Live Database Migration for Elasticity in a Multitenant Database for Cloud Platforms

Sudipto Das[‡] Shoji Nishimura[§*] Divyakant Agrawal[‡] Amr El Abbadi[‡]

[‡]Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106-5110, USA
{sudipto, agrawal, amr}@cs.ucsb.edu

[§]Service Platforms Research Laboratories
NEC Corporation
Kawasaki, Kanagawa 211-8666, Japan
s-nishimura@bk.jp.nec.com

ABSTRACT

The growing popularity of cloud computing as a platform for deploying internet scale applications has seen a large number of web applications being deployed in the cloud. These applications (or *tenants*) are typically characterized by small data footprints, different schemas, and variable load patterns. Scalable multitenant database management systems (DBMS) running on a cluster of commodity servers are thus critical for a cloud service provider to support a large number of small applications. Multitenant DBMSs often collocate multiple tenants' databases on a single server for effective resource sharing. Due to the variability in load, elastic load balancing of tenants' data is critical for performance and cost minimization. On demand migration of tenants' databases to distribute load on an elastic cluster of machines is a critical technology for elastic load balancing. Therefore, efficient *live* database migration techniques with minimal disruption and impact in service is paramount in such systems. Unfortunately, most popular DBMSs were not designed to be nimble enough for efficient migration, resulting in downtime and disruption in service when the live databases need migration. We focus on this problem of live database migration in a multitenant cloud DBMS. We evaluate different database multitenancy models in the context of migration and propose an efficient technique for live migration of a tenant's database with minimal downtime and impact on performance. We implement the proposed migration technique in a database system designed for the cloud. Our evaluation using standard OLTP benchmarks shows that our proposed technique can migrate a *live* tenant database with as low as 70 ms service disruption; an order of magnitude improvement compared to known heavy weight techniques for migrating a database.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Concurrency, Relational database, Transaction processing*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems*

General Terms

Algorithms, Design, Performance.

Keywords

Cloud computing, Elastic Data Management, Fault-tolerance, Live migration, Multitenancy, Scalability, Transactions.

*The author conducted this work as a visiting researcher at UCSB.

1. INTRODUCTION

The last couple of years have seen the widespread popularity of cloud computing platforms for deploying scalable and highly available web applications. This popularity has resulted in the emergence of a number of cloud platforms (Google AppEngine, Microsoft Azure, and Facebook for instance), and a deluge of applications developed for and deployed in the cloud. For instance, the Facebook platform has more than a million developers and more than 550K active applications [18]; other cloud application platforms such as Google AppEngine and Microsoft Azure are also growing in popularity. In addition to the traditional challenges of scalability, fault-tolerance, and high availability, database management systems (DBMS) that serve these cloud platforms face the challenge of managing small data footprints of a large number of tenants with erratic load patterns [24, 34] – a characteristic feature of these new applications. To minimize the operating cost in a system with thousands of such applications (referred to as the *tenants* in the system), effective resource sharing amongst the tenants is critical. *Multitenancy*, a technique to consolidate multiple customer applications in a single operational system, is frequently used to obviate the need for separate systems for each tenant. This consolidation has resulted in different multitenancy models at different levels of the software stack. Multitenancy in the database layer – an aspect often neglected in the design of traditional DBMSs– is the focus of this paper. Considering the application domain, we restrict our discussion to OLTP systems executing short update transactions.

Different models for database multitenancy have been proposed [24] and used [3, 11, 31, 33] for different application domains. Irrespective of the multitenancy model, the pay-per-use pricing of cloud resources necessitates effective resource sharing and elastic load balancing[1] to deal with variations in load and to minimize the operating costs of the system. Multitenant DBMSs serving thousands of tenants scale up by collocating multiple tenant databases at the same machine, and scale out by spanning to a cluster of nodes. In such an architecture, as the load on the system changes during different usage periods, migration of tenant databases is critical for elasticity.[2] Furthermore, such systems cannot tolerate downtime or disruption in service since unavailability is invariably associated with lost revenue and customer dissatisfaction [14]. Thus, to be

[1]Elasticity is the ability to adapt to varying loads by adding more resources when the load increases, or consolidating the system to lesser number of nodes as the load decreases; all in a live system without disruption in the service.

[2]Note that our use of the term migration does not refer to data migration due to schema evolution, or data migration between DBMS versions, or data migration different database systems.

effectively used for elasticity, database migration should not cause downtime or long disruption in service for the tenants whose data is being migrated; a feature referred to as *Live Migration* in the virtualization literature [7]. Traditional relational databases (RDBMS) were designed to typically deal with static over provisioned infrastructures, and focus was primarily on optimizing the performance of the system for a given infrastructure. Elasticity and hence migration of a live tenant database within a multitenant DBMS was not an important feature in the design space. Even though most *Key-Value* stores (such as Bigtable [6], PNUTS [8], Dynamo [14], etc.) support migration of data fragments for fault-tolerance or load balancing, they commonly use heavyweight techniques like stopping a part of the database and then moving it to the new node and restarting (referred to as *stop and copy*); or simple optimizations to this common technique [6, 8]. Existing DBMSs (both RDBMSs and *Key-Value* stores) are therefore not amenable to migration, and require heavyweight techniques that lead to downtime, disruption in service, and high overhead; thus making database migration not attractive.

Our focus is the problem of *efficient live migration of tenant databases in a multitenant DBMS* such that database migration can be effectively used for elastic load balancing. Database multitenancy is analogous to virtualization in the database tier. Similar to virtual machine (VM) technologies which are a critical component of cloud infrastructures, our vision is that database multitenancy is one of the critical factors for the success of cloud data platforms. Live migration in a multitenant DBMS not only allows effective elastic load balancing, but also eases administration and management, decoupling a tenant's database from the node in the cluster hosting the database. We thus propose elasticity, enabled through live database migration, as a first class feature similar to consistency, scalability, fault-tolerance, and availability. We propose a technique for live migration in an operational DBMS where logically contained portions of a DBMS representing a tenant's data are migrated from one node to another. We refer to this proposed technique as *Iterative Copy*. In addition to minimizing the disruption in service of the tenant whose database is being migrated, *Iterative Copy* results in no downtime of the overall system while providing proven safety and liveness guarantees. We implemented the proposed migration technique in ElasTraS [11, 12], a scalable and multitenant DBMS for the cloud. Our evaluation of the proposed design using YCSB [9] and TPC-C [32] shows that the migration of a live tenant database can be achieved with as low as 70 ms window of service disruption for the tenant whose data is being migrated without any disruption in the rest of the system. The effectiveness of *Iterative Copy* is evident when compared to the tens of seconds to minutes of disruption when migrating a tenant in a traditional RDBMS like MySQL, and more than 500 ms disruption for a simple *stop and copy* migration technique in ElasTraS. This further asserts the need for efficient live migration techniques to enable systems to effectively use the elasticity in cloud computing infrastructures. The major contributions of this paper are:

- We formalize the problem of live migration in a multitenant DBMS for cloud platforms and propose a technique for effective live migration. To the best of our knowledge, this is the first work that addresses live migration in the context of databases and proposes a solution.
- We prove the safety and liveness of the proposed *Iterative Copy* technique and characterize its behavior under different failure scenarios. We demonstrate the effectiveness of the proposed technique by experimental evaluation using standard OLTP benchmarks.

The rest of the paper is organized as follows: Section 2 provides background on the different database multitenancy models, formalizes the problem of live migration, and provides a survey of some known migration techniques. Section 3 explains the proposed migration technique and proves the safety and liveness guarantees. Section 4 provides details of implementing the proposed technique in a cloud DBMS. Section 5 provides experimental evaluation of the proposed migration technique. Section 6 provides a survey and background on the state of the art in data management in cloud, database virtualization, and multitenancy, and Section 7 concludes the paper. In Appendix A, we explore some future extensions of these techniques.

