scalability. Elasca consists of a mechanism for enabling elastic scalability, and a workload-aware optimizer for determining optimal partition placement and migration plans. Our optimizer minimizes computing resources required and balances load effectively without compromising system performance, even in the presence of variations in intensity and skew of the load. The results of our experiments show that Elasca is able to achieve performance close to a fully provisioned system while saving 35% resources on average. Furthermore, Elasca's workload-aware optimizer performs up to 79% less data movement than a greedy approach to resource minimization, and also balance load much more effectively.

Acknowledgements

I would like to begin by thanking Allah for providing me with every opportunity to learn, progress and succeed. I cannot begin to express my gratitude for everything I have been gifted in this life.

On the worldly side, there are many people who have been instrumental to my success. First of all, I would like to extend a very special thanks to my wife, Filza Mazahir, who has always been a pillar of strength for me. I could not have completed this work without her continuous support, encouragement and understanding. I would also like to express my undying gratefulness to my parents and siblings, who have always supported and guided me in more ways than I can imagine.

I would like to thank Professor Ashraf Aboulnaga, my supervisor at the University of Waterloo, for providing me with the opportunity to work with him and pursue my research interests. I could not have asked for a better supervisor to help me learn the process of research and develop my knowledge in the field of databases. I would also like to thank Umar Farooq Minhas for providing valuable feedback and insights throughout the course of this work, and for his initial work on this project. I would also like to thank Jonathan Ng, Rui Liu and Sean Robertson whose work on this project preceded me, but was very helpful to me nevertheless. Finally, I would like to thank Professor Kenneth Salem and Professor Tamer Özsu for reading this thesis and providing their feedback.

Dedication

Dedicated to my loving wife, Filza, and to *Ammi* and *Abbu*, my dear parents.

Table of Contents

List of Tables **ix**

List of Figures **x**

1 Introduction **1**

2 Related Work **4**

3 Elastic Scale-Out in VoltDB **7**

3.1 Overview of VoltDB 7

3.2 Elastic Scale-Out Mechanism 8

3.2.1 Adding/Removing Nodes 10

3.2.2 Partition Migration 10

3.2.3 Overcommitting Cores 10

4 Partition Placement Problem **12**

4.1 Problem Overview 12

4.2 Optimization Objectives 12

4.3 Transaction Model 13

4.4 Solution Overview 14

5 Workload-Aware Optimizer **17**
5.1 Notation 19
5.2 Objective Function 21
5.3 Constraints 22
5.4 Conversion to a Linear Program 24
5.5 Staggering Scale-In 24

6 Experimental Evaluation **26**
6.1 Experimental Setup 26
6.1.1 System Configuration 26
6.1.2 Different Optimizers Compared in the Experiments 27
6.1.3 Benchmarks Used 28
6.1.4 Dynamically Changing Workloads 29
6.2 Optimizer Effectiveness 32
6.2.1 Minimizing Data Movement 32
6.2.2 Saving Computing Resources 34
6.2.3 Load Balance Across Nodes 35
6.3 Overall Throughput 37
6.4 Scalability of the Optimizer 41

7 Supporting Multi-Partition Transactions **45**
7.1 Modifications to the Transaction Model 45
7.2 CPU Load Model Modifications 46
7.3 Effect of Multi-Partition Transactions on Database Throughput 47
7.3.1 Maximum MPT Throughput 48
7.3.2 Probability of Multi-Partition Transactions 48
7.4 Performance of the Optimizer with MPTs 49
7.4.1 Computing Resources Saved 49
7.4.2 Data Movement 50

8 Conclusion and Future Work **52**

8.1 Conclusion 52

8.2 Suggestions for Future Work 53

8.2.1 Changes in VoltDB 53

8.2.2 Multi-Partition Transaction Support 54

8.2.3 Automated Collection of Satistics and Parameter Tuning 54

8.2.4 Examining the Transaction Mix 54

8.2.5 Workload Prediction 55

References **56**

APPENDICES **61**

A Offline Optimizer **62**

A.1 Notation 62

A.2 Objective Function 62

A.3 Constraints 63

A.4 Solution 63

List of Tables

5.1 Notation used in this thesis. 18

6.1 Optimizer parameters. 28

6.2 Benchmark-specific parameters. 30

6.3 Statistical distributions used for load skew. 32

List of Figures

3.1 VoltDB Architecture 9

4.1 A sample transaction in our database system. 14

4.2 Overview of Elasca's system architecture. 15

5.1 Maximum number of fully loaded partitions per host possible. 20

5.2 The effect of ϵ on data movement and load balance. 23

6.1 Maximum throughput capacity with and without load skew. 31

6.2 Amount of data moved by the optimizers for various skewed load distributions. 33

6.3 Data movement performed by optimizers with increasing database size. . . 34

6.4 Data movement performed by optimizers for a fluctuating workload. 35

6.5 Average percentage of computing resources saved. 36

6.6 Average variance in CPU utilization of each node. 36

6.7 Overall throughput for the TPC-C, TATP and YCSB benchmarks. 38

6.8 Throughput over time of the optimizers for the TPC-C benchmark (NORM load skew). 39

6.9 Measured and predicted CPU utilization of the hosts over time for ELASCA (NORM load skew). 40

6.10 Average throughput for the TPC-C benchmark, for a fluctuating workload. 41

6.11 Increase in the amount of computing resources used by ELASCA when compared to a fractional idealized solution. 42

6.12 Time taken by ELASCA to find solution with increasing problem size. . . . 43
6.13 Quality of the solution returned by OFFLINE with increasing problem size. 44

7.1 Effect of η on maximum throughput possible. 48
7.2 Effect of p_{mpt} on maximum throughput possible. 49
7.3 Computing resources saved by ELASCA and ELASCA-MPT. 50
7.4 Data moved by ELASCA and ELASCA-MPT. 51

Chapter 1

Introduction

Today's computing platforms, such as computing clouds and private clusters favor systems that can effectively use resources in an on demand fashion, for example, by using more physical servers when the load increases and releasing them when the load decreases [6]. Such *elastically* scalable systems fit well with the pay-as-you-go cost model of computing clouds such as Amazon's EC2 [1]. Applications can access a large number of physical servers and can *scale out* and *scale in* depending on the needs of their (time varying) workload. However, not all applications are designed to achieve elastic scalability. Cloud database management systems (DBMS) are one particular example where attaining elasticity conflicts with the requirement to maintain consistency of the database. Elasticity becomes even more challenging in the presence of other requirements such as providing fault tolerance at very large scales. While there have been some efforts to build fault tolerant cloud database systems [32], providing elasticity is still largely open to research. Database systems are at the core of many data intensive applications deployed in the cloud, which are directly affected by the scalability of the DBMS. Database systems have to be (re)designed to provide elastic scalability to make them better suited for today's computing needs.

Traditional scalable database systems enable multiple *nodes* (i.e., virtual or physical servers) to manage a database, but the set of nodes is static. These systems distribute the database among the nodes by relying on *replication* [2, 3, 26, 47] or *partitioning* [13, 29]. 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 seems to be gaining popularity as a way to scale database systems (e.g., [38, 39]), and in this thesis we argue that partitioned, shared-nothing databases 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 [1], 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 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 [40, 45]. Thus, this approach requires an efficient *mechanism* for partition redistribution.

However, having this efficient mechanism isn't enough to enable a database management system to scale elastically in response to a time varying workload. An elastically scalable database system needs to be able to decide when to scale-out/in or shuffle partitions among existing nodes. 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 turn leads to either lost customers or a waste of costly computing resources [6]. Also, once the DBMS decides to migrate partitions, it also needs to decide which partitions to move to which nodes during a scaling operation, which is highly dependent on the workload. Therefore, in addition to an efficient mechanism, the DBMS needs *policies* to effectively govern the placement and migration of database partitions [31].

In this thesis, we present Elasca (short for Elastically Scalable), a system that extends an existing parallel shared-nothing partition based database system (namely VoltDB [46]) with (a) an efficient partition redistribution mechanism and (b) a workload-aware optimizer to determine partition placement and migration plans. We briefly describe the changes that we made to VoltDB to enable us to dynamically add new nodes to the cluster and redistribute partitions. However, the main focus of this thesis is Elasca's optimizer for solving the problem outlined in the previous paragraph, which we call the *partition placement problem*. We show that depending on the workload, partition placement and migration policies can have a significant impact on the cost and performance of an elastically scalable database system. We present a *workload-aware* partition placement strategy that relies on mathematical optimization techniques to find an optimal partition placement based on workload characteristics. More specifically, we present a problem formulation for the partition placement problem as a mixed-integer linear program (MILP) in an *online* setting, where we do not know the future load. We use a general purpose solver (IBM ILOG

CPLEX [15]) to find a solution to the partition placement problem. We proceed to present a detailed evaluation of Elasca through simulations and actual experiments, and show that our optimizer outperforms a simple greedy optimizer, and matches the performance of an offline optimizer (i.e., an optimizer that has knowledge of the future workload). We further show that our online optimizer provides better scalability than the offline optimizer.

The main contributions of this work are:

- **Workload-Aware Optimizer:** An optimizer for solving the partition placement problem in an online setting. We provide a mathematical formulation of the partition placement problem that can be efficiently solved by using a general purpose solver. Our formulation also takes into account various characteristics of the workload for finding an optimal assignment of partitions to nodes. We further present a heuristic that enhances the performance of our optimizer in certain cases.

- **Experimental Evaluation:** Detailed experiments that show the effectiveness of our optimizer in finding partiton placements that balance load among all nodes even in the presence of skew, while at the same time minimizing computing resources needed (in terms of physical servers as well as network and data migration costs).

The rest of this thesis is organized as follows. In Chapter 2, we present related work in the areas of scalable database systems and partition placement. In Chapter 3, we present an overview of VoltDB and the elastic scale-out mechanism. Chapter 4 then focuses on the specific problem of partition placement and migration in an elastically scalable system. Elasca's workload-aware partition placement optimizer is presented in Chapter 5. We present experimental results that show the effectiveness of our optimizer for a variety of workloads in Chapter 6, as well as experiments that show the advantage of our partition placement strategy over other approaches. Next we discuss how multi-partition transaction support can be added to our optimizer in Chapter 7. Finally, we present some suggestions for future work in Chapter 8.

Chapter 2

Related Work

Most of the existing techniques for scaling database systems either rely on data replication [2, 3, 8, 11, 22, 26, 33, 47] or data partitioning [9, 14, 20, 29, 35]. 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 on-going transaction processing [40, 45]. Furthermore, none of the existing techniques explicitly deal with elastic scale-out.

Similar to our work, the Relational Cloud project at MIT [16] 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 [18] 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. In other research [17] related to the same project, the authors present techniques for defining optimal database partitioning and/or replication in a workload-aware fashion that improves database scalability. Pavlo et al. [34] present another technique for partitioning the database in a skew-aware manner. 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 [19] 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 (i.e., node), requiring a more elaborate mechanism for data migration (Chapter 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 thesis is applicable to both of these systems.

There is an increasing body of work in the area of elastically scalable data stores, in particular, for cloud computing environments. Key-value stores [10, 23, 28] 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* against a given *key* from the store. SQL is not supported, hence the name “NoSQL”. 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 (VoltDB) that provides full SQL functionality.

Recently, there has also been interest in the problem of automatic resource allocation in the context of cloud storage systems. In [44] the authors present a control framework called *SCADS* for scaling a distributed storage system in response to workload changes. For their controller, they use an intelligent greedy heuristic that moves data from the overloaded servers and coalesces underloaded servers. Although our work bears close similarity to the problem discussed in [44], there are several key differences: we choose to find an exact solution to the partition placement problem versus using heuristics, and we leverage several key properties of shared-nothing partition based database systems in our solution (e.g., the transaction model).

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 [5, 9, 13, 24, 29, 37] and file servers [21, 48]. 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 to maximize load balance [18].

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, which has not been addressed before.

The general partition placement problem has been shown to be NP-complete [37]. 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 techniques used for solving that problem, such as genetic algorithms, can be used for finding a solution [27]. In this work, we use mathematical programming to solve this problem. Similar to our work, in [7, 41] 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 cost of operating a data center. An integer programming approach to assign servers to applications has been presented in [36], 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. Rather we use a general purpose solver to find a solution.

Chapter 3

Elastic Scale-Out in VoltDB

3.1 Overview of VoltDB

VoltDB [46] is a in-memory database system derived from H-Store [25]. 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. 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:

- All data is stored in main memory, which avoids slow disk operations.
- All operations (transactions) that need to be performed on the database are predefined in a set of stored procedures that execute on the server, i.e., ad hoc transactions are not allowed.
- Transactions are executed on each database partition serially, with no concurrent transactions within a partition, thereby eliminating the need for concurrency control.
- 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. There are two types of threads in VoltDB. Firstly, there are multiple *execution site* threads per host, which are responsible for executing transactions against their respective partitions. Note that there is a one-to-one mapping between execution sites and database partitions. Secondly, there is a host-level *initiator* thread, which is responsible for coordinating transaction execution, and returning the results to the clients. Therefore the work performed at each host can be divided into two parts; the serially executed transaction initiation/coordination part, and the actual execution of transactions which is done in parallel.

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 *execution site* managing a partition will always be executing on its own CPU core, and there will still be some cores left for transaction initiation and coordination. Thus, threads never contend with other threads for cores, leading to maximum performance. 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. Figure 3.1 shows the architecture of VoltDB.

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

1. **Single-Partition Transactions:** These transactions involve only a single database partition and are therefore very fast.

2. **Multi-Partition Transactions:** These transaction 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 often the case for the kinds of transactional workloads that VoltDB targets, and is also our focus in this thesis. We discuss possible ways to add multi-partition transaction support to Elasca in the Chapter 7.

3.2 Elastic Scale-Out Mechanism

We briefly discuss Elasca's elastic scale-out mechanism in this section. However, the scale-out mechanism is not the focus of this thesis, and hence we do not go into many of the

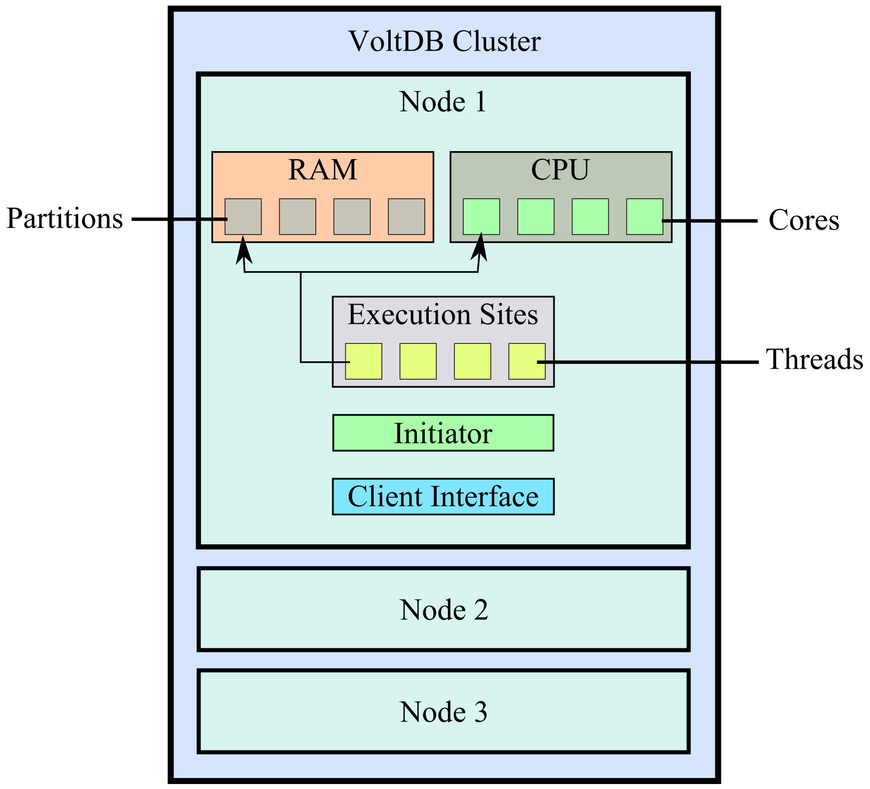

Figure 3.1: VoltDB Architecture

implementation details. More details can be found in [31].

In the case of a partition based database system, elastic scale-out primarily consists of two tasks:

1. **Adding/Removing Nodes:** Adding nodes to the existing cluster, or removing nodes from the cluster. This is required in the case of a scale out or scale in operation.
2. **Partition Migration:** Migrating partitions between nodes. This is required during scale out, scale in, and partition shuffle operations.

3.2.1 Adding/Removing Nodes

In order to implement scale out, a new type of node, a *scale-out node* was added to the VoltDB implementation. A scale-out node is a node that can potentially be added to the VoltDB cluster to support scale out operations. VoltDB is aware of all the available scale-out nodes, but these nodes are idle and do not store data or serve requests until they join the cluster. VoltDB can has a built-in *failure detection* and *node rejoin* mechanism to deal with node failures. In our mechanism, a scale-out node is perceived by VoltDB as a 'failed' node, until a scale out operation is initiated. At that point, the scale-out node becomes 'active' and rejoins the cluster as if it were a failed node that came back up. Similarly, for the case of a scale in operation, a node is considered 'failed' and returned to the pool of scale-out nodes once all the partitions are migrated from it.

3.2.2 Partition Migration

In order to implement partition migration, VoltDB's *recovery* mechanism was utilized. The recovery mechanism is used by failed nodes to recover data from replicas once they become active again. The recovery mechanism copies data from the source partition to the destination partition in a transactionally consistent manner. In the case of a scale out operation, the recovery mechanism is used for all the partitions assigned to the new node. The source partition is removed once the destination partition has fully 'recovered' and becomes active.

3.2.3 Overcommitting Cores

VoltDB suggests that the number of partitions assigned to a node be less than or equal to the number of cores in the node so that each server thread can run on its own core. However, this is not always required. If the system is lightly loaded then multiple threads can be assigned to the same core without incurring any performance penalty. This observation is fundamental to the effectiveness of our optimizer. The system can aggregate a large set of partitions on one node if they are lightly loaded. As soon as the load on some partitions increases, the system can be scaled out accordingly. Similarly, as load decreases, the partitions can be aggregated together again.

Chapter 4

Partition Placement Problem

In this chapter we provide an overview of the partition placement problem and our proposed solution. We discuss the problem briefly, and then proceed to discuss the objectives that any solution must meet. Next, we present a *transaction model* for partition based database systems. This transaction model acts as the basis of our solution, as we use it to effectively estimate the load exerted by each partition. Finally, we present a high-level overview of our solution and discuss Elasca's system architecture.

4.1 Problem Overview

Given a set of database partitions, the instantaneous load on each partition, and a set of nodes (where each node has a certain CPU and memory capacity), our goal is to find an "optimal" solution that gives us the minimum number of nodes required and a mapping of partitions to these nodes to effectively handle the transactional load.

4.2 Optimization Objectives

There are a number of ways to define optimality of the solution with respect to the partition placement problem:

- **Maximizing Throughput:** An optimal solution should match the performance of a fully scaled out system, regardless of the request rate or load skew. A fully scaled out system is defined as a system in which each partition is assigned its own thread and allowed to run without any contention between threads.

- **Minimizing Resource Consumption:** An optimal solution should minimize resources required (number of nodes in our case) while at the same time maximizing system throughput.
- **Minimizing Data Movement:** An optimal solution should minimize the amount of data that needs to be moved as data movement adversely affects performance and incurs additional network utilization cost.
- **Balancing Load Effectively:** An optimal solution should prefer assignment of partitions to nodes that divide system load evenly among nodes over unbalanced assignments. This is done in order to minimize the risk of overloading any particular node, potentially affecting the throughput of the entire cluster as a result.

4.3 Transaction Model

In order to devise a solution to the partition placement problem, we must first present a *transaction model* for partition based database systems which will act as the mathematical foundation of the solution. The transaction model represents the 'work' that needs to be performed in order to complete one transaction.

In this thesis, we follow a transaction model that closely resembles that of VoltDB for single-partition transactions. We assume that all the transactions in our system will be single partition, and consist of two parts:

1. **Transaction Initiation and Coordination:** A client can send a transaction request to any host that is part of the database cluster. The request is received by a host-level *initiator*, which creates a transaction and forwards it to the respective *execution site* that is responsible for the partition specified in the transaction. As each initiator is created at the host-level, the number of initiators in the system is equal to the number of nodes.
2. **Transaction Execution:** Each partition has a one-to-one mapping with an execution site. Each execution site can receive transaction work from any of the initiators in the cluster. Once the work is completed, the execution site replies to the respective initiator which insures that all the replicas of the partition successfully completed their work. The transaction is committed at this point.

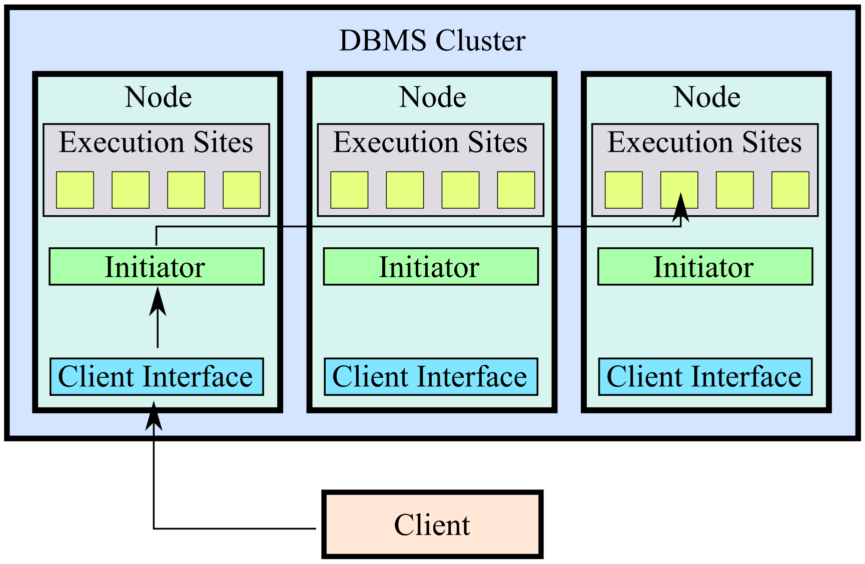

Figure 4.1: A sample transaction in our database system.

Figure 4.1 presents a sample transaction that follows the transaction model outlined above. The client sends a transaction request to the *client interface* of any node that forms a part of the database cluster. The client interface forwards the request to the initiator, which routes it to the relevant execution site. The execution site performs the requested transaction work and responds to the initiator, which then returns the result of the transaction to the client through the client interface.

4.4 Solution Overview

The partition placement problem can be formulated using mathematical programming (e.g., integer programming) and solved using mathematical optimization techniques. This is the approach we take in Elasca. Elasca's optimizer assumes that we know the instantaneous load on each partition (transactions per second or TPS) and the system state (e.g., current CPU load 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, find the optimal partition-to-node mapping for the the time interval $t+1$.

Figure 4.2 presents an overview of the system architecture of Elasca. Elasca's operation can be divided into two phases: pre-processing and execution.

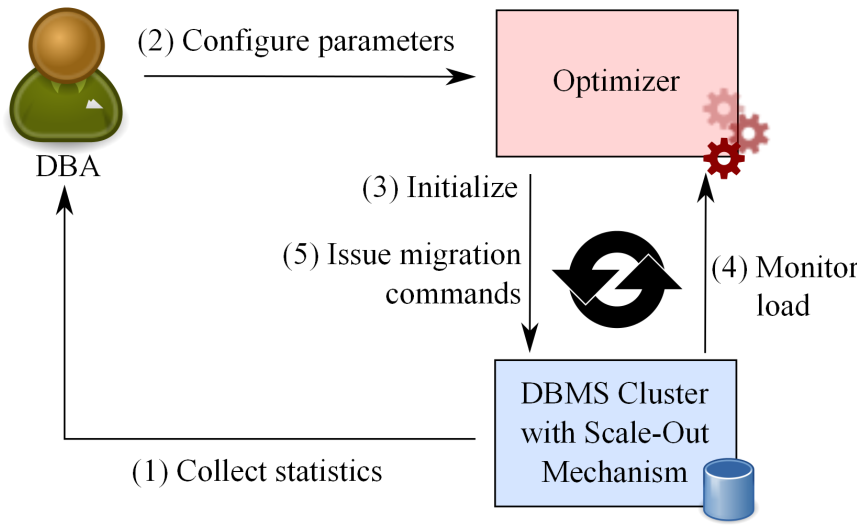

Figure 4.2: Overview of Elasca's system architecture.

1. **Pre-Processing:** Initially, the database administrator (DBA) needs to collect certain statistics about the workload (step 1 in Figure 4.2). These statistics can be collected by running the workload on a multiple node cluster having one partition on each node. A multiple node cluster provides a realistic estimate of these values as it includes the cost of network communication between the nodes, while a single node instance provides artificially boosted estimates due to lack of network communication. The following workload-specific statistics need to be collected:

    - The maximum number of transactions that can be executed on a single partition when there is no CPU contention. In VoltDB terminology, each partition's *execution site* thread runs on a separate CPU core.
    - The average CPU utilization of threads involved in host-level tasks, i.e., the *initiator* thread in case of VoltDB.

    The collected statistics are used to estimate the CPU load generated by the transactions on a particular partition during the actual deployment of the system. The number of nodes in the cluster used for collecting these statistics should be set at a relatively high value to approximately reflect the cross node communication in a scaled out system. This is done to provide a conservative estimate to the optimizer about the maximum number of transactions that a single partition can execute.

2. **Execution:** After collection of these statistics, the DBA provides these values as input to the optimizer along with the number of partitions in the database, the size of each partition, system specifications (such as the number of cores per node and amount of physical memory), and the frequency at which to recalculate the partition-to-node mapping (Step 2 in Figure 4.2).

   The optimizer uses the values provided by the DBA to generate a valid initial partition-to-node placement and initialize the cluster (Step 3). Once the cluster has been initialized, the optimizer monitors the request rate per partition (Step 4), estimates the CPU load generated by each partition and recalculates the partition-to-node mapping at the provided frequency. If partition migration is required, the optimizer issues commands to perform live migration of the respective database partitions (Step 5).

In the next chapter, we present more details about these steps when we discuss the Elasca optimizer.

Chapter 5

Workload-Aware Optimizer

This chapter presents Elasca's workload-aware partition placement optimizer. The optimizer is designed to be used in an online setting, and hence must be able to react to changes in the workload quickly and effectively, without requiring any knowledge of future changes in the workload. In contrast, one can consider an offline optimizer that has full knowledge of how the workload varies over time (i.e., full knowledge of the future) when it makes its optimization decisions. Such an optimizer would be useful if the workload has strong periodic patterns so that the workload in one period can be used as an estimate of load in a future period. Such an optimizer is also useful as a baseline for comparison since an offline solution that is based on full knowledge of the future is likely better than an online solution that is based on instantaneous information. An offline optimizer was developed as part of [30]. We compare against it in our experiments, and present more details about it in Appendix A.