2. PRELIMINARIES

2.1 Multitenancy Models and Migration

Database multitenancy allows effective resource sharing for customer applications that have small but varying resource requirements [3, 24]. SaaS providers like Salesforce.com [33] are the most common use cases for multitenancy in both the application as well as the database tier. A *tenant* is a customer application instance with its own set of clients and data associated with the instance. Sharing of resources amongst tenants at different levels of abstraction and distinct isolation levels results in various multitenancy models. The three multitenancy models explored in the past [24] are: *shared machine*, *shared process*, and *shared table*. The Salesforce.com model uses shared table [33], Das et al. [11] propose a design that uses the *shared process* model, and Soror et al. [31] propose using the *shared machine* model to improve resource utilization.

In the different multitenancy models, tenants' data are stored in various forms. For shared machine, an entire VM corresponds to a tenant, while for shared table, a few rows in a table correspond to a tenant. Furthermore, the association of a tenant to a database can be more than just the data for the client, and can include metadata or even the execution state. We thus define a common logical concept of tenant *cell* as:

DEFINITION 1. *A **Tenant Cell** (or **Cell** for brevity) is a self-contained granule of application data, meta data, and state representing a tenant in the database.*

A multitenant DBMS consists of thousands of *cells*, and the actual physical interpretation of a *cell* depends on the form of multitenancy used in the DBMS. The stronger the isolation between the *cells*, the easier it is to compartmentalize a tenant's data for migration. For instance, in the *shared hardware* model, the VM level isolation between tenants allows effective resource provisioning, security, and isolation from misbehaving tenants – which is much stronger compared to the row level isolation in the *shared table* model. On the other hand, stronger isolation comes at the cost of limiting the number of tenants that can be hosted at a node. For instance, only a few VMs or independent DBMS processes can be executed efficiently at a single node. The multitenancy model therefore impacts the ability to migrate, and for a system designed for migration, the goal is to select the sweet spot between the two ends of the spectrum. Note that even though the shared table model has been popularized and used successfully by Salesforce.com [33], such a multitenancy model leads to various technical challenges. These challenges include dealing with different schemas across tenants, operational reasons of complexity involved in meeting the per tenant service level agreements, and practical reasons of security and isolation of tenant's data. Therefore, even though popular, the

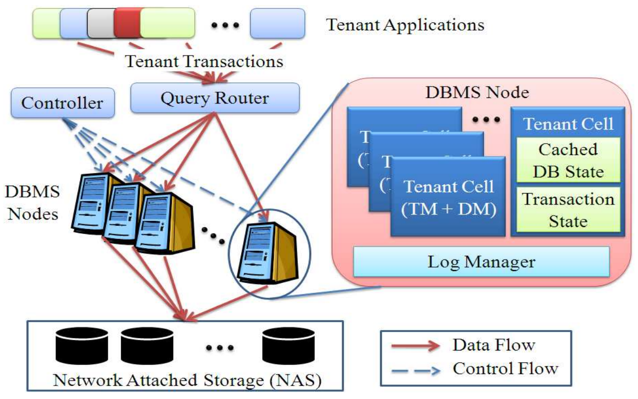


Figure 1: Reference database system model.

shared table model is not an ideal multitenancy model. With this understanding, we now present an abstract multitenant DBMS architecture that allows effective migration, while effectively sharing the resources to minimize the operating cost.

2.2 Design Rationale and Reference System Model

Our vision of database migration is based on the concept of virtualization in the database layer. The goal is to *decouple the logical representation of a* cell *from the node hosting it*. This is analogous to how the software stack of a cloud server is decoupled from the hardware executing the server by use of virtualization technologies. With this intent, we abstract the model of a DBMS for the cloud, and develop techniques which are not specific to any DBMS implementation. Figure 1 provides an overview of the proposed multitenant database system model which spans a cluster of nodes. Application clients connect through a query router which hides the distributed nature of the underlying system. A set of DBMS nodes execute the client operations. A network attached storage abstraction provides scalable, available, and fault-tolerant storage for data. A system controller is responsible for cluster control operations including initiating migration. Each tenant's *cell* is uniquely *owned* by a single DBMS node, and the system supports a relational data model and transactional guarantees on a *cell*. This multitenant DBMS model reflects *five* design rationales for designing an elastic and scalable multitenant DBMS that is amenable to migration, while being representative of typical DBMS cluster installations in cloud and enterprise infrastructures.

RATIONALE 1. ***Limit tenant transactions to single nodes.*** *In a majority of scenarios, a tenant's* cell *is small enough to be served at a single node [24, 34]. Designing the system to host a* cell *at only a single node rather than distributing it over a cluster allows transactions on a* cell *to be executed locally without the need for distributed synchronization. This is a crucial aspect of the overall design to ensure high scalability and availability [1, 22]. Thus, transaction processing on a particular* cell *is handled only by a single node in the cluster. Each such cluster node exclusively* owns[3] *a number of* cells.

RATIONALE 2. ***Decouple Ownership from Data storage.*** *Even though the DBMS nodes are independent in terms of* cell ownership, *they use a* network attached storage (NAS) *(a storage area network or a distributed file system for instance) where the actual DBMS data is stored persistently. Abstracting the storage as a separate network attached device is common in DBMS installations and allows fault-tolerance and linear scaling of the storage layer in terms of capacity and throughput. Furthermore, this decoupling allows light weight transfer of ownership from one DBMS node to another during migration, without need for large data transfers. The NAS abstraction is also common in cloud infrastructures, for instance, Elastic Block Storage (EBS) in Amazon EC2. Such decoupling is also observed in a number of other systems [13, 28].*

RATIONALE 3. ***Tenants are oblivious of the physical location of their data****. Tenants interact with their corresponding* cells *in the DBMS cluster through gateway node(s) referred to as the* query router. *The query router is responsible for routing the transactions to the appropriate DBMS node hosting the* cell. *Since the clients need not be aware of the node executing the transactions on the* cell, *this abstraction hides the dynamics within the DBMS, such as migration of* cells *and failures.*

RATIONALE 4. ***Balance resource sharing and tenant isolation.*** *Each DBMS node in the cluster runs a single DBMS instance and hosts multiple* cells. *Each* cell *has its independent transaction manager (TM) and data manager (DM), but shares the logging infrastructure for maintaining the transaction log. Since tenants are independent of each other and transactions and data do not span tenants, isolating the tenants' TMs provides better isolation and scalability, while isolating the DM eases guaranteeing the quality of service guarantees and the service level agreements (SLAs) for the tenants.*[4] *The log is shared amongst the* cells *to prevent multiple independent and competing disk writes arising from appends to the log by different* cells. *Sharing a common log allows batching of appends which also results in considerable performance improvement during normal operation. Internally, each* cell *contains a cached memory image of the tenant's database (e.g., cached pages) and the state of the active transactions (e.g., lock tables for a locking based scheduler; read and write sets for a validation based scheduler).*

RATIONALE 5. ***Balance functionality with scale.*** Key-Value *stores [6, 8, 14] are often the preferred systems for scalable and fault-tolerant data management in the cloud. Due to the requirements imposed by the application domains for which* Key-Value *stores were originally designed – for instance high availability, almost infinite scalability, geographic replication– they support considerably simpler functionality compared to RDBMSs. For instance,* Key-Value *stores support a simple data model, with no transactional support or attribute based accesses. Even though multitenant systems host thousands of tenants, each tenant's data is small. In this context, a* Key-Value *data model is not attractive from a single tenant's perspective since the developers have to trade the lack of features for scaling which is not to their direct benefit. Therefore, our system model supports a relational model and provides transactional support for individual* cells, *providing support for rich functionality and flexibility.*

2.3 Cost of Migration

In order to make migration attractive, it must be "inexpensive." Recall that in a cloud infrastructure, in addition to optimizing performance (throughput and latency of operations), the goal is also to optimize the operating cost, i.e., money spent for cloud resources. Thus, the metrics for measuring the cost of migration depend on whether the system is trying to optimize the operating cost of the

[3]Ownership refers to the exclusive rights to execute transactions on a *cell*.

[4]Some DBMSs, like MySQL, do not support independent TM and DM for different tenants, primarily because such systems were not designed for multitenancy, rather for databases where transactions can span entire data in an instance. As multitenancy becomes more prevalent, tenant level isolation is expected to become a norm.

system or performance, or a combination of the two. Irrespective of the choice of the cost metric, we identify some factors that contribute to the "cost" of migration and should be considered for evaluating a migration technique.