Elasca's workload-aware online optimizer uses mathematical optimization techniques to find an exact solution to the partition placement problem, which is formulated as a mixed-integer linear programming (MILP) problem. Although it is possible to use heuristic techniques to obtain approximate solutions to this problem, we chose the mathematical optimization approach as it provides the best possible solution in a wide variety of scenarios (i.e., it is general purpose). The following sections present the details of our problem formulation.

Symbol	**Description**
P	Number of partitions
K	Number of replicas of each partition
t	Current time interval
N^t	Number of live nodes in the cluster at time t
H	Maximum number of nodes in the cluster
C	Number of CPU cores in each node
M	Physical memory in each node
m_i^t	Size of partition i at time t
α	Max throughput of one partition in transactions per second
β	Transaction initiation and coordination overhead of a loaded partition
r_i^t	The observed request rate for partition i at time t
l_i^t	CPU load generated by partition i and time t
γ^t	Transaction initiation and coordination overhead for a single node at time t
μ^t	Average CPU load on a single node at time t
A_{ij}^t	Assignment of partition i to host j at time t
TP	Length of time, in minutes, for which a particular workload is executed
F	Frequency, in minutes, at which the optimizer is run
T	Total number of time intervals

Table 5.1: Notation used in this thesis.

5.1 Notation

The notation used in this thesis is summarized in Table 5.1. Let P denote the total number of unique partitions in a database, each replicated K times for durability. The maximum number of nodes required to sustain P fully loaded partitions is represented by H. The number of *live* nodes at time t is represented by a variable $N^t \in \{1, \ldots, H\}$. N^t is calculated as:

$$N^t = \sum_j signum(\sum_i A^t_{ij}) \tag{5.1}$$

where A^t is a P by H matrix representing the assignment of partitions to nodes. Element A^t_{ij} is 1 if partition i is assigned to host j at time t and 0 otherwise. The role of the optimizer is to find the best A^t matrix.

We turn now to the cluster configuration. Each node has C cores, each of which can handle CPU load between 0 and 100% (denoted by the range $[0, 1]$). The amount of physical memory in each node is represented by M. The size of each partition at time t is stored in a vector m^t where m^t_i represents the size of partition i.

The maximum number of transactions that a single partition can execute per second is denoted by α, and the serial overhead (for transaction initiation and coordination) of a fully loaded partition is denoted by β (range $(0, 1]$). β represents the percentage CPU utilization of the *initiator* thread of VoltDB. These values are collected and provided by the database administrator. In order to account for the serial overhead when all the partitions are loaded to their full capacity, we calculate the maximum number of nodes required to sustain P partitions (H) as:

$$H = max\left(ceil\left(\frac{P(1+\beta)}{C}\right), ceil\left(P * \beta\right)\right) \tag{5.2}$$

The greater the serial overhead, the less partitions a node can host without the serial overhead becoming the bottleneck. The first parameter of the max function represents the number of nodes required if the serial overhead is not the bottleneck. The second parameter represents the number of nodes required if the serial overhead is the bottleneck. We take max of both these values to ensure that both cases are handled adequately. Figure 5.1 shows the effect of the value of β on P/H. If we assume that there are enough cores available, then Figure 5.1 demonstrates the maximum number of fully loaded partitions a single host can sustain without exhausting the serial bottleneck.

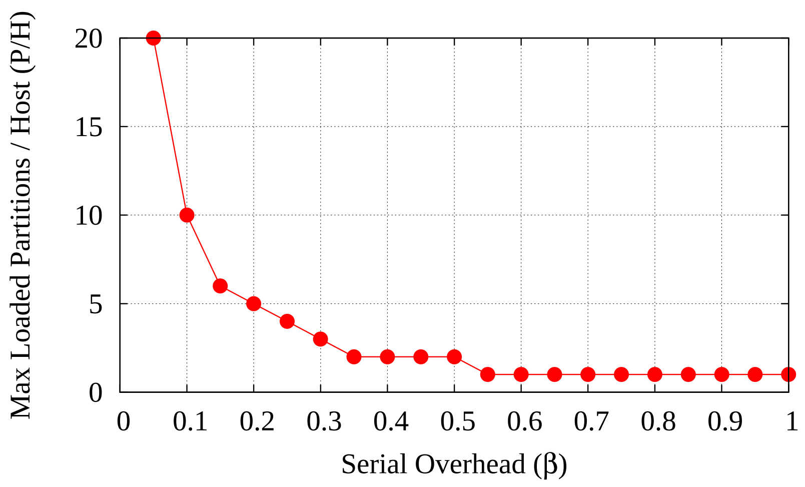

Figure 5.1: Maximum number of fully loaded partitions per host possible.

The observed request rate for each partition at time interval t is stored in a vector r^t where r_i^t represents the request rate for partition i. The estimated CPU load generated by each partition is stored in a vector l^t where l_i^t represents the load generated by partition i. The value of l_i^t is calculated as:

$$l_i^t = min(r_i^t/\alpha, 1) \tag{5.3}$$

As β represents the serial overhead of a single fully loaded partition, the projected average serial overhead on a single node for time $t+1$, denoted by γ^{t+1}, is calculated as:

$$\gamma^{t+1} = \frac{\sum_{i=1}^{P} l_i^t * \beta}{N^{t+1}} \tag{5.4}$$

As the above equation shows, we use partition load estimates from time t to calculate γ^{t+1}. Therefore we implicitly assume that the load per partition in the next time interval will be similar to the current load per partition. The projected average CPU load on each node at time $t+1$, denoted by μ^{t+1}, is calculated as:

$$\mu^{t+1} = \gamma^{t+1} + \frac{\sum_{i=1}^{P} l_i^t}{N^{t+1}} \tag{5.5}$$

The length of time in which a particular workload is executed (e.g., the run length of a benchmark) is denoted by TP, and the frequency at which the optimizer is run (i.e., time between two executions of the optimizer) is denoted by F. Therefore the number of time intervals in the time period TP, denoted by T, is calculated as:

$$T = \frac{TP}{F} \tag{5.6}$$

5.2 Objective Function

Given the current state of the system, the objective of the optimizer is to find an assignment of partition to nodes for time step $t+1$, i.e., to find the allocation matrix A^{t+1}. The optimal A^{t+1} is the one that minimizes the objective function, which is formally stated as:

$$\begin{aligned} \textit{minimize} \qquad & \sum_{j}^{H} \sum_{i}^{P} \left(|A_{ij}^{t+1} - A_{ij}^{t}| * m_i^t \right) \quad + \\ \epsilon * \quad & \sum_{j=1}^{N^{t+1}} | \left(\sum_{i=1}^{P} A_{ij}^{t+1} * l_i^t \right) + \gamma^{t+1} - \mu^{t+1} | \end{aligned} \tag{5.7}$$

The objective function is a sum of two parts:

- **Data Movement:** The first part of the objective function represents the total movement of partition data during the migration process.
- **Load Balance:** The second part of the objective function represents the standard deviation in load per node, summed over all the nodes in the cluster, and then multiplied by a small real number ϵ.

Note that there are two parameters in the objective function whose values need to be determined before running the optimization:

- $\mathbf{N^{t+1}}$: N^{t+1} is not present explicitly in the objective function, but it is used to calculate γ^{t+1} and μ^{t+1}. We set the value of N^{t+1} to the minimum number of nodes that can 'fit' the estimated load:

$$N^{t+1} = max\left(ceil\left(\frac{\sum_{i=1}^{P} l_i^t(1+\beta)}{C}\right), ceil\left(\sum_{i=1}^{P} l_i^t * \beta\right)\right) \tag{5.8}$$

This is done to minimize the usage of computing resources (i.e., the nodes used). As the value of N^{t+1} is set before the optimization is run, this is the 'actual' primary objective of our optimizer, while data movement and load balance are secondary considerations. The optimizer only considers those solutions that can fit in the minimum number of nodes required.

- **Epsilon**: Figure 5.2 demonstrates the effect of ϵ on data movement and load balance using a sample problem instance. If ϵ is set to a value less than the minimum sensitivity of the optimizer ($\epsilon < 10^{-6}$ in our case), the objective function only attempts to minimize data movement. In this case, there is intermediate amount of data movement and high variance in per host load. As the value of epsilon is increased, the objective function gives increasingly higher weight to the load balancing objective. For $10^{-6} \leq \epsilon \leq 10^{-2}$, the objective function achieves minimal data movement and intermediate load balance. This range represents the best range of values of ϵ, and is represented by the inset box in Figure 5.2. For values of $\epsilon > 10^{-2}$, the load balancing objective interferes with the data movement objective, resulting in minimal variance in per host load but substantial data movement. Based on these observations, we set the value of ϵ to 10^{-6} in our experiments, in order to prefer solutions that achieve minimal data movement and intermediate load balance.

The objective function is hence designed to ensure that all of the objectives of optimization (Section 4.2) are met. The value of N^{t+1} ensures that minimum computing resources are used, while the objective function ensures that data movement is minimized and load is balanced effectively. .

5.3 Constraints

Any feasible solution needs to satisfy the following constraints.

Active Nodes: The first N^{t+1} nodes must be active in the returned solution.

$$\sum_{i=1}^{P} A_{ij}^{t+1} > 0 \quad \text{for } j = 1, \ldots, N^{t+1} \tag{5.9}$$

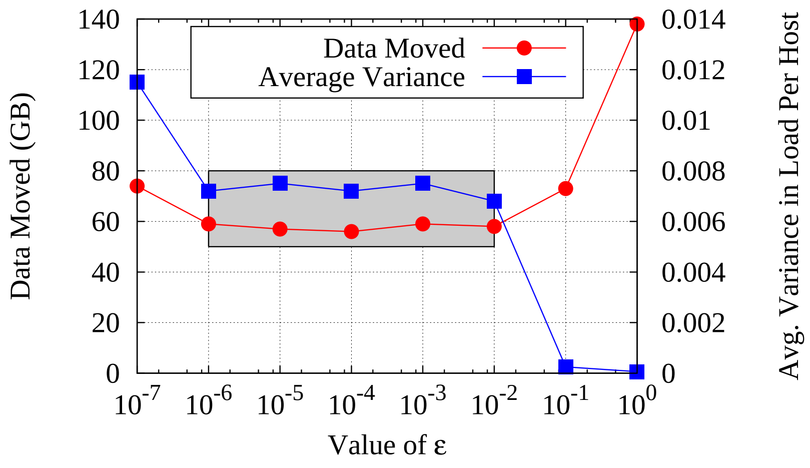

Figure 5.2: The effect of ϵ on data movement and load balance.

$$\sum_{i=1}^{P} A_{ij}^{t+1} = 0 \quad \text{for } j = N^{t+1} + 1, \ldots, H \tag{5.10}$$

Replication: Replicas of a given partition must be assigned to different hosts. As the the decision variables A_{ij}^{t+1} are binary, the sum of every row must be equal to K to ensure that the replication factor is preserved.

$$\sum_{j=1}^{H} A_{ij}^{t+1} = K \quad \text{for } i = 1, \ldots, P \tag{5.11}$$

CPU Capacity: The sum of the CPU loads of individual partitions placed on each node must not exceed the maximum processing capacity of that node.

$$\sum_{i=1}^{P} A_{ij}^{t+1} * l_i^t + \gamma^{t+1} \le C \quad \text{for } j = 1, \ldots, H \tag{5.12}$$

Memory Capacity: The sum of the memory required by individual partitions placed on each node must not exceed the memory available in that node.

$$\sum_{i=1}^{P} A_{ij}^{t+1} * m_i^t < M \quad \text{for } j = 1, \ldots, H \tag{5.13}$$

Serial Overhead: The serial overhead of each node must not exceed the capacity of a single core.

$$\gamma^{t+1} \leq 1 \tag{5.14}$$

5.4 Conversion to a Linear Program

We have now defined the optimization problem that would be solved by Elasca's optimizer. The constraints in this problem are *linear* constraints but the objective function is *non-linear* due to the presence of absolute values (the $|x|$ function). 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. Two constraints of the form $x - y \leq 0, -x - y \leq 0$ are additionally introduced. We used this technique to make the problem formulation linear. This MILP formulation can be solved efficiently using any one of many well known and widely available general purpose solvers. For this thesis we use IBM ILOG CPLEX [15].

5.5 Staggering Scale-In

As we show later in Chapter 6, the optimizer presented above works very well in most cases. However, in cases where the workload is highly fluctuating, with sudden spikes and troughs, the optimizer performs an excessive amount of data movement due to the instantaneous nature of its decision making. In this section, we present an enhancement which can be used to obtain more stable solutions by *staggering* the scaling in of nodes when the load fluctuates.

The idea behind this approach is that when the load fluctuates suddenly, the optimizer performs rapid scale-in and scale-out operations. We want to delay the scale-in operation till we are sure that the load is actually decreasing, and not fluctuating temporarily. To do this, we define a concept known as a *stability window*, i.e., a time window of s previous time steps. We use the same optimizer that we have defined above, but with one small change: we set the value of N^{t+1} to the maximum of the set $\{N^{t+1}, N^t, \ldots, N^{t-s+1}\}$. This approach delays the scale-in operation by s time intervals. This may mean that the system is over-provisioned for some time steps, but it helps avoid excessive scale-out and scale-in operations for highly fluctuating workloads. Scale-out and partition-shuffle (where the number of nodes is not affected but the assignment of partitions to node is changed) operations are not affected by the stability window, since we need to be responsive to load increases to avoid overloading the system.

The value of s determines the sensitivity of the optimizer to changes in the workload. If we consider a time interval of 15 minutes, then $s = 4$ means that the optimizer would wait for 60 minutes before performing a scale-in operation. If the value of s is set to 1, then the optimizer will behave in an instantaneous manner for scale-in operations. Setting the value of s too high would mean that the optimizer will be quick to scale out but very slow to scale in. The optimal size of the stability window is likely to depend on the size of a time interval and other workload-dependent characteristics. Finding this optimal value is left to future work.

Chapter 6

Experimental Evaluation

In this chapter, we show experimental results that demonstrate the effectiveness of Elasca's optimizer in maximizing resource utilization, minimizing data movement, and balancing load across nodes without sacrificing system throughput. The results of our experiments show that our optimizer can find solutions that are similar in quality to the solutions found by a workload-aware offline optimizer [30], and are superior to the solutions found by a simple greedy optimizer in terms of all the optimization objectives (Section 4.2). We demonstrate that the solutions obtained by the optimizer provides system throughput very close to a fully scaled out system, while requiring significantly less computing resources than the greedy optimizer. Furthermore, we show that Elasca's online optimizer is scalable to hundreds of partitions and nodes.

6.1 Experimental Setup

6.1.1 System Configuration

All our experiments were performed on 6 nodes of an IBM Blade Centre. Four of these nodes were used to run the VoltDB cluster, while the other two were used to run the benchmark clients. Each node has two AMD Opteron 2216 HE processors (Dual Core, 2.4 GHz) and 8GB of physical memory. All nodes are running OpenSUSE 11.3 (64-bit) with the Linux Kernel version 2.6.34-12. The version of VoltDB that we used is 1.3, modified to enable elastic scale-out as described in [31]. We use IBM ILOG CPLEX [15] to solve the mathematical optimization problem that is part of the online workload-aware optimizer.

6.1.2 Different Optimizers Compared in the Experiments

We compare all of the following optimizers in our experiments, unless otherwise noted:

- **ELASCA**: Elasca's workload-aware online optimizer described in Chapter 5.
- **ELASCA-S**: Elasca's optimizer with the staggering scale-in option enabled. We used a value of $s = 3$ (Section 5.5) for these experiments.
- **OFFLINE**: A workload-aware offline optimizer that knows the entire workload in advance. This optimizer was developed independently of this work as part of [30], and is presented here as a baseline to demonstrate the effectiveness of Elasca's optimizer. Details of this optimizer are presented in Appendix A.
- **GREEDY**: A greedy first-fit optimizer. We present this as an example of a naive optimizer, to demonstrate the benefits of our approach. The greedy optimizer starts by sorting all P partitions by their estimated CPU load. The optimizer then goes through the sorted list in descending order (i.e, the partitions with largest CPU load come first) and assigns partitions to nodes. When a node fills up, the greedy algorithm moves to the next node until all partitions are assigned. In order to make the greedy optimizer more "intelligent", we provide the same CPU load estimates to the greedy algorithm that our optimizer uses, and use a modified comparator function which ignores small differences ($\leq 5\%$) in CPU load (i.e., partitions that generate CPU load within 5% of each other are assumed to have the same load).
- **SCO**: As a baseline for evaluating system performance, we consider a fully scaled out cluster with no elasticity. This case uses the maximum number of nodes H, and the partition-to-node allocation is static. There is no data movement or load balancing performed. In terms of overall throughput, we do not expect to do any better than this case.

We run all our experiments for one hour (i.e., $TP = 60$ minutes) and we run each optimizer at a frequency of $F = 4$ minutes, resulting in a total of $T = 15$ time intervals (and decision points). Table 6.1 lists a set of parameters and their corresponding values which were used when running the optimizers.

Optimizer	Parameter	Value
All	Number of Cores (C)	4
	Memory/Node (M)	$8GB$
	Max Load (L)	100%
	Replication Factor (K)	1
	Time Period of Execution (TP)	60 minutes
	Frequency of Optimization (F)	4 minutes
	Number of Time Intervals (T)	15
	Maximum Number of Hosts (H)	4
OFFLINE	Server Cost (S)	1000
	Migration Cost (I)	150
ELASCA-S	Stability Window (s)	3 time intervals

Table 6.1: Optimizer parameters.

6.1.3 Benchmarks Used

In order to thoroughly test our optimizers, we use a variety of popular benchmarks. The results from these benchmarks highlight different characteristics of our optimizers and their effectiveness in obtaining the optimal solution in each case. The following benchmarks are used in our experiments:

1. **TPC-C**: The TPC-C benchmark [43] simulates the operations of a mid-size retailer. We make certain modifications to the TPC-C benchmark in order to make it cleanly partitionable and fit in memory. 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 experimental 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. We use a TPC-C database with a warehouse size of 300 MB. Since each warehouse is mapped to a unique partition, the total database size is equal to $P * 300$ MB.

2. **TATP**: The Telecommunication Application Transaction Processing (TATP) benchmark is designed to measure the performance of a relational DBMS in a typical telecommunication application [42]. We make slight modifications to the TATP benchmark in order to make it cleanly partitionable. Specifically, we ensure that all transactions use the *subscriber id* as the search key. We set the size of the *Subscriber* table to 1 million records, corresponding to a database size of 250 MB.

3. **YCSB**: The Yahoo! Cloud Serving Benchmark (YCSB) [12] is designed to evaluate the performance of various cloud data-stores. We use an unmodified version of YCSB with 1 million records, corresponding to a database size of 1 GB, and a 50-50 read/write ratio.

Table 6.2 shows the benchmark-specific values provided to the optimizer. As the table shows, the value of β for TPC-C was measured to be 0.32, indicating that each host can support transactions over 3 fully loaded partitions (i.e. $P/H = 3$ using Equation 5.2). However, for the TATP and YCSB benchmarks, the value of β was measured to be close to 1. This indicates that the serial part of the transactions is the bottleneck for these two benchmarks and hence $P/H = 1$ for these benchmarks. The reason for the difference between these two cases can be attributed to the nature of transactions in these benchmarks. TPC-C has relatively complex transactions, requiring reads and writes to multiple rows in multiple tables in each transaction, resulting in long-running transactions. In comparison, TATP and YCSB have relatively simple transactions consisting of single-row reads and writes. Therefore the serial transaction initiation thread consumes most of the transaction time while the actual transaction execution takes a very short time. Our problem formulation takes this difference into account, and the difference in the value of P/H reflects this. We will show that Elasca's optimizer can effectively produce scale out plans for these different cases.

6.1.4 Dynamically Changing Workloads

In order to test the performance of our optimizer we generated dynamically changing workloads for each of the three benchmarks mentioned above. We used two methods to create dynamism in the workloads:

Benchmark	**Parameter**	**Value**
TPC-C	Max Transactions/Partition (α)	4500 TPS
	Serial Overhead (β)	0.33
	Partition/Host Ratio (P/H)	3
TATP	Max Transactions/Partition (α)	32500 TPS
	Serial Overhead (β)	0.91
	Partition/Host Ratio (P/H)	1
YCSB	Max Transactions/Partition (α)	30000 TPS
	Serial Overhead (β)	0.88
	Partition/Host Ratio (P/H)	1

Table 6.2: Benchmark-specific parameters.

1. **Aggregate Request Rate**: We varied the aggregate request rate over time using periodic waveforms (triangle, sine, square and sawtooth) of varying frequencies (denoted by f). For each waveform, we ran a number of tests with the frequency varying from 0.5 to 4 cycles per hour. We used open-loop clients to generate the load for our experiments. The load on each partition was dynamically varied using the different skew distributions provided next.

2. **Load Skew:** We generated *temporal skew* in the distribution of load per partition using a variety of statistical distributions. While in the case of *static skew* the distribution of load across partitions remains static throughout, in temporal skew the distribution of load slides across all the partitions, depending on the time interval. For example, if we use a Zipfian distribution to temporally skew the load among partitions, then initially partition 1 will get the most load, then partition 2, and so on until all the partitions have received the most load (i.e., the load distribution 'slides' across all the partitions).

   An important point to consider is that the database system saturates faster when the load is skewed, versus when it is uniform. This is because the system is unable to reach its maximum throughput capacity, as some partitions receive load higher than they can sustain, while others receive less than their maximum capacity. This phenomenon is demonstrated in Figure 6.1, which shows the throughput over time for a TPC-C workload in which we varied the aggregate request rate using a sine wave. The figure demonstrates how load skew affects the maximum throughput capacity of a fully provisioned VoltDB cluster having four nodes. The figure shows that the system is able to sustain a higher amount of load when there is no load skew (i.e., the load distribution is uniform). On the other hand, the system saturates more quickly

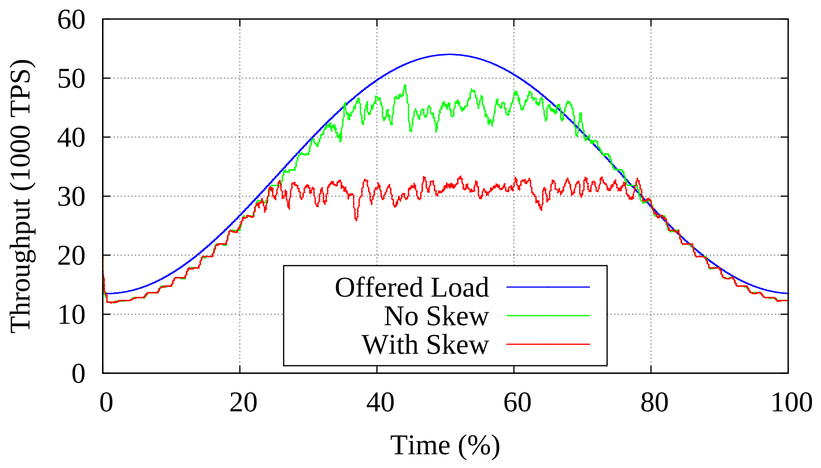

Figure 6.1: Maximum throughput capacity with and without load skew.

in the presence of load skew (we used a normal distribution to skew the load for this case). We do not show the offered load on the remaining figures in this chapter, as the aim of Elasca's optimizer is to achieve the performance of a fully provisioned system in each case, and not to mitigate the effects of load skew. Hence, we compare the performance of Elasca with the performance of a fully provisioned system, rather than with the offered load.

The distributions we used to skew the load, along with their parameter values are summarized in Table 6.3. We discuss them in greater detail below:

UNIF: The aggregate transactional load is evenly distributed across all the partitions throughout the run.

CATG: The aggregate transactional load is distributed among partitions using a Categorical Distribution with the set of probabilities p indicating the likelihood of a partition getting a request. For example, according to the value of p given in Table 6.3, one fourth of the partitions will get 10% of the requests, one fourth will get 20%, and so on. The probability at which the partitions receive requests slides among partitions with respect to time to generate temporal skew.

NORM: The aggregate transactional load is distributed among partitions using a Normal Distribution. The standard deviation of the distribution is denoted by σ. The mean varies among all partitions with respect to time to general temporal skew.

ZIPF: The aggregate transactional load is distributed among partitions using a

Abbreviation	**Distribution Name**	**Parameters**
UNIF	Uniform	N/A
CATG	Categorical	$p = 0.1, 0.2, 0.3, 0.4$
NORM	Normal	$\sigma = P/4$
ZIPF	Zipfian	$z = 0.5$

Table 6.3: Statistical distributions used for load skew.

Zipfian Distribution. The value of the exponent characterizing the distribution (i.e., the skew parameter of the Zipf distribution) is represented by z. The probability of requests for each partition returned by the Zipfian distribution slides among all the partitions with respect to time to generate temporal skew.

6.2 Optimizer Effectiveness

In this section, we present a comparison of three optimization objectives that directly relate to the quality of the solutions produced by our optimizer, i.e., data movement, minimization of computing resources used, and load balance. We establish the effectiveness of our optimizer through multiple simulated runs where we vary different parameters and demonstrate that the optimizer reacts effectively in each case. All the experiments in this section use the TPC-C benchmark, and all were run with the default values of $T = 15$ time intervals and an optimization period of $F = 4$ minutes per interval. Thus, each experiment runs for one hour. We ran experiments involving different number of partitions ($P \in \{16, 32, 64\}$) and skewed load distributions (Table 6.3). We used a triangle wave function with two different frequencies ($f \in \{1, 4\}$ cycles per hour) to vary the aggregate request rate between 25% and 100% of the combined maximum request rate of P partitions (i.e., 100% load is equal to $P * \alpha$ requests per second). We use simulation for these experiments to validate that the optimizer meets the objectives that it should meet, and we report results from actual runs in Section 6.3.