- **Downtime or Service Interruption during migration.** Since the primary goal of these DBMSs is to serve online web-applications, high availability is a critical requirement. Therefore, downtime and interruption in service as a result of migration should be minimized. Downtime refers to the overall system unavailability, while service interruption is unavailability or aborted transactions for the tenant whose *cell* is being migrated.
- **Migration Overhead.** To minimize the effect of migration on the normal operation of the DBMS, migration should incur low overhead on the tenant transactions. We define overhead as the additional work done and the corresponding impact on transaction execution to facilitate migration. The overhead can be further sub-divided into the following classes:
  - *Overhead before migration.* The overhead incurred during normal operation, if any, to facilitate migration.
  - *Overhead during migration.* Unless migration is instantaneous, the overhead incurred on the system while a *cell* is being migrated.
  - *Overhead after migration.* The overhead on transactions executing on the new DBMS node after migration of the *cell* has completed.

In order to allow effective and regular use of migration for elasticity and load balancing, the goal is to minimize the overhead associated with migration. We therefore define *"Live Migration"* (similar to that used in VM migration [7]) for low overhead migration as:

DEFINITION 2. ***Live Migration** in a database management system is the process of migrating a* cell *(or a logically contained part of the database) with minimal service interruption, no system downtime, and minimal overhead resulting from migration.*

Live migration in a multitenant DBMS is our focus in this paper. In Section 3, we present techniques for live migration in a multitenant DBMS for the cloud, but before that, we present some migration techniques supported by present DBMSs, and reason why such techniques are not effective.

2.4 Known Migration Techniques

Stop and copy. The simplest approach for migrating a *cell* is to stop serving the *cell* at the source DBMS node, move data to the destination DBMS node, and start serving the *cell* at the destination. This approach is referred to as *stop and copy* technique. Though extremely simple, this approach results in considerably long down times for clients of the tenant whose *cell* is being migrated. Furthermore, the entire database cache is lost when the *cell* is restarted at the destination DBMS node, thereby incurring a high post migration overhead for warming up the database cache. Thus, even though *safe* migration can be achieved using *stop and copy*, the accompanying disruption in service during migration does not make it a suitable technique for *live migration*.

On demand migration. To minimize the downtime resulting from migration, a technique outlined in [10] proposes transferring minimal information during a fast *stop and copy migration* that moves "control" of the *cell* and minimal information about the *cell*. New transactions start executing at the destination DBMS once the *cell* comes online at the destination. Since data is not moved immediately, transactions that access data which has not been migrated incur expensive "cache misses" followed by the on demand fetching of database pages from the remote source node. Even though this technique is effective in reducing service interruption, not only does it introduce a high post migration overhead resulting from page faults, but recovery in the presence of failures is also complicated and would require expensive synchronization between the source and destination servers.

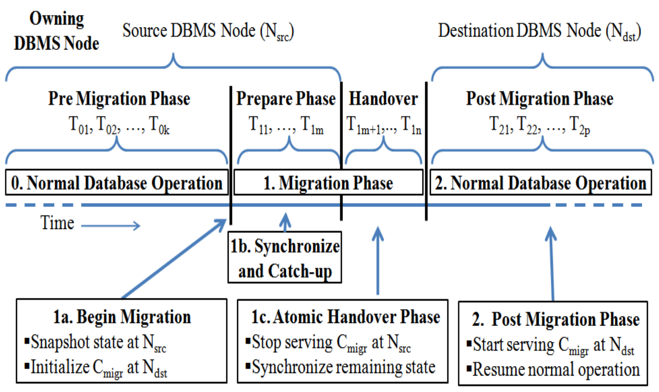

Owning DBMS Node
Source DBMS Node (N_{src})
Destination DBMS Node (N_{dst})
Pre Migration Phase
$T_{01}, T_{02}, \ldots, T_{0k}$
Prepare Phase
$T_{11}, \ldots, T_{1m}$
Handover
$T_{1m+1}, \ldots, T_{1n}$
Post Migration Phase
$T_{21}, T_{22}, \ldots, T_{2p}$
0. Normal Database Operation
1. Migration Phase
2. Normal Database Operation
Time
1b. Synchronize and Catch-up
1a. Begin Migration
▪Snapshot state at N_{src}
▪Initialize C_{migr} at N_{dst}
1c. Atomic Handover Phase
▪Stop serving C_{migr} at N_{src}
▪Synchronize remaining state
2. Post Migration Phase
▪Start serving C_{migr} at N_{dst}
▪Resume normal operation

Figure 2: Database migration timeline (times not drawn to scale).

3. LIVE DATABASE MIGRATION

We now explain our proposed *Iterative Copy* technique for the live migration of a tenant's *cell*. We describe this technique using the abstract database system model described in Figure 1. The actual implementation details inside a specific DBMS engine are explained in Section 4. In general, the problem of database migration can be sub-divided into two sub problems: (*i*) modeling system behavior to determine *when* to migrate *which cell* and *where* to migrate; and (*ii*) performing the actual migration. Sub problem (i) is beyond the scope of this paper. This paper focusses on (ii): given a *cell* and the destination DBMS node, *how* to perform efficient and safe live migration of the *cell*.

3.1 Iterative Copy Migration

In our system model, the persistent data of a *cell* is stored in shared storage and hence does not need migration. To minimize the service disruption and low post migration overhead, the *Iterative Copy* technique focusses on transferring the main memory state of the *cell* so that the *cell* restarts "warm" at the destination node. As depicted in Figure 1, the main-memory image of a *cell* consists of the cached database state (DB state), and the transaction execution state (Transaction state). For most common database engines [21], the DB state includes the cached database pages or buffer pool, or some variant of this. The transaction state, on the other hand, includes the state of the active transactions and in some cases a subset of committed transactions needed to validate the active transactions. For instance, in a locking based scheduler (2PL for instance), the transaction state consists of the lock table; while for validation based schedulers (like optimistic concurrency control [26]), this state consists of the read-write sets of active transactions and a subset of committed transactions.

Figure 2 depicts the timeline of the operations performed during the migration of a *cell*; refer to Table 1 for notational conventions. All the actions depicted in Figure 2 are at the *cell* being migrated (C_{migr}) and the DBMS nodes (N_{src} and N_{dst}) involved in the process. The overall timeline is divided into three broad phases – *Phase 0:* the *pre-migration phase* when C_{migr} is served from

Table 1: Notational Conventions.

Notation	Description
C_{migr}	The *cell* being migrated
N_{src}	Source DBMS node for C_{migr}
N_{dst}	Destination DBMS node for C_{migr}

N_{src}; *Phase 1:* the *migration phase* when C_{migr} is migrated from N_{src} to N_{dst}; and *Phase 2: post-migration phase* when C_{migr} is served from N_{dst}. We first conceptually explain the steps for migrating C_{migr}; failure handling and optimizations are explained in Section 3.2. Note that an **owner** of a *cell* is the DBMS node which has the exclusive rights to processing transactions on the *cell* and updating the persistent image of the database resident in the network attached storage (NAS). There are no replicas of a *cell* in the DBMS nodes, all replication for fault-tolerance is abstracted by the NAS.[5]

Phase 0: *Pre Migration Phase:* In this phase, the database is operating in normal mode, executing transactions as they arrive. Transactions $T_{01}, T_{02}, \ldots, T_{0k}$ are the transactions which have completed (i.e., either committed or aborted) in this phase.

Phase 1: *Migration Phase:* This phase is initiated by the *controller* notifying N_{src} and N_{dst} to start the migration. $T_{11}, T_{12}, \ldots, T_{1n}$ are the transactions that are executing in this phase. This phase can be sub divided into the following sub-phases:

Phase 1a: *Begin Migration:* Migration is initiated by a quick snapshot of the database state at N_{src}. This snapshot is then moved to N_{dst} and C_{migr} is initialized. N_{src} continues processing transactions while this migration is in progress.

Phase 1b: *Iterative Copy:* Since N_{src} continues serving C_{migr} while the snapshot is being migrated, the state of C_{migr} at N_{dst} will lag behind that at N_{src}. In this phase, N_{dst} tries to "catch-up" and synchronize the state of C_{migr} at N_{src} and N_{dst}. The state of C_{migr} is copied iteratively: in iteration i, the changes to the state of C_{migr} since the snapshot at iteration $i-1$ are copied and transferred to N_{dst}. In order to ensure that no changes are lost during migration, N_{src} must track the changes between the snapshots. Since this tracking is not needed during normal operation, this contributes to the migration overhead. $T_{11}, T_{12}, \ldots, T_{1m}$ are the transactions that have completed at N_{src} during this phase. This phase is terminated when the amount of state transferred in subsequent iterations converges.

Phase 1c: *Atomic Handover:* In this phase, the *ownership* of C_{migr} is transferred from N_{src} to N_{dst}. During this phase, N_{src} stops serving C_{migr}, copies the final un-synchronized state to N_{dst}, flushes changes from committed transactions to the persistent storage, and transfers control of C_{migr} to N_{dst} while informing the *query router* of the new location of C_{migr}. This operation should be *atomic* to deal with failures; details of the atomicity guarantees is explained in Section 3.2. Transactions $T_{1m+1}, \ldots, T_{1n}$ are the transactions that were active at the start of this handover phase. Since these transactions have not committed, the system can decide to abort them entirely, abort at N_{src} and restart at N_{dst}, or migrate them in a way that the transactions start execution at N_{src} and complete at N_{dst}. Irrespective of the choice, correctness is guaranteed. The choice, however, has an impact on the disruption in service which clients of C_{migr} observe. The successful completion of this phase makes N_{dst} the owner of C_{migr}.

Phase 2: *Post Migration Phase:* This phase marks the resumption of normal operations on C_{migr} at N_{dst}. The query router sends all future transactions ($T_{21}, T_{22}, \ldots, T_{2p}$) to N_{dst}. Transactions $T_{1m+1}, \ldots, T_{1n}$ which were in-flight when the handover started, are completed at N_{dst} depending on the decision taken in Phase 1c.

Note that the proposed technique involves a step similar to *stop and copy* in the *atomic handover* phase; the goal is to minimize the amount of state to be copied and flushed in the stop and copy phase so that the disruption in service observed by the clients is minimized. If the transactional workload on C_{migr} is not write-heavy, the amount of state that needs synchronization in Phase 1c will be small.

3.2 Failure Handling during Migration

We now discuss failure handling during the various phases of migration. We assume a reliable communication channel and hence we only consider node failures (and possibly network partitions). Furthermore, node failures do not lead to complete loss of data – either the node recovers or the data is recovered from NAS where data persists beyond DBMS node failures. We do not consider malicious node behavior.

In the presence of a failure, the first goal is to ensure safety of data; progress towards successful completion of migration is secondary. If either N_{src} or N_{dst} fails before Phase 1c, migration of C_{migr} is aborted. As a result, no logging of migration is necessary until Phase 1c. Failures in Phase 0 and 2 are handled as normal DBMS node failures. If N_{src} fails during phases 1a or 1b, the state of N_{src} is recovered and the migration progress is lost during this recovery, since there is no persistent information of migration in the commit log of N_{src}. N_{dst} eventually detects this failure and in turn aborts this migration. If N_{dst} fails, again migration is aborted. Since no log entries exist at N_{dst} indicating this migration, the recovery state of N_{dst} does not have any migration progress. Thus, in case of failure of either node, migration is aborted and the recovery of a node does not require coordination with any other node in the system.

Atomic Handover: The atomic handover phase (Phase 1c) consists of the following major steps: (*i*) ensure that changes from all completed transactions ($T_{01}, \ldots, T_{0k}, T_{11}, \ldots, T_{1m}$) at N_{src} have been flushed to stable storage; (*ii*) synchronize the remaining state of C_{migr}; (*iii*) transfer ownership of C_{migr} from N_{src} to N_{dst}; and (*iv*) notify the query router that all future transactions ($T_{21}, \ldots, T_{2p}$) must be routed to N_{dst}. Steps (i) and (ii) can be done in parallel; steps (iii) and (iv) can only be performed once the previous steps have completed. This transfer involves three participants – N_{src}, N_{dst}, and the query router– and requires that the transfer is atomic (i.e., either all or nothing). We perform this handover as a *transfer transaction* and use a protocol similar to Two Phase Commit (2PC) [20], a standard protocol with guaranteed atomicity used in distributed transaction commitment, with N_{src} as the coordinator. Even though we use a protocol similar to 2PC for atomic handover, our adaptation of the protocol does not suffer from blocking, a common criticism of 2PC. In the first phase of the handover, N_{src} initiates the process by executing steps (i) and (ii) in parallel, and soliciting a *yes* vote from the participants. Once all the sites acknowledge and vote *yes*, the transaction enters the second phase, where N_{src} relinquishes control of C_{migr} and transfers it to N_{dst}. In case, one of the participants votes *no*, the *transfer transaction* is aborted and N_{src} remains the owner of C_{migr}. This

[5]Note that having replicas of tenant databases does not help with elastic load balancing, since addition of new capacity in the system requires migration of databases to these newly added nodes which will not have database replicas.

step completes the transfer transaction at N_{src}, and after logging the outcome, N_{src} notifies the participants about the decision. If the handover was successful, N_{dst} assumes ownership of C_{migr} once it receives the notification from N_{src}. Every protocol action involves logging, similar to that used in 2PC.

3.3 Correctness Guarantees

Safety and *liveness* are two important properties which are needed for ensuring the correctness of the system in the presence of different types of failures. Safety implies that the system's state or data is not left in an inconsistent state by migration, or due to a failure during any stage of migration, while liveness ensures progress. In this section, we formally define safety and liveness of migration and prove that the guarantees are satisfied.

DEFINITION 3. ***Safe Migration.*** *A migration technique is safe if the following conditions are met even in the presence of arbitrary failures:* (*i*) Data Safety and Unique ownership: *The disk resident image of a* cell *is consistent at any instant of time. This is in turn guaranteed if at any instant of time, only a single DBMS node* owns *the* cell *being migrated; and* (*ii*) Durability: *Updates from committed transactions are either in the disk resident database image or are recoverable.*

DEFINITION 4. ***Liveness:*** *The liveness of a migration technique requires that:* (*i*) *If N_{src} and N_{dst} are not faulty and can communicate with each other for a sufficiently long duration in Phase 1, migration of C_{migr} is successfully completed; and* (*ii*) *Furthermore, C_{migr} is not orphaned (i.e., left without an owner) even in the presence of repeated failures.*

THEOREM 1. ***Atomicity of handover.*** *In spite of failures, C_{migr} is owned by exactly one of N_{src} and N_{dst}, i.e. C_{migr} has at least and at most one owner.*

PROOF. The atomicity proof is similar to that of 2PC. With no failures, the handover transaction commits, and ownership of C_{migr} is transferred. We prove that logging during normal operation combined with recovery is sufficient to ensure atomicity.
Failure in the first phase of handover: If either of N_{src} or N_{dst} fails during the prepare phase of handover, migration can be safely aborted, and N_{src} remains the owner of C_{migr}. If N_{src} fails, C_{migr} is unavailable until N_{src} recovers. If N_{dst} fails, it recovers its own state and "forgets" about the migration in progress. Failure in this phase does not need coordinated recovery.
Failure in the second phase: After receiving responses (both yes and no votes), N_{src} is ready to complete the *transfer transaction*. Once the decision about the outcome is forced into the log, the transfer transaction enters the second phase. A failure in this phase requires coordinated recovery. If N_{src} decided on a commit, N_{dst} is now the new owner of C_{migr}, otherwise N_{src} continues as the owner. If N_{src} failed before notifying N_{dst}, N_{dst} has to wait till the state of N_{src} is recovered before it can start serving C_{migr}.
At least one owner: We can see that C_{migr}, even in the presence of failures, has at most one owner. A pathological condition arises when after committing the transfer transaction at N_{src}, both N_{src} and N_{dst} fail. But the synchronized recovery of the participants as outlined above ensures that N_{dst} knows about the outcome once both nodes have recovered. □

We now articulate two important properties which allows the system to gracefully tolerate failures.