6.2.1 Minimizing Data Movement

We demonstrate that our optimizers effectively minimize data movement in this section. Figure 6.2 shows the amount of data movement performed by the optimizers for $P = 16$ and $f = 1$ case. Our optimizer outperformed GREEDY by a large margin for every load

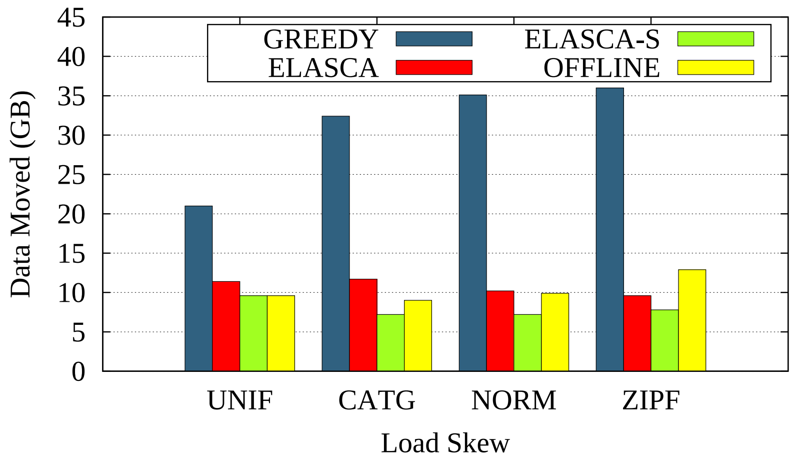

Figure 6.2: Amount of data moved by the optimizers for various skewed load distributions.

distribution, resulting in 63% less data movement on average for ELASCA, and 72.5% for ELASCA-S. Thus, we can see that the staggering scale-in enhancement is effective at reducing the amount of data movement that needs to be performed. On average, ELASCA-S performs 21% better than OFFLINE and 25% better than ELASCA in terms of data movement. OFFLINE offers a slight advantage over ELASCA in most cases. However, it does worse than ELASCA in a few cases. The offline optimizer solves a more complex problem than Elasca's online optimizer, so it can take a very long time to find the optimal solution. Because of that, the offline optimizer is run with a time limit, and if it cannot find the optimal within this time limit it returns the best solution it could find. In the experimental runs where the offline optimizer does worse than Elasca, the solution that the offline optimizer returns in the given time limit is slightly sub-optimal.

Figure 6.3 shows how the optimizers perform with increasing number of partitions (database size) for the Zipfian distribution when we use a triangle wave with $f = 1$ cycle per hour to vary the aggregate request rate. We do not include OFFLINE for $P = 64$ in Figure 6.3 as we were unable to get a sufficiently optimized solution for this case even after running the optimizer for 24 hours. As the figure shows, GREEDY performs excessive data movement as the database size increases. For $P = 64$, ELASCA performs 73% less data movement than GREEDY on average, while ELASCA-S performs 79% less data movement.

To further demonstrate the benefit of staggering scale-in, we consider a case where the workload is fluctuating rapidly. We use $P = 16$ partitions and set the frequency of

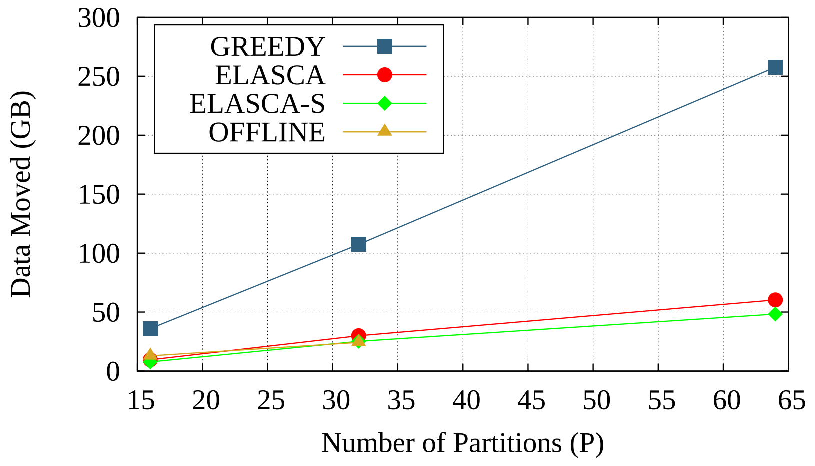

Figure 6.3: Data movement performed by optimizers with increasing database size.

the triangle wave to four ($f = 4$ cycles per hour). Thus, each 'wave' takes 15 minutes, corresponding to almost 4 decision points per wave. Figure 6.4 shows the difference in data movement between the optimizers for this case. On average, ELASCA-S does 82% less data movement than GREEDY, 57% less than ELASCA and 53% less than OFFLINE. This shows that by delaying the scale-in operation we can get significant savings in data movement for fluctuating workloads, at the cost of using slightly more computing resources, as we show in the next section.

6.2.2 Saving Computing Resources

We now show the effectiveness of our optimizer in saving computing resources. We use a triangle wave with $f = 1$ cycle per hour for these experiments. Figure 6.5 shows the average percentage of computing resources saved by each optimizer (averaged over different number of partitions, i.e., $P = 16$, 32, and 64) when compared to a fully scaled out case (SCO). IDEAL represents the maximum savings possible, using an optimal solution with non-integer assignments to the matrix A^t, and without any feasibility checking (recall that all the elements of A^t in a valid solution must be 0 or 1). ELASCA performs much better than GREEDY for cases when the workload is skewed, saving up to 14% more computing resources. ELASCA is able to match IDEAL when the workload is skewed. For the UNIF case, the optimizers utilize slightly more computing resources as it is infeasible to fit the partitions into the number of nodes provided by a non-integer optimal solution.

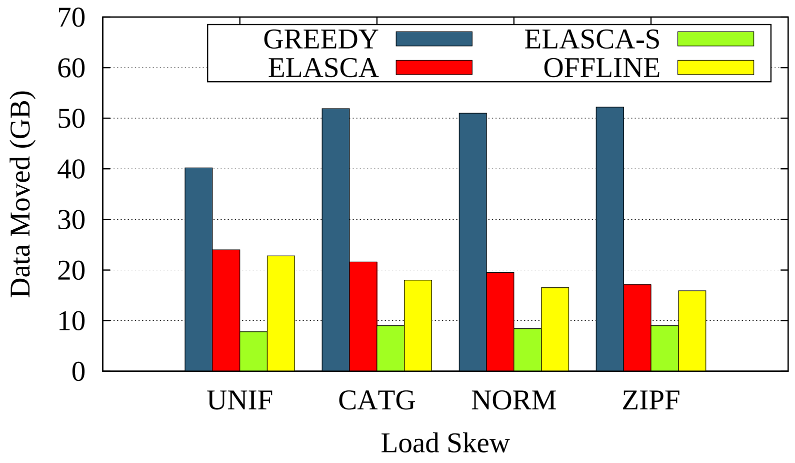

Figure 6.4: Data movement performed by optimizers for a fluctuating workload.

ELASCA-S uses comparatively more computing resources than ELASCA and OFFLINE as it staggers scale-in, resulting in slightly more conservative resource savings. This is the price that ELASCA-S pays to reduce data movement. However, ELASCA-S still offers greater savings than GREEDY.

6.2.3 Load Balance Across Nodes

In this section, we demonstrate that the solutions provided by our optimizer are evenly balanced across nodes. This helps minimize the amount of data movement that needs to be performed in future time intervals. Figure 6.6 presents the average variance in CPU utilization of each node for the $P = 16$, $f = 1$ case. Each bar in the figure shows the average of 15 variance values, computed at each decision point (i.e., every 4 minutes). GREEDY has 4.2× higher variance on average than ELASCA, and 3.7× higher than ELASCA-S. The difference between ELASCA, ELASCA-S and OFFLINE in terms of load balancing is not statistically significant, except for the ZIPF distribution where OFFLINE performs better.

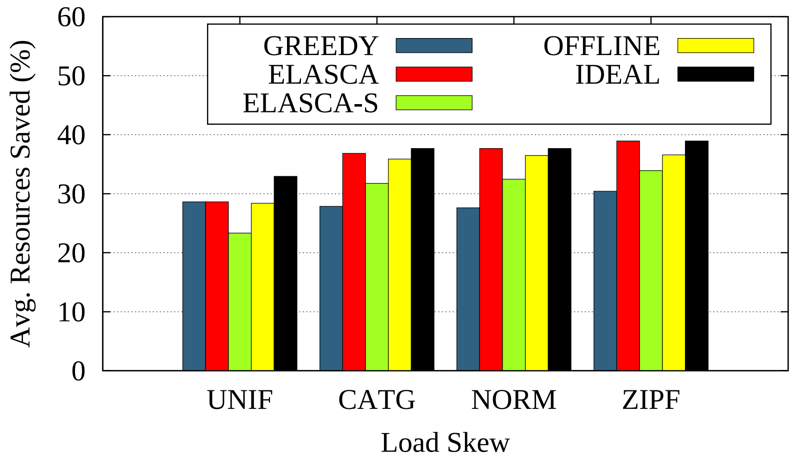

Figure 6.5: Average percentage of computing resources saved.

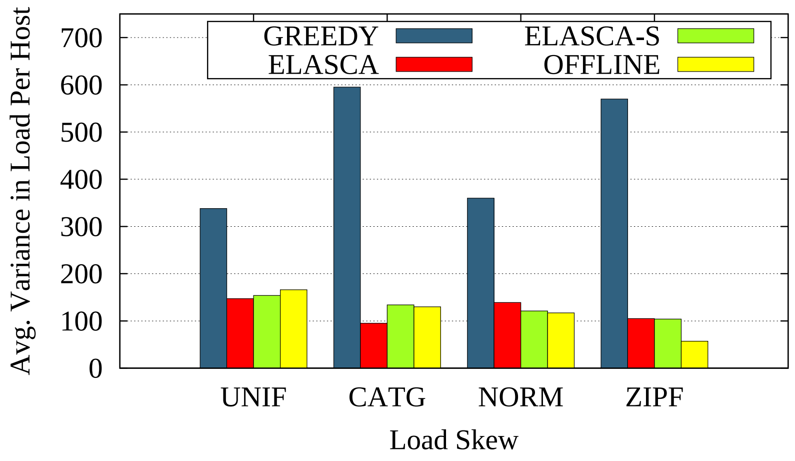

Figure 6.6: Average variance in CPU utilization of each node.

6.3 Overall Throughput

After establishing the effectiveness of our optimizer through simulations, we now present results from experiments on an actual VoltDB cluster with $H = 4$. Using the benchmark-specific partition-to-host ratios given in Table 6.2, we can support a maximum of $P = 12$ partitions for TPC-C and $P = 4$ partitions for the other two benchmarks.

For the first set of experiments, we vary the request rate between 25% to 100% of the cluster capacity, using a sine wave with $f = 2$ cycles per hour, and use different skewed loads. For each experiment, we measure the overall throughput in terms of transactions per second (TPS), averaged over the duration of the experiment. Each experiment is one hour long and we set the value of $T = 15$, where each time interval is four minutes long ($F = 4$ minutes). We run the same set of experiments for all the three benchmarks. We do not report TPS numbers for ELASCA-S in this experiment as its numbers were similar to ELASCA. The results are presented in Figures 6.7 (a) to (c). The y-axes represent the average transactions per second (TPS) while the load skew distributions are presented on the x-axes.

As the figures show, our optimizers perform very close to SCO. In the best case, the performance of ELASCA is within 1% of the optimal, while in the worst case it differs by 6.5%. OFFLINE behaves similar to ELASCA in terms of overall throughput. GREEDY manages to perform reasonably well for this case too, as Elasca's elastic scale-out mechanism incurs low data migration cost. Best case and worst case performance for GREEDY is 2.5% and 14.5% worse than the optimal, respectively.

Figures 6.8 (a) to (d) present a deeper analysis of the optimizers' performance. The figures show the throughput over time for the TPC-C benchmark as the request rate is varied using a sine wave with $f = 2$ cycles per hour. We use the NORM load skew case for these figures. The x-axis represents time in percentage, while the y-axis represents database throughput. The drops in throughput represent partition migrations performed by the optimizers. As the figure shows, OFFLINE offers the most stable database performance, while ELASCA provides much more stable performance in comparison to GREEDY. SCO does not perform any data migration so its performance is the most stable, but it represents an inelastic, overprovisioned system.

Next, we demonstrate that the model we use to estimate the CPU load from the observed request rates works effectively. Figure 6.9 shows the measured and predicted CPU utilization of the hosts over time for the TPC-C benchmark using ONLINE, under the NORM load skew case. Our model effectively predicts the increase and decrease in CPU utilization accurately, which is what our optimizers need when deciding to scale out/in.

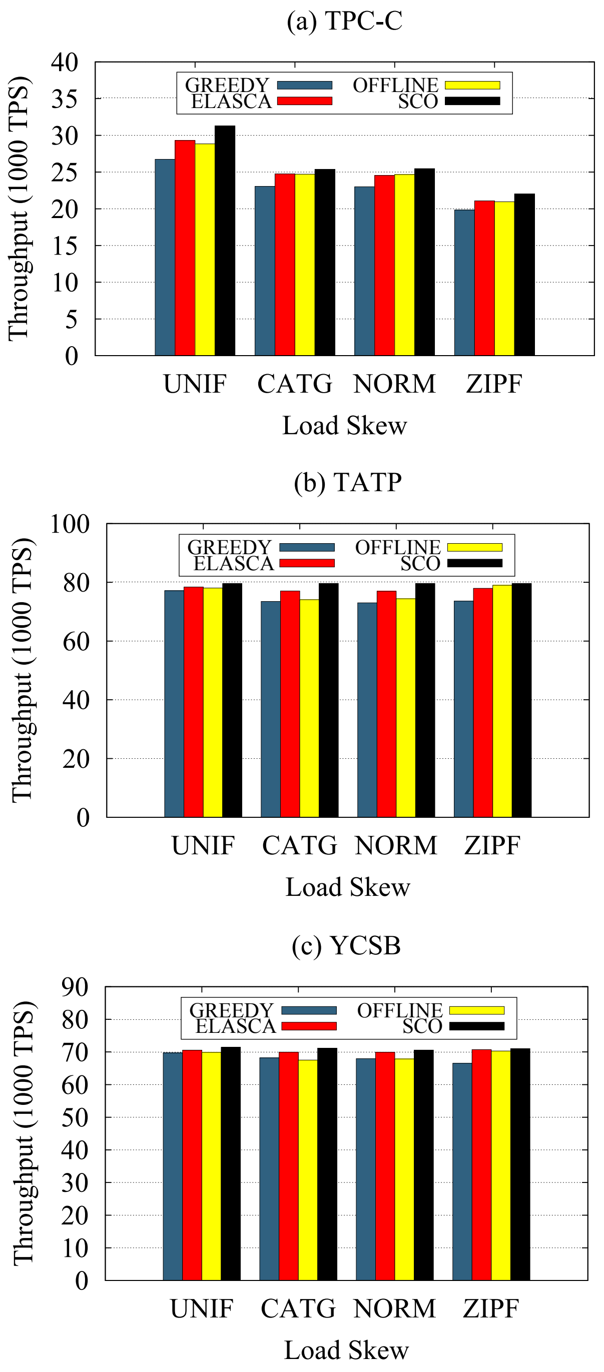

Figure 6.7: Overall throughput for the TPC-C, TATP and YCSB benchmarks.