THEOREM 2. ***Independent Recovery.*** *Except during the execution of the atomic handover protocol, recovery from a failure of N_{src} or N_{dst} can be performed independently.*

PROOF. The ability to safely abort migration at an incomplete state and the single owner philosophy ensures that a failure during migration would allow independent recovery of the failed node's state, without synchronizing state with any other node. At any point of time before the commit phase, N_{src} is the owner of C_{migr}. If N_{src} fails, it recovers without any interaction with N_{dst}, still being the owner of C_{migr}. Similarly, if N_{dst} fail, it recovers its state. Unless the handover phase was initiated (Phase 1c), N_{dst} has no log record about the migration in progress, so it "forgets" the migration and continues normal operation. Similarly, once handover has been successfully completed, N_{dst} becomes the new owner of C_{migr}. A failure of N_{src} at this instant can be recovered independently as N_{src} does not need to recover state of C_{migr}. Similarly, a failure of N_{dst} requires recovery of only its state, and it can also independently recover state of C_{migr}, since it had successfully acquired the ownership of C_{migr}. □

THEOREM 3. ***A single failure does not incur additional unavailability.*** *Any unavailability of C_{migr} resulting from a failure of one of N_{src} or N_{dst} during migration is equivalent to unavailability due to failure during normal operation.*

PROOF. From an external observer's perspective, N_{src} is the owner of C_{migr} until the atomic handover phase (Phase 1c) has successfully completed. Any failure of N_{dst} before Phase 1c does not affect the availability of C_{migr}. A failure of N_{src} during this phase makes N_{src} unavailable, but this is equivalent to a failure of N_{src} under normal operation where C_{migr} would also become unavailable. Similarly, after migration is complete (Phase 2), N_{dst} becomes the owner of C_{migr}. Any failure of N_{src} does not affect C_{migr}, and a failure of N_{dst} which makes C_{migr} unavailable is equivalent to the failure of N_{dst} during normal operation. The only complexity arises in the case of a failure in Phase 1c when a coordinated recovery is needed. If N_{src} fails before successful completion of Phase 1c, even if N_{src} had locally relinquished ownership of C_{migr}, if the transaction did not complete, N_{dst} cannot start serving C_{migr} in which case it becomes unavailable. This is similar to the blocking behavior in 2PC [20]. But since the handover transaction did not complete, from an observer's perspective, N_{src} was still the owner of C_{migr}, and hence this unavailability is equivalent to the failure of N_{src} during normal operation. Thus, it is evident, single site failures during migration does not impact availability of C_{migr}. □

The properties of independent recovery and graceful handling of failures during migration are crucial for effective use of migration for elasticity. Furthermore, one of the implications of Theorem 3 is that in spite of using a protocol similar to 2PC, the handover phase does not block any system resource in the presence of a failure. The following corollary follows directly from Theorem 1 and guarantees unique ownership.

COROLLARY 4. *Provided C_{migr} had a unique owner (N_{src}) before migration, C_{migr} continues to have a unique owner (N_{dst} if migration was successful, and N_{src} if it failed) during as well as after the migration phase.*

LEMMA 5. *Changes made by aborted transactions are neither persistently stored or copied over during migration.*

PROOF. This proof follows from the assertion that in the steady state, the combination of the database buffer and the persistent disk image do not have changes from aborted transactions. In locking based schedulers, the buffer or the disk images might have changes from uncommitted transactions, but changes from aborted transactions are undone as part of processing the abort. For validation based schedulers like OCC, changes from aborted transactions are never made visible to other transactions from the system. The goal of the iterative copy phase is to replicate this buffer state at N_{dst} and hence, effects of aborted transactions which might have been copied over, are guaranteed to be undone by the last stop-and-copy phase, at which point buffers at both N_{src} and N_{dst} are synchronized. □

LEMMA 6. *Changes made by committed transactions are persistently stored and the log entries of completed transactions on C_{migr} at N_{src} can be discarded after successful migration.*

PROOF. Changes from a committed transactions are recoverable by definition due to the forcing of log entries before a commit. One of the steps in the handover protocol enures that changes from completed transactions are persisted to disk as part of the handover. □

LEMMA 7. *Migration of active transactions during migration does not violate the durability condition even if the write ahead log at N_{src} is discarded after successful migration.*

PROOF. In progress transactions ($T_{1m+1}, \dots, T_{1n}$) transferred from N_{src} to N_{dst} might have some entries in the commit log at N_{src}. Ensuring that those entries are re-inserted into the log at N_{dst} during commit of the transactions $T_{1m+1}, \dots, T_{1n}$ ensures that they are recoverable, and hence does not violate the durability condition even if the log is discarded at N_{src}. □

Corollary 4 guarantees the *data safety* condition and Lemmas 5, 6, and 7 together guarantee the *durability* condition, thus proving the safety of the proposed migration technique. As noted earlier, in the presence of a failure of either N_{src} or N_{dst}, the migration process is aborted without jeopardizing the safety of the system.

LEMMA 8. ***Progress guarantee.*** *Migration succeeds if N_{src} and N_{dst} can communicate during the entire duration of Phase 1 of migration.*

3.4 Minimizing service disruption

C_{migr} is unavailable during Phase 1c when ownership is transferred from N_{src} to N_{dst}. The goal of Phase 1b is to minimize this service disruption by minimizing the amount of state that needs copying in the final step. Another optimization to minimizing disruption is how transactions $T_{1m+1}, \dots, T_{1n}$, which were active at the start of the atomic handover process, are handled – i.e., whether they are aborted and restarted, or carried over to complete execution at N_{dst}.

The easiest choice is to abort transactions $T_{1m+1}, \dots, T_{1n}$ at N_{src} and notify the clients who might restart them at a later time. In this case, no active transaction state needs transfer from N_{src} to N_{dst}. N_{src} only has to ensure that the effects of the aborted transactions have been undone. But this behavior might not be acceptable for many applications that cannot tolerate such heavy disruptions. We now present two techniques to minimize the disruption in service: one to deal with transactions whose logic is being executed at the client, another technique for dealing with stored procedures. In both the techniques, the clients do not notice transactions aborted by the system, rather some transactions having a higher than average latency.

In the case where the transaction logic is executed at the client (or any node other than N_{src}) and the DBMS node is serving only the read/write requests, the transactions can be seamlessly transferred from N_{src} to N_{dst} without any loss of work. The stop-and-copy phase copies the active transaction state along with the database state. For a 2PL scheduler, this amounts to copying the state of

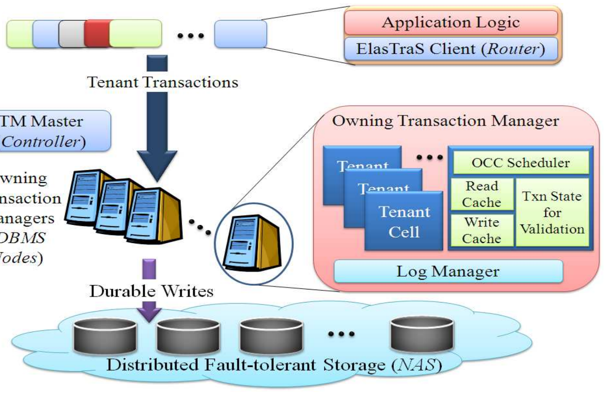


Figure 3: System architecture of ElasTraS and its mapping to the components of the reference architecture in Figure 1.

the lock table, and for an OCC scheduler, this amounts to copying the read/write sets of the active transactions and the subset of committed transactions whose state is needed for validation. For a 2PL scheduler, updates of active transactions are done in place in the buffer pool, and hence are copied over during the final copy phase. While for OCC, the local writes of the active transactions must be copied to N_{dst}. Note that irrespective of the concurrency control technique used, to ensure the recoverability of transactions $T_{1m+1}, \dots, T_{1n}$ which complete at N_{dst}, the part of the commit log for these transactions must also be transferred from N_{src} to N_{dst}. Furthermore, since these transactions are still active when they resume execution at N_{dst}, the log entries of these transactions need to be persistently stored only during commit and not during the handover.

For efficiency and to minimize the number of network round-trips needed by a transaction, many OLTP systems support transactions whose logic is represented by stored procedures which are executed locally in the DBMS node, with the clients initiating the transactions by passing the necessary parameters. These transactions execute as threads within the DBMS engines or processes local to the node. Since we do not perform any process level transfer during migration, it is not straightforward to migrate such live executing transactions. Therefore, we have to abort these transactions at N_{src}. But since the client only passes the parameters to the transactions, these transactions can be restarted at N_{dst} without the client's intervention, and the parameters passed by the client must be migrated to N_{dst}. In such a scenario, these transactions are similar to new transactions $T_{21}, \dots, T_{2p}$ arriving directly to N_{dst}, the only difference being that they are carried over transactions and are restarted as part of starting C_{migr} at N_{dst}. For N_{src}, these transactions are like aborted transactions whose effects must be undone.

4. IMPLEMENTATION DETAILS

4.1 System Architecture

We implemented the proposed migration technique in ElasTraS [11], a database system designed for supporting multitenant cloud applications. Figure 3 provides a high level illustration of the architecture of ElasTraS. ElasTraS has been designed to provide scalable, elastic, and fault-tolerant data management functionality for a multitenant environment, and adheres to the design principles outlined in Section 2.2. At the bottom of the stack is a distributed fault-tolerant storage layer, equivalent to the NAS abstraction, which stores the persistent image of the database and the write ahead logs. ElasTraS uses the Hadoop distributed file system (HDFS) for providing a fault-tolerant file system interface. At the heart of the system are the *Owning Transaction Managers (OTM)*, equivalent to the DBMS nodes, which *own* one or more *cells* and provide transactional guarantees on them. Each OTM owns a number of *cells* and is responsible for executing the transactions for those *cells*. The *TM Master* acts as the controller of the system and monitors the correct operation, and in our implementation, initiates migration. The TM master only performs control operations and is not in the data path. ElasTraS provides a *client library* that abstracts the logic of connecting to the appropriate OTM. The client library uses a collection of metadata tables that stores the mappings of a *cell* to the OTM which is currently serving the *cell*. Thus, the combination of the client library and the metadata tables together constitute the query router. For fault-tolerance and loose synchronization between the different components in the system, ElasTraS uses Zookeeper [23], a replicated and fault-tolerant leasing and notification service. Each of the components in the system obtains leases from the Zookeeper installation. The owning of a lease guarantees mutual exclusion that allows the TM master and the OTMs to perform their operations independently without requiring any distributed synchronization. More details about the design of ElasTraS can be found in [11].

ElasTraS operates in the granularity of database partitions. Each partition encapsulates a *cell* which has its independent transaction manager (TM), data manager (TM), and database buffer. All *cells* at a single OTM share a common commit log. ElasTraS uses optimistic concurrency control [26] for providing serializable transaction execution. In OCC, updates made by an active transaction are initially kept local and are applied to the database only if the transaction commits. One important difference in the implementation of ElasTraS when compared to traditional RDBMSs is that it uses a append only log structured representation of data. Since there are no in-place updates to database pages, the database buffer is split between a read cache and a write cache, and read requests are answered by a merged view of the two caches. All incoming write requests are cached in the write buffer, which is periodically flushed to the persistent storage. Periodic *compactions* are used for garbage collecting the entries that are obsolete due to more recent updates or deletes, and for reducing the number of files [6,11]. Since constructing the merged view is expensive, the transfer of these two caches during migration is important to ensure the low overhead of migration.

4.2 Implementation Design

In ElasTraS, the TM master initiates the migration of a *cell* (C_{migr}), and its role in the migration is only limited to notifying the source OTM (N_{src}) and the destination OTM (N_{dst}) to initiate migration. N_{src} and N_{dst} coordinate to perform the migration and can complete the migration without any further intervention of the master. Recall that due to the NAS abstraction, migration does not need the transfer of the persistent image of C_{migr} from N_{src} to N_{dst}. The goal of migration is to transfer the cached database state from N_{src} to N_{dst} to minimize the overhead of migration for operations post migration. We now explain the implementation details of the different phases.

Copying the database cache. As noted earlier, the database cache in ElasTraS is split into two components: the read cache which caches the contents of the log structured files, and the write cache which buffers the new writes till they are flushed to stable storage. Recall that in the handover phase, any changes made by committed transactions are flushed to the stable storage. Thus, the write cache is not copied during migration. The iterative copy phase thus only copies the read cache from N_{src} to N_{dst}. During migration, the data manager of C_{migr} tracks changes to the read cache, and successive iterations only copy the changes to the cache since the previous iteration. The write cache is flushed in the handover phase, and is copied over into the read cache of N_{dst}. Thus, N_{dst} starts serving C_{migr} with an empty write cache, but the combination of the read and write cache contains the same state of data. Since the data manager hides this separation of caches and provides the transaction manager an abstraction of the merged view, both the active transactions $T_{1m+1}, \ldots, T_{1}n$ and the new transactions $T_{21}, \ldots, T_{2p}$ for C_{migr} can be begin execution at N_{dst} unaware of the migration.

Copying the transaction state. ElasTraS uses OCC [26] for concurrency control. In OCC, the transaction state comprises of the read and write sets of the active transactions as well as the subset of committed transactions which are needed for validating the new transactions. Similar to the database cache, the transaction state is also viewed as two subsets. The iterative copy phase copies over only the state of the committed transactions, while the state of the in-flight transactions is copied in the final handover phase. Recall that in OCC, writes from an active transaction are maintained locally with the transaction. Therefore, while copying the state of the active transactions, these local transaction writes are also copied over to N_{dst}. Additionally, the TM maintains counters which are used to assign identifiers for new and committed transactions. The value of these counters are also copied over during the final handover phase.

Handover phase. The handover phase starts with a flush of any changes from committed transactions to the persistent storage. It then copies over the state of the active transactions, the write cache, and the changes to the read cache since the previous copy. In ElasTraS, the query router is a combination of the system metadata and the client library that uses this metadata to route the transactions to the appropriate OTM. The client library maintains a cache of the pertinent system metadata state. Therefore, the *transfer transaction* must only update the metadata to reflect the new *owner* of C_{migr}. ElasTraS manages the metadata as a partition served by an OTM, and the OTM serving the metadata partition is also a participant in the *transfer transaction*. Once migration is complete, the client cache will be invalidated by a failed attempt to access C_{migr} at N_{src}, at which time the updated meta information with the new destination of C_{migr} is read and cached by the client. Various optimizations are possible for improving client behavior and redirecting client requests for C_{migr} to N_{dst}. In our implementation, clients who have open connections with C_{migr} at N_{src} are directly notified about the migration and the address of N_{dst}. This prevents the clients from making an additional network round-trip to obtain the location from the OTM serving the metadata.

5. EXPERIMENTAL EVALUATION

We now experimentally evaluate our implementation of the proposed migration technique. We implemented the *iterative copy* migration technique as well as the simple *stop and copy* migration technique in the ElasTraS multitenant database system. The experiments were performed on a cluster of nodes in Amazon Elastic Compute Cloud (EC2). We use a ten node cluster for our experiments, where each node is a "Standard Extra Large Instance" (`m1.xlarge`) with 15 GB of memory, 8 EC2 Compute Units (4 virtual cores with 2 EC2 Compute Units each), and 1690 GB of local instance storage. The distributed fault-tolerant storage (Hadoop distributed file system in our case) and the OTMs were co-located in the cluster of ten worker nodes. The TM master (controller) and the clients generating the workloads were executed on a separate set of nodes. ElasTraS also uses a Zookeeper cluster [23] for loose distributed synchronization, and a separate three node zookeeper ensemble is used. These experiments measure the cost of migration in terms of disruption in service: number of tenant operations failing, and the time window for which C_{migr} is unavailable for serving requests. Since during a specific instance of migration, only the source and destination of migration are involved, the scale of nodes in the system is irrelevant for this evaluation. We select a system of ten nodes as a representative size; refer to [11] for scalability experiments on ElasTraS. We use the *stop and copy migration* technique as a baseline for comparison. Since in ElasTraS, the data corresponding to a *cell* is stored in the NAS, stop and copy migration does not involve any movement of the persistent image of the database, thus providing a fair baseline for comparison. In [16], we evaluated various other off-the-shelf migration techniques on popular open-source RDBMSs like MySQL. These techniques resulted in tens of seconds to minutes of disruption in service for the database being migrated [16]. We do not include these numbers in this paper since they include the cost of migrating the tenant's data and are using a system not originally designed for such scenarios.