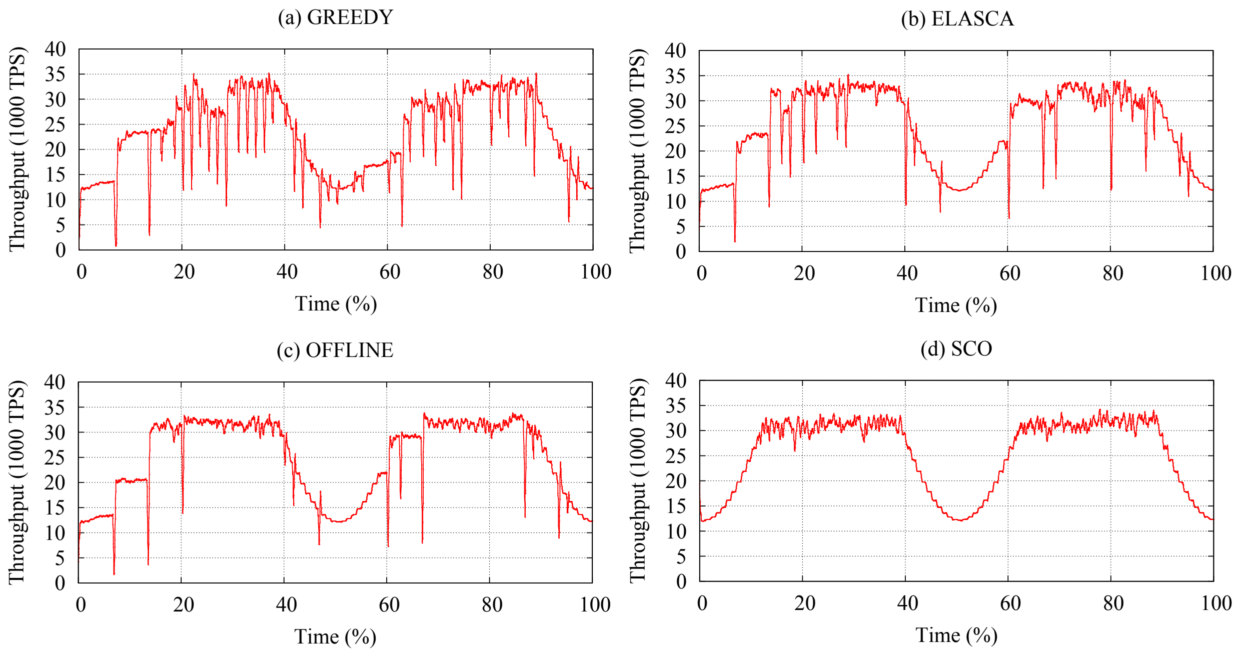

Figure 6.8: Throughput over time of the optimizers for the TPC-C benchmark (NORM load skew).

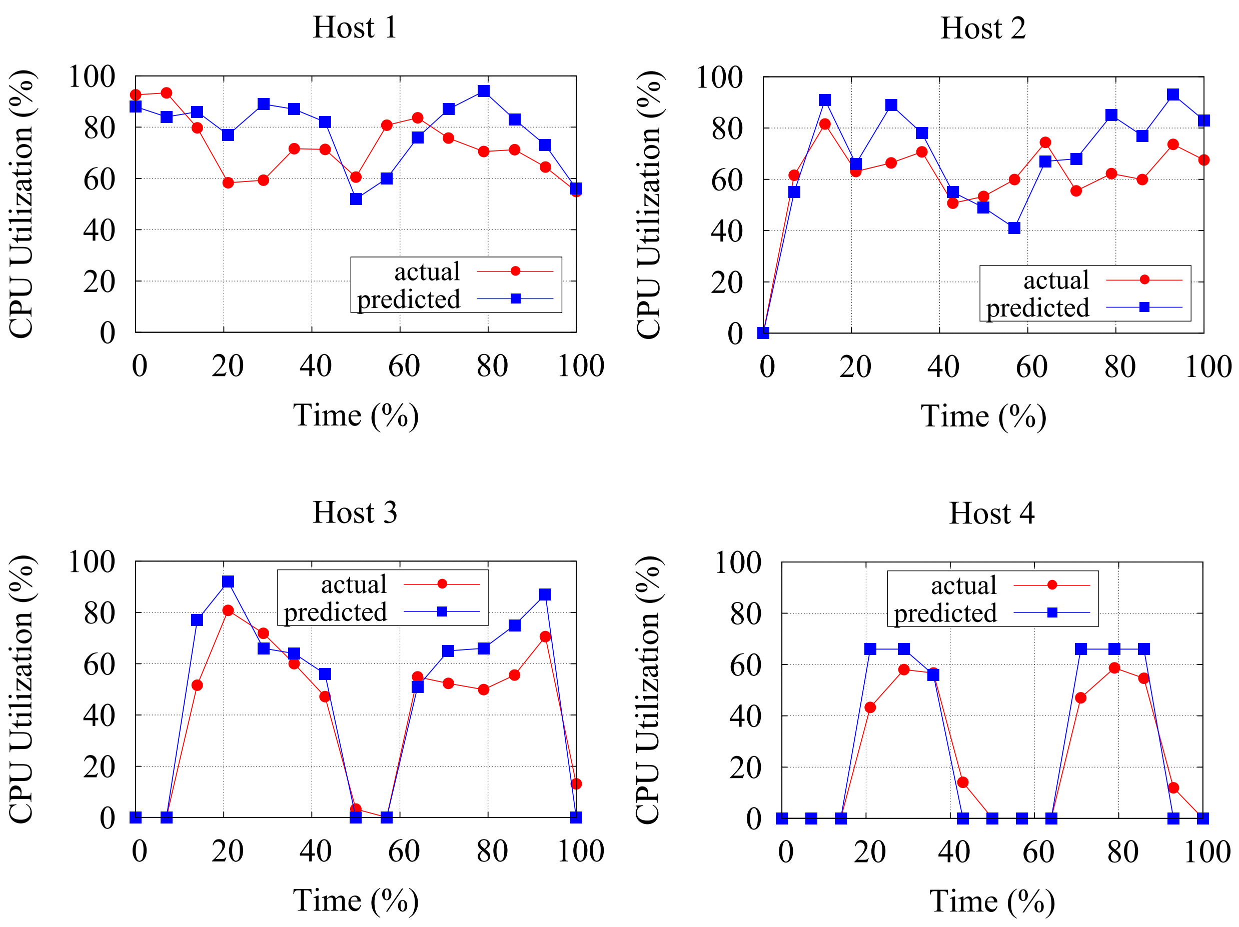

Figure 6.9: Measured and predicted CPU utilization of the hosts over time for ELASCA (NORM load skew).

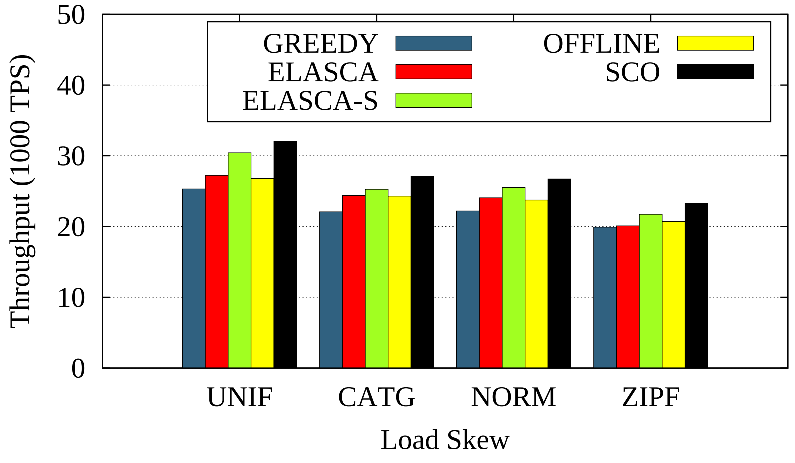

Figure 6.10: Average throughput for the TPC-C benchmark, for a fluctuating workload.

We present results from one further set of experiments in which we vary the load more quickly for the TPC-C benchmark. For these experiments, we vary the request rate using a triangle wave with $f = 4$ cycles per hour. As before, each experiment is an hour long with fifteen time intervals. We present the overall throughput using the TPC-C benchmark in Figure 6.10. The y-axis denotes the average transactions per second (TPS) while the different load skew distributions are shown on the x-axis.

Figure 6.10 shows that ELASCA-S handles short-term fluctuations in load better than ELASCA or OFFLINE. The performance of ELASCA-S is 5.7% worse than SCO on average, while ELASCA and OFFLINE are around 12.2% worse off. GREEDY performs 17.7% worse. Therefore, in order to deal with rapid changes in load effectively, ELASCA-S with a small stability window is recommended. This choice is not the best in terms of computing resources saved, but it strikes a good balance between computing resources on one side and throughput, data movement, and responsiveness to load increases on the other side.

6.4 Scalability of the Optimizer

As ELASCA is meant to be used in an online setting, it must be able to return a solution quickly for any case. In this section, we show that ELASCA effectively manages to come up with a solution for large problem sizes quickly. In order to ensure that the optimizer

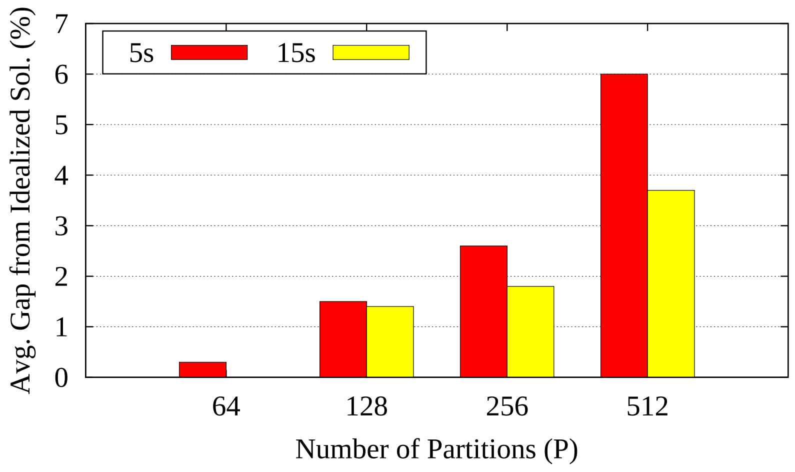

Figure 6.11: Increase in the amount of computing resources used by ELASCA when compared to a fractional idealized solution.

finishes quickly, we set a time limit for the solver to find a solution for the given optimization problem. We also tell the solver to only return solutions that are close to optimal (i.e., within 1% of optimal). If an optimal or near-optimal solution is not found in the given time limit, we increment the value of N^{t+1} in the objective function and run the solver again, as it is easier for the solver to find a valid solution with greater number of nodes (i.e., partitions do not need to be packed as closely). We repeat this process until a solution is found. The solver usually finds a solution within the first or second attempt.

The time limit for the solver is flexible and can be increased or decreased as required without impacting the solution quality significantly. We have experimentally found that most optimizations finish within the set time limit, particularly for smaller number of partitions ($P < 100$). As the number of partitions increases further, there are more time-outs and hence a slightly higher number of nodes than the minimum are used. Figure 6.11 illustrates this phenomenon. The figure measures the quality of the solution found by ELASCA for the TPC-C benchmark. For a time limit of 5 seconds, the optimizer uses up to 6% more resources than a fractional (i.e., non-integer) idealized solution in the worst case. When the time limit is increased to 15 seconds, this value falls to 3.7%. In most practical scenarios, the number of database partitions P is typically small and therefore we do not expect the time limit to be a significant issue.