5.1 Workload Specification

The cost incurred due to migration depends on a number of factors: the type of workload on the system, the size of the database cache, and the data access patterns for the transactions. To evaluate this spectrum of factors affecting the cost, we use two different OLTP benchmarks which have been appropriately adapted for transactional workloads on a multitenant relational database: (*i*) the Yahoo! cloud serving benchmark (YCSB) [9] adapted for transactional workloads to evaluate performance of the proposed technique under different read/write loads and access patterns; and (*ii*) the TPC-C benchmark [32] representing a complex transactional workload for typical relational databases. We focus on two cost measures: the length of time for which a *cell* becomes unavailable during migration, and the number of operations that fail while a *cell* is being migrated.

5.1.1 Yahoo! Cloud Serving Benchmark

YCSB [9] is a recently proposed benchmark for evaluating systems that drive web applications; the authors refer to such systems as "serving systems." The goal of this benchmark is complementary to that of traditional OLTP benchmarks like TPC-C, and is designed to provide a comprehensive evaluation of the performance space by varying the type of workloads (read-write percentage) and the access patterns (uniform, zipfian, and latest). We extend the benchmark by adding different transactional workloads and adapting it to a multitenant system where the system is viewed as a collection of large number of small databases, instead of a single large database.

5.1.2 TPC-C Benchmark

The TPC-C benchmark is an industry standard benchmark for evaluating the performance of OLTP systems [32]. Our goal of using this benchmark in the evaluation is to determine the overhead of migration on a complex transactional workload. The benchmark suite consists of nine tables and five transactions that portray a wholesale supplier. The TPC-C transaction mix represents a good mix of read/write transactions. The system comprises of a number of warehouses which in turn determine the scale of the system. More details about the benchmark can be found in [32]. For a multitenant system, we configure the system such that a number of warehouses comprise a tenant; the number determines the size of the tenants. The benchmark is adapted to ensure that transactions are always limited to a single tenant.

5.2 Evaluating Cost of Migration

5.2.1 Yahoo! Cloud Serving Benchmark

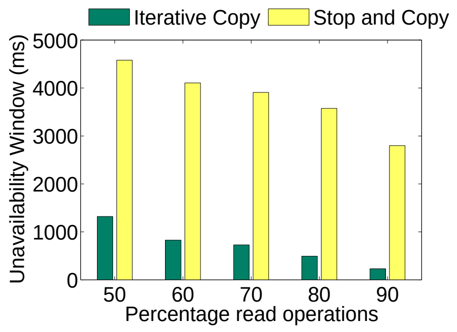
We first evaluate the cost of migration using the YCSB benchmark that allows us to vary different parameters that cover a wide spectrum of workloads. The parameters varied in the experiments include the percentage of reads in a transaction (default is 80%), the number of operations in the transaction (default is 10), the size of the tenant database (default is 200 MB), and the distribution of the access patterns (default is uniform). In each of the experiments, we vary one of these parameters while using the default values of the rest of the parameters. In addition, the load on each *cell* is set to a constant of about 2000 transactions per minute in all the experiments. We evaluate the impact of load in our evaluation using the TPC-C benchmark. Figures 4 and 5 plot the results from the various experiments using YCSB; each sub-figure corresponds to the different experiments which we explain later in this section. Figure 4 plots the unavailability window (in ms) for the *cell*, while Figure 5 plots the number of failed operations for the *cell* being migrated. The x-axes of the figures corresponds to the parameter being varied. In all the figures, the two bars correspond to the two migration techniques being evaluated: the proposed *iterative copy* technique and the hitherto used *stop and copy* technique implemented in ElasTraS.

In the first experiment (Figures 4(a) and 5(a)), we vary the percentage of read operations in a transaction from 50% to 90%. The goal of this experiment is to evaluate the impact of update load on the cost of migration. As evident from Figures 4(a) and 5(a), the cost of migration is higher when the percentage of updates in the workload is higher. The increased cost of migration is because a higher update rate results in larger amount of data which need to be synchronized as well as flushed during migration. This results in a longer unavailability window which translates to a larger number of operations failing which the *cell* becomes unavailable.

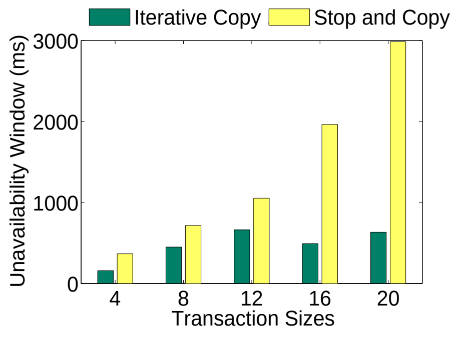
In the next experiment (Figures 4(b) and 5(b)), we vary the number of operations in a transaction from 4 to 20. The goal of this experiment is to evaluate the impact of the duration of transactions on the disruption observed by the tenants. Since the applied load on each *cell* is kept a constant irrespective of the size of the transaction, the larger the number of operations in a transaction, the larger is the percentage of update operations. We therefore see a trend similar to that observed in the previous experiment where the cost of migration increases as the update load on the *cell* increases.

In the third experiment (Figures 4(c) and 5(c)), we vary the size of a *cell* from 100 MB to 500 MB. Even though none of the migration techniques being evaluated involve migration of the persistent image of the *cell*, the size of the *cell* has an impact on the size of the cache which is flushed by both the techniques as well as migrated in the iterative copy technique. With uniform access patterns, the size of the cache is a fraction of the size of the *cell*, and therefore, a larger *cell* size implies greater amount of data being flushed as well as synchronized. This increase in the cost of migration with an increase in the size of the *cell* is also observed in Figures 4(c) and 5(c).

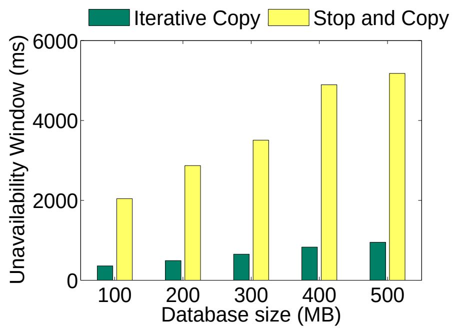
In the final experiment using YCSB (Figures 4(d) and 5(d)), we vary the distributions driving the access patterns for the transactions. The access pattern determines the size of the hot data as well the amount of state that must be flushed and synchronized during



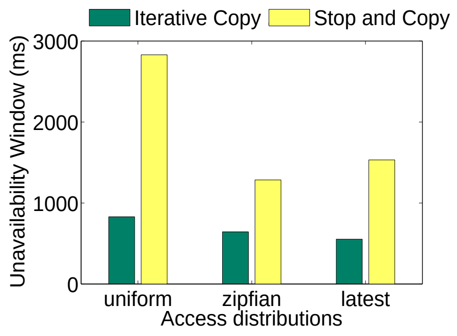
(a) Varying read percentage.



(b) Varying transaction sizes.



(c) Varying the size of the database.



(d) Varying the access pattern.

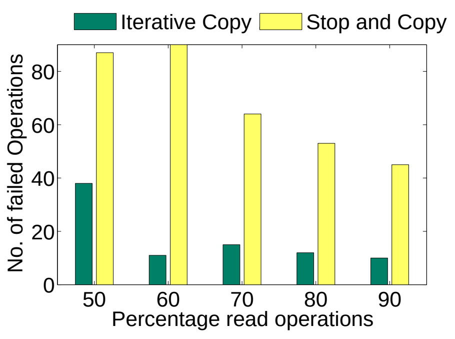
Figure 4: Time duration for which a Tenant *cell* is unavailable during migration. Evaluation using YCSB.

migration. We use three different access patterns in this experiment: *uniform*, *zipfian*, and *latest*. Refer to [9] for a discussion on these access patterns. With both latest as well as zipfian, only a small subset of a tenant's data is hot, while a uniform distribution does not result in hot spots. Therefore, the amount of state that changes as a result of uniform access distribution is expected to be greater compared to that of zipfian or latest. This trend is also evident in the cost of migration shown in Figures 4(d) and 5(d).