Figure 6.12 shows the scalability of ELASCA with increasing number of partitions. We

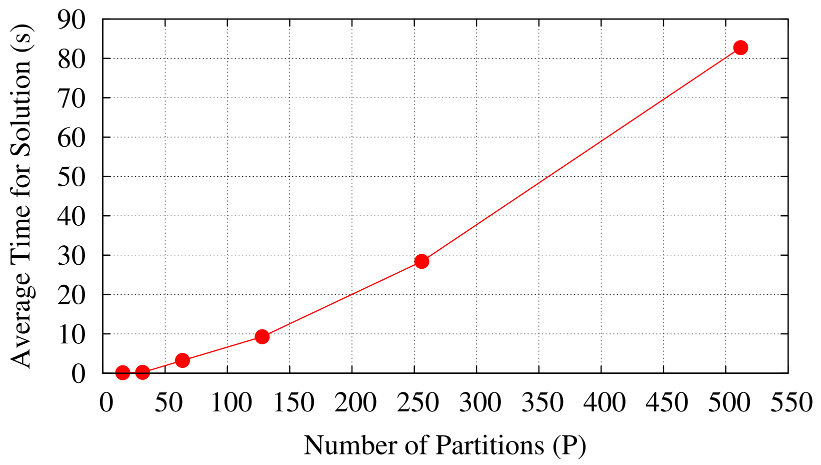

Figure 6.12: Time taken by ELASCA to find solution with increasing problem size.

report the average time it took for the optimizer to come up with a valid solution with a 15 second time limit. We ran the optimizer for a large number of problems with different load distributions and request rates. As the figure shows, on average ELASCA returns a solution within 80 seconds for up to 512 partitions. The relatively small optimization time required by ELASCA makes it particularly suitable for use in an online setting.

In order to provide comparison with ELASCA, we also measured the scalability of GREEDY and OFFLINE. As the GREEDY optimizer is implemented as Java code and does not use any mathematical optimization techniques, it returns a solution very fast (less than 1 second) for all problem sizes.

We evaluated the scalability of OFFLINE by fixing the optimization time limit to four hours and running the optimizer for a number of workloads with increasing number of partitions. Figure 6.13 shows that as the number of partitions (problem size) increase, the solutions returned by OFFLINE degrade substantially with respect to optimality. The gap shown on the y-axis of the figure represents the percentage gap between the value of the objective function for the solution returned by the solver, and the known minimum value among the sub-problems yet to be solved (i.e., *absolute gap* in CPLEX). The solutions are acceptable for smaller problem sizes ($P \in \{16, 32\}$). However, for larger problem sizes ($P \in \{64, 128\}$), the gap between the value of the objective function returned by OFFLINE and the global optimum becomes unacceptably large. Therefore, OFFLINE is not a practical choice for use in large partition migration problems.

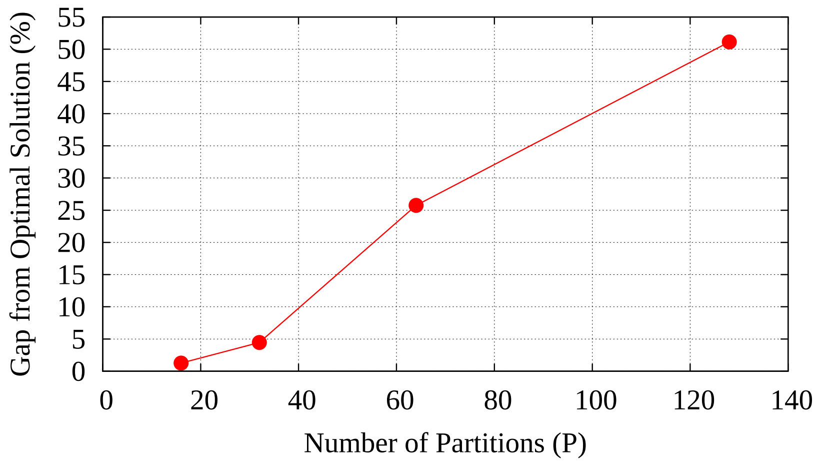

Figure 6.13: Quality of the solution returned by OFFLINE with increasing problem size.

As our experimental results demonstrated earlier, ELASCA manages to achieve a level of performance equal to OFFLINE for most cases. Coupling this with the fact that OFFLINE returns sub-optimal solutions for large problem sizes results in the conclusion that ELASCA is the better choice for most scenarios.

Chapter 7

Supporting Multi-Partition Transactions

In this chapter, we consider the scenario where the database is not cleanly partitionable and multi-partition transactions (*MPTs*) cannot be entirely avoided. Our aim is to provide insight into the problem of partition placement in the presence of MPTs. We do not present a comprehensive solution to this problem; rather, we provide some directions for future work in this area.

This chapter is organized as follows. First, we present the changes that need to be made to our transaction model in order to take MPTs into account. Next, we make changes to the model we use to predict the CPU load of each partition, based on changes in the transaction model. After that, we discuss the impact of MPTs on database throughput through a number of simulations. Finally, we compare the performance of Elasca's optimizer with and without MPTs.

7.1 Modifications to the Transaction Model

In order to support MPTs, we need to modify the transaction model presented in Section 4.3. The transaction model presented earlier made an implicit assumption that all transactions are single-partition, as the initiator simply forwards the transaction request to the respective execution site responsible for a partition. We make the following changes to the model to incorporate MPTs:

1. **Transaction Initiation and Coordination:** In the case of MPTs, the initiator forwards the transaction request to a randomly selected execution site which acts as the *coordinator* for the entire transaction. The coordinator then sends transaction work to the relevant execution sites responsible for the partitions that are present in the transaction request. The transaction cannot be completed until the coordinator has heard back from all the execution sites. This incurs additional network round-trips, as well as delays since some sites may respond slower than others due to pending transactions, and the coordinator will need to wait for the slowest site. Once the coordinator has aggregated the responses, it sends a message to all the execution sites involved in the transaction, telling them to commit or roll-back the transaction work performed. The initiator is then notified of the response, signifying that the transaction is complete.

2. **Transaction Execution:** As execution sites also act as coordinators for MPTs, they cannot perform transaction work in an uninterrupted manner as in the case of single-partition transactions.

The new transaction model closely reflects VoltDB's actual transaction model for MPTs.

7.2 CPU Load Model Modifications

Elasca models the CPU load l_i^t generated by a partition i at time t by calculating the ratio of current request rate against that partition (r_i^t) with the maximum number of single-partition requests a partition can satisfy (α). This calculation is shown in Equation 5.3 and is reproduced below.

$$l_i^t = min(r_i^t/\alpha, 1) \tag{7.1}$$

This estimate is adequate for the case when all the transactions are single-partition. However, if we want to introduce multi-partition transactions, we need to model the effect of MPTs on the generated CPU load. In the new case, the load on each partition is equal to the sum of (a) the load of the transaction work performed by that partition (as in the case of single-partition transactions), and (b) the load of coordinating multi-partition requests.

For (a), we can simply assume that each single-partition transaction counts as one transaction against its respective partition, while each multi-partition request counts as one transaction against each of the partitions it involves. Thus, r_i^t represents the sum of the single-partition and multi-partition requests per second, for partition i during time interval t.

To include (b) in the equation, we need to introduce additional notation. Let η represent the maximum multi-partition transaction throughput of a partition. This throughput represents the number of MPTs that the partition can *coordinate* in a second. Let q_i^t denote the number of MPT coordination requests partition i receives, per second, during time interval t.

We can now modify Equation 7.1 to include both (a) and (b):

$$l_i^t = min\left(\frac{r_i^t}{\alpha} + \frac{q_i^t}{\eta}, 1\right) \tag{7.2}$$

This equation reflects the effect of multi-partition transactions on the CPU load generated by a partition. We use this equation to demonstrate the impact of multi-partition transactions on system performance in the next section.

7.3 Effect of Multi-Partition Transactions on Database Throughput

There are are multiple factors related to multi-partition transactions that can affect database performance, including:

1. **Maximum MPT Throughput (η):** Database performance is dependent on the maximum MPT throughput of each partition. Informally, if each execution site can coordinate a greater number of MPTs per second, the database performance is likely to be better.

2. **Probability of Multi-Partition Transactions:** Database performance also depends on the percentage of MPTs among all transactions. Database performance will suffer with greater percentage of MPTs. We represent this value with p_{mpt}.

3. **Partitions involved in MPTs:** The number of partitions involved in the MPTs can also impact database performance. It is likely that an MPT involving a greater number of partitions will take longer, as it depends on the response of more execution sites.

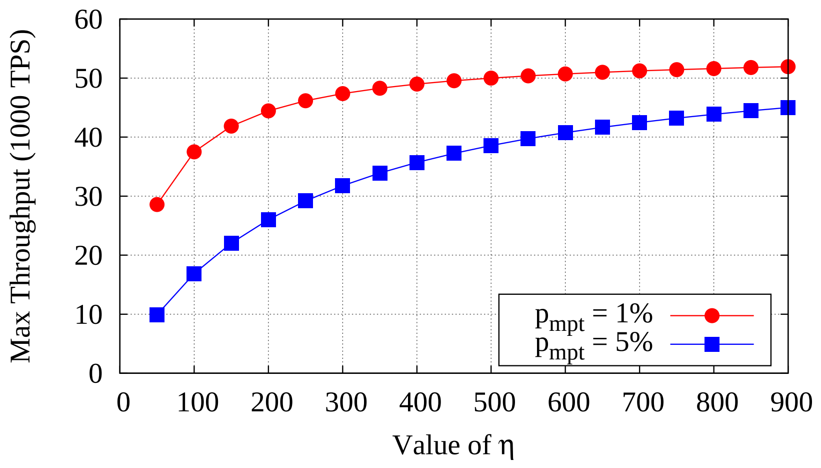

Figure 7.1: Effect of η on maximum throughput possible.

We ran simulations based on Equation 7.2 to demonstrate the impact of the first two factors on database performance. For these simulations, we set the number of partitions (P) to 12, and use a value of $\alpha = 4500$ (same as TPC-C). We assume that there is no load skew. The maximum possible throughput for these values of P and α is $P * \alpha = 54000$ TPS, for the case of single-partition transactions. The results of the simulation are presented below.

7.3.1 Maximum MPT Throughput

Figure 7.1 shows the effect of η on maximum database throughput possible. We varied the value of η between 50 and 900 in increments of 50, and calculated the maximum throughput possible for $p_{mpt} = 1\%$ and 5%. For $p_{mpt} = 5\%$ and low values of η, the maximum database throughput possible is particularly poor (around 12500 for $\eta = 50$), but the performance improves with greater values of η. As expected, the database performance is better with a lower percentage of MPTs ($p_{mpt} = 1\%$).

7.3.2 Probability of Multi-Partition Transactions

Figure 7.2 shows the effect of increasing the probability of MPTs on maximum database throughput possible, for two values of η. Database performance drops quickly, with even

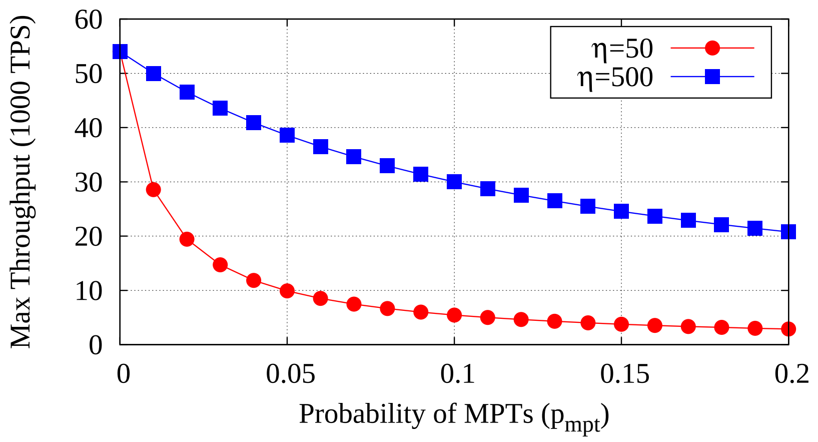

Figure 7.2: Effect of p_{mpt} on maximum throughput possible.

a small percentage of MPTs, for $\eta = 50$. However, for $\eta = 500$, the performance drop is much less severe.

7.4 Performance of the Optimizer with MPTs

In this section, we present the performance of Elasca's workload aware optimizer with the CPU load model modifications presented in Section 7.2 to support MPTs. We compare the performance of the optimizer without any MPTs (ELASCA) and with 5% MPTs (ELASCA-MPT). We examine the amount of data movement performed by the optimizer as well as the computing resources saved in both cases. For these simulations, we use $P = 16$, $\eta = 100, T = 15$, and $\alpha = 4500$. We use a sine wave with $f = 1$ to vary the overall request rate, and present a comparison of all four load skew distributions mentioned in 6.1.4.

7.4.1 Computing Resources Saved

Figure 7.3 shows the percentage of computing resources saved by ELASCA and ELASCA-MPT for various load distributions. ELASCA saves 2 to 3× more computing resources

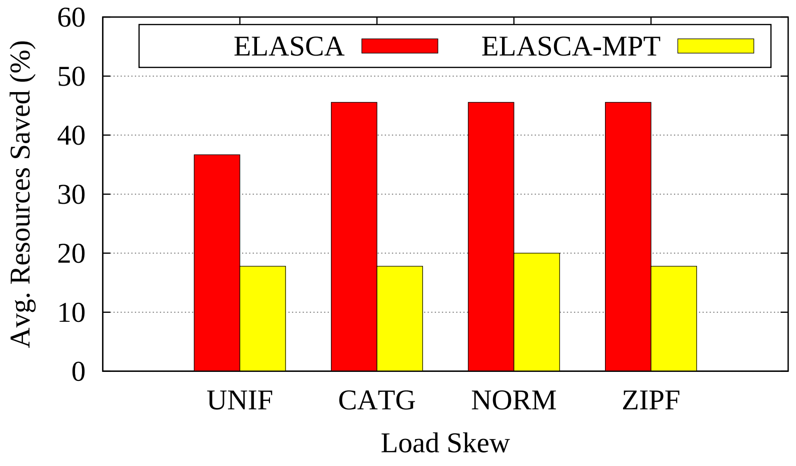

Figure 7.3: Computing resources saved by ELASCA and ELASCA-MPT.

than ELASCA-MPT. The workload for ELASCA in this figure has no MPTs, so it places less load on each partition, hence the greater savings.