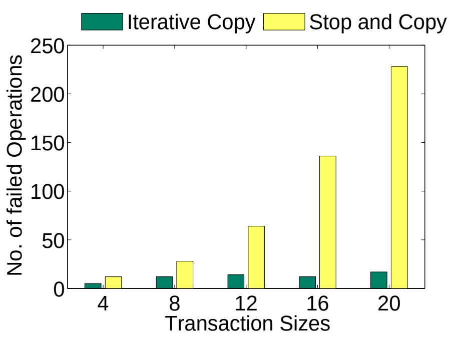
In summary, for all the workload types, the proposed iterative copy migration technique outperforms the stop and copy migration technique; both in terms of service disruption as well as the number of failed operations. For a lightly loaded *cell*, we observe an unavailability window as low as 70 ms using the proposed iterative copy technique, compared to about 500 ms unavailability window in stop and copy. Even for heaver workloads, the unavailability window for iterative copy is between 3 to 10 times smaller compared to stop and copy. The ElasTraS client retries (or query router) the operations that failed during the unavailability window. With the unavailability window less than a second in all but one of the experiments, using the iterative copy technique allows the system to migrate the *cell* within the retry interval (which is typically set to one second). This successful retry reduces the number of failed operations which the application clients observe. Using stop and copy, the longer unavailability window results in the retry operations failing as well. This explains the very small number of failed operations in *iterative copy* compared to that of *stop and copy*. Furthermore, in iterative copy, transactions active during migration start at N_{src} and commit at N_{dst}. Therefore, there are no transaction aborts. Whereas in stop and copy, all transactions active during migration are aborted which further contributes to the number of failed oeprations. We are currently working on optimizing our implementation to further minimize the cost of migration. The amount of data to be synchronized and flushed in the final handover step has a big impact on performance; and our future work is focussed on further minimizing the amount of data transferred in the handover phase of iterative copy.

5.2.2 TPC-C Benchmark

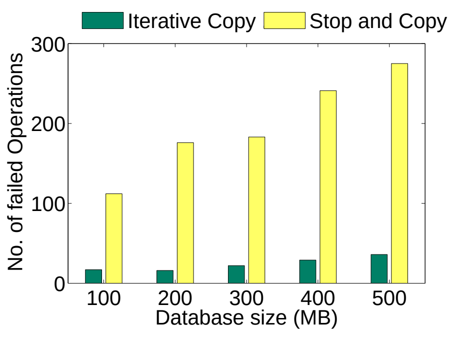
We now evaluate the *iterative copy* technique using the industry standard TPC-C benchmark [32], adapted for a multitenant setting. The goal of this evaluation using the TPC-C benchmark is to evaluate the performance of iterative copy with complex transaction workloads representing real-life business logic and larger tenant databases. We made two modifications to the TPC-C benchmark: *first*, instead of having a single large TPC-C database, we set up the system as a collection of small to medium TPC-C databases comprised of a number of warehouses; and *second*, we limit all transactions to a single tenant *cell*. Figure 6 plots the results from the experiments using the TPC-C benchmark where we varied the load on each of the tenant *cells*. The y-axes plot the cost of migration measures, while the x-axes plots the load on the system. In these experiments, each tenant database size was about 1 GB and contained four TPC-C warehouses. We vary the load on each tenant from 500 tpmC (transactions per minute TPC-C) to 2500 tpmC. As the load on each *cell* increases, the amount of state that needs to be synchronized and copied over also increases, since the TPC-C benchmarks has a high percentage of update transactions (more than 90% of the transactions have a minimum of one update). As a result, the length of the unavailability window increases with an increase in the load of the system (see Figure 6(a)). Furthermore, since at higher load, the rate of arrival of operations is larger, an increase in the unavailability window has a greater impact at higher loads, as more operations fail during the period when the *cell* is being migrated. This increase can be observed in Figure 6(b) where the number of failed operations increases with an increase in the the load on the system. Note that even at such high loads, migration



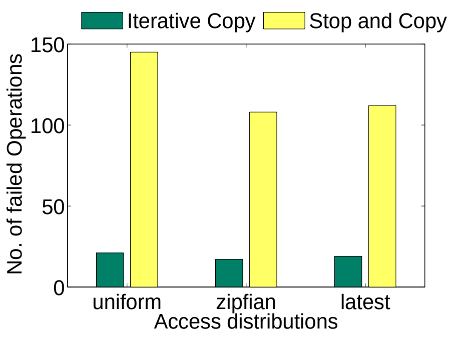
(a) Varying read percentage.



(b) Varying transaction sizes.



(c) Varying the size of the database.



(d) Varying the access pattern.

Figure 5: Number of operation and transaction failures during migration. Evaluation using YCSB.

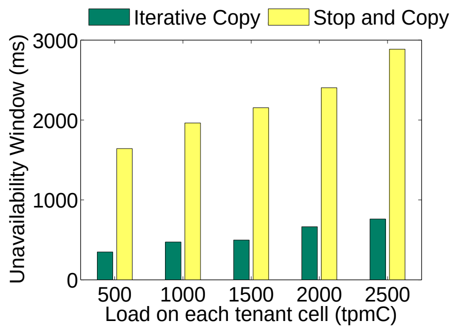
can be performed with a low overhead and without any disruption in service of other tenants. Furthermore, the performance of the iterative copy technique is considerably better than that of the stop and copy technique; up to 3–5 times in terms of unavailability window, and up to 10 times in terms of failed operations. As seen in the experiments with YCSB, the longer unavailability window in stop and copy results in a considerable higher number of failed operations. From the experiments, it is evident that migrating a *cell* with a smaller load causes less disruption compared to a heavily loaded *cell*. Therefore, when a DBMS node becomes heavily loaded, it is prudent to migrate the less loaded *cells* to other lightly loaded nodes. Migrating the less loaded *cells* helps ensure the SLA's of these *cells* without incurring high overhead due to migration.

6. RELATED WORK

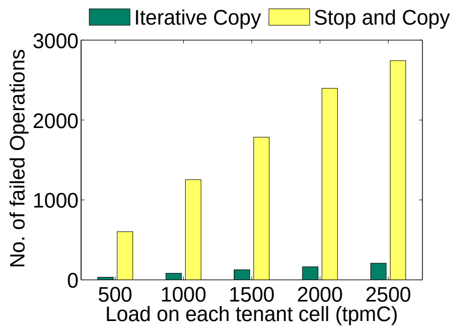
A lot of research has focussed on cloud DBMSs, DMBSs in a virtualized environment, and multitenancy data models. However, to the best of our knowledge, this is the first work that focuses on live database migration for elastic load balancing. Multitenancy in the database tier is common in various systems that support Software as a Service paradigms such as Salesforce.com [33]. Different forms of multitenancy have been proposed in the literature [3, 24, 33], the predominant forms being: *shared table* [33], shared process [11], and shared machine [31, 34]. Jacobs et al. [24] provide a summary of these different multitenancy models and the different tradeoff associated with each such form. Elmore et al. [16] extend these multitenancy models and propose a finer sub-division specific to the different cloud computing models. Aulbach et al. [3] and Weissman et al. [33] explain the design of two large multitenant systems built using the shared table model. They also explain the different optimizations and structures used to efficiently processing tenant transactions in the shared table model. Soror et al. [31] analyze the shared machine model where they leverage virtualization technologies to collocate multiple tenant databases on the same machine, and evaluate the different implications of such a design.

Even though very little work has focused on live migration of databases, a number of techniques have been proposed in the virtualization literature to deal with the problem of live migration of virtual machines (VM) [7, 27]. VM migration is a standard feature supported by most VMs, and is a primitive used for load balancing in large virtualized clusters, specifically the cloud infrastructures. Clark et al. [7] propose a migration technique similar to the iterative copy technique proposed in this paper; the major difference being that Clark et al. use VM level memory page copying and low level techniques for transferring network connects and live processes, while we use database level cache copying and rely on the query router to route transactions. Liu et al. [27] propose an improvement to minimize the downtime and the amount of database synchronized by using a log based copying approach. Note that in the database literature, migration has been studied in a different context – migrating data as the database schema evolves, or between different versions of the database system, or from one system to another. Different approaches