7.4.2 Data Movement

The amount of data moved by ELASCA and ELASCA-MPT for various load distributions is shown in Figure 7.4. ELASCA-MPT moves less data than ELASCA for all the cases. This is because ELASCA-MPT scales out the database system to full capacity quickly, and keeps it scaled out almost till the end of the experimental run. On the other hand, ELASCA scales out the system more slowly as the CPU load on the partitions increases in smaller increments and the CPU capacity is not fully exhausted until the load is almost maximum (i.e., the sine wave is at its *crest*).

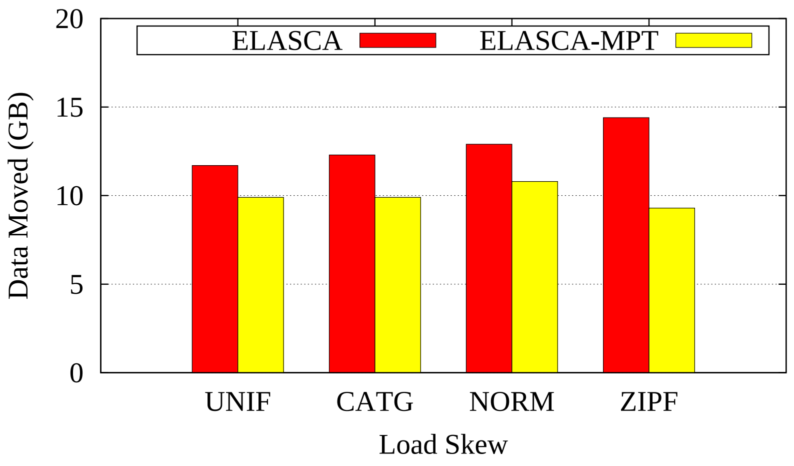

Figure 7.4: Data moved by ELASCA and ELASCA-MPT.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

In this thesis we described Elasca, a system that provides automatic elastic scalability using a shared-nothing partition based parallel database system, namely VoltDB. We presented the required changes that can be incorporated in a system like VoltDB to make it elastically scalable. The elastic scale-out mechanism enables the efficient addition/removal of nodes on-the-fly as well as redistribution of database partitions among these nodes. We further discussed the partition placement and migration problem in detail and presented a workload-aware online optimizer for solving the problem in an elastically scalable partitioned database system. Our optimizer utilizes mathematical optimization techniques to obtain optimal solutions that minimize the cost of computing and network resources, while ensuring a high level of system performance.

We evaluated many aspects of our optimizer, such as its effectiveness in finding optimal solutions with respect to data movement, computing resources used and load balance. We tested the quality of our optimizer by solving a large number of problems with varying load skew distributions and request rates, and running experiments on a VoltDB cluster to measure database performance. Our experiments show that the solutions found by our optimizer perform nearly optimally in terms of database throughput and offer excellent resource utilization. Furthermore, we evaluated the scalability of the Elasca optimizer and found that it manages to solve problems of all sizes that we tried quickly, as required for an online setting. We further demonstrated that the solutions found by our optimizer are comparable in quality to an offline optimizer that has full knowledge of the entire workload, including future requests. Therefore, Elasca's workload-aware optimizer offers a reasonable solution for dealing with large problem sizes, both in online and offline scenarios.

8.2 Suggestions for Future Work

8.2.1 Changes in VoltDB

The latest version of VoltDB (version 3.0) makes a number of changes in system architecture. These changes also need to be addressed in the elastic scale-out mechanism in order to make it compatible with new and upcoming versions of VoltDB. Some of the fundamental changes in VoltDB that affect the implementation of elastic scale-out are given below:

- **Inter-Node Coordination:** The current version of VoltDB uses Apache ZooKeeper [4] for inter-node coordination and keeping track of cluster-wide properties, while the version used for in this thesis relied on a custom communication framework for this purpose. The use of ZooKeeper simplifies keeping track of system changes, so it is likely to be easier to support elastic scale-out in the new version of VoltDB.

- **Master/Slave Partitions:** There have been a number of changes in the way transactions are handled in VoltDB. In the new version, reads are load balanced across all partition replicas, while writes are sent to the *master* partition first and then propagated to *slave* partitions. This master/slave replication was not present in earlier versions.

- **Client Affinity:** Client affinity is a feature of VoltDB that enables the client to route a transaction to the node containing the master partition when the transaction is a write, and to load balance across all the replicas if the transaction is read. Client affinity reduces the network load the cluster experiences during transactions as compared to previous versions of VoltDB. Clients in the previous versions of VoltDB were oblivious to partition placement and used a round-robin scheme to direct transaction requests to nodes. The new request routing approach reduces network load, but it requires that clients know where partitions reside. This requirement slightly complicates elastic scale-out as the clients need to be informed every time any partitions are moved.

8.2.2 Multi-Partition Transaction Support

An important direction of future work is supporting multi-partition transactions in Elasca and experimentally quantifying their effect. We briefly discussed multi-partition transactions in Chapter 7 and presented a possible method of incorporating them in our system. However, there are still many open problems in this area.

One such problem is determining which partitions a particular multi-partition transaction touches. This information can be used to coalesce those partitions together on a single node to support faster multi-partition transactions.

Another avenue of research is looking at other possible transaction models for supporting multi-partition transactions. The current transaction model puts the load of coordination on execution sites; it is possible to separate the coordination of transactions from execution sites and rather have a separate multi-partition transaction coordinator that runs exclusively on each node.

8.2.3 Automated Collection of Satistics and Parameter Tuning

Elasca currently requires the database administrator to collect a number of statistics about the workload and provide them to the optimizer. This method of system tuning is error-prone. A better approach would be to have the system collect these statistics by running an automated test and setting the parameters accordingly.

Another problem with the current approach is that workload characteristics can change over time and become outdated, resulting in the optimizer making inaccurate decisions based on old data. Therefore it is better for the system to periodically update workload statistics and use the new statistics to make future decisions.

8.2.4 Examining the Transaction Mix

We currently treat the workload as a ‘black box’ and do not take various types of transactions (or the *transaction mix*) into account when determining the maximum throughput of a partition. In a real world scenario, it is likely that the transaction mix would have a significant effect on the throughput of a partition. For example, a partition is likely to service small reads much faster than complex transactions that involve reads as well as writes. Therefore if we determine the separate throughput of each transaction type, we can use this information to make Elasca more flexible in adapting to changes in the transaction mix.

8.2.5 Workload Prediction

Elasca's optimizer currently behaves in a reactive manner. It would be interesting to extend the optimizer with predictive capabilities, so that the optimizer can react to expected changes in the workload in advance. This can be used to reduce the adverse impact of staggering scale-in on the computing resources required by predicting if scale-in is needed and doing it early. One way to implement a predictive optimizer is by monitoring the load on the database over time and learning the changes that occur periodically. These periodic changes can then be dealt with in advance.

References

[1] Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/.

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

[3] Todd Anderson, Yuri Breitbart, Henry F. Korth, and Avishai Wool. Replication, consistency, and practicality: are these mutually exclusive? In *Proc. Int. Conf. on Management of Data (SIGMOD)*, 1998.

[4] Apache ZooKeeper, 2010. http://zookeeper.apache.org/.

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

[6] 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.

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

[8] Yuri Breitbart, Raghavan Komondoor, Rajeev Rastogi, S. Seshadri, and Abraham Silberschatz. Update propagation protocols for replicated databases. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, 1999.

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

[10] 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.

[11] Parvathi Chundi, Daniel J. Rosenkrantz, and S. S. Ravi. Deferred updates and data placement in distributed databases. In *Proc. Int. Conf. on Data Engineering (ICDE)*, 1996.

[12] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proc. Symposium on Cloud Computing (SOCC)*, 2010.

[13] George P. Copeland, William Alexander, Ellen E. Boughter, and Tom W. Keller. Data placement in bubba. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, 1988.

[14] George P. Copeland, William Alexander, Ellen E. Boughter, and Tom W. Keller. Data placement in Bubba. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, 1988.

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

[16] 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 *Proc. Conf. on Innovative Data Systems Research (CIDR)*, 2011.

[17] 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.

[18] Carlo Curino, Evan P.C. Jones, Samuel Madden, and Hari Balakrishnan. Workload-aware database monitoring and consolidation. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, 2011.

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

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

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

[22] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, 1996.

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

[24] Kien A. Hua and Chiang Lee. An adaptive data placement scheme for parallel database computer systems. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 1990.

[25] 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.

[26] Bettina Kemme and Gustavo Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 2000.

[27] Sami Khuri, Thomas Bäck, and Jörg Heitkötter. The zero/one multiple knapsack problem and genetic algorithms. In *Proc. Symposium on Applied Computing*, 1994.

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

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

[30] Umar Farooq Minhas. *Scalable and Highly Available Database Systems in the Cloud.* PhD thesis, University of Waterloo, 2012.

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

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

[33] Esther Pacitti, Pascale Minet, and Eric Simon. Fast algorithms for maintaining replica consistency in lazy master replicated databases. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 1999.

[34] Andrew Pavlo, Carlo Curino, and Stanley B. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, 2012.

[35] Pedro I. Rivera-Vega, Ravi Varadarajan, and Shamkant B. Navathe. Scheduling data redistribution in distributed databases. In *Proc. Int. Conf. on Data Engineering (ICDE)*, 1990.

[36] J. Rolia, A. Andrzejak, and M. Arlitt. Automating enterprise application placement in resource utilities. In *Self-Managing Distributed Systems*, volume 2867 of *Lecture Notes in Computer Science*. Springer, 2003.

[37] Domenico Saccà and Gio Wiederhold. Database partitioning in a cluster of processors. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 1983.

[38] Database sharding whitepaper. www.dbshards.com/articles/database-sharding-whitepapers/.

[39] Database Sharding at Netlog, with MySQL and PHP. http://nl.netlog.com/go/developer/blog/blogid=3071854.

[40] Gokul Soundararajan, Cristiana Amza, and Ashvin Goel. Database replication policies for dynamic content applications. In *Proc. European Conf. on Computer Systems (EuroSys)*, 2006.

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

[42] Telecommunication Application Transaction Processing (TATP) Benchmark Description, 2009. http://tatpbenchmark.sourceforge.net/TATP_Description.pdf.

[43] The TPC-C Benchmark, 1992. http://www.tpc.org/tpcc/.

[44] Beth Trushkowsky, Peter Bodík, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. The SCADS director: Scaling a distributed storage system under stringent performance requirements. In *Proceedings of the 9th USENIX*

conference on File and stroage technologies, FAST'11, pages 12–12, Berkeley, CA, USA, 2011. USENIX Association.

[45] Patrick Valduriez. Parallel database systems: Open problems and new issues. *Distributed and Parallel Databases (DPD)*, 1(2), 1993.

[46] VoltDB. http://voltdb.com/.

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

[48] J. Wolf. The placement optimization program: a practical solution to the disk file assignment problem. In *Proc. Int. Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 1989.

APPENDICES

Appendix A

Offline Optimizer

The offline optimizer is aimed at scenarios where the workload is known in advance. For example, in the case of a periodic workload where an optimal placement of partitions is required for, say, a daily, weekly, or monthly *planning period*. This appendix presents a formulation for the offline partition placement problem that was developed as part of [30].

A.1 Notation

The offline optimizer uses the same notation as used in Elasca's optimizer (introduced in Chapter 5), with a few additions. Let T denote the number of time intervals in the planning period. Let X_j^t denote the assignment vector for nodes. A value of $X_j^t = 1$ means that the j-th node is used in the time interval t. A node will be used if there are partitions allocated to it in a given time interval t. Let S denote the cost of running a node, and I denote the cost of migrating a partition.

A.2 Objective Function

The goal of the optimizer is to find a set of allocation matrices $A = \{A^0, ..., A^T\}$ that minimize 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 formally expressed as:

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

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 partition migrations.

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 a few extra nodes. Choosing the optimal setting for the cost of a node, or migration cost is dependent on the workload and the particular server environment, and is not presented here.

A.3 Constraints

The constraints for the offline optimizer are the same as Elasca's optimizer, with minor modifications. First, the constraints specifying the number of active nodes (Equations 5.9 and 5.10) are removed, as the offline optimizer determines the number of nodes required as part of the objective function. Second, the left side of the constraint for CPU capacity (Equation 5.12) is modified to include X_{jt}:

$$\sum_{i=1}^{P} A_{ij}^t * l_i^t + +\gamma^t \leq L * C * X_j^t \text{ for } j = 1, \ldots, H \text{ and } t = 1, \ldots, T \tag{A.2}$$

A.4 Solution

Similar to the online partition placement problem formulation, the offline formulation is reduced to a linear program using the method mentioned in Section 5.4. However, the objective function for the offline formulation is more complicated than the online formulation because of the folliwing reasons:

- There are a much larger number of decision variables in the offline formulation. The solver needs to find partition-to-node assignments for T decision points (i.e., A^t where $t \in \{1, \ldots, T\}$).
- The offline formulation lets the solver determine the number of hosts required to minimize the server cost. This leads to a much larger search space.

In this thesis, we used IBM ILOG CPLEX [15] to solve the offline partition placement problem, as we did for the online problem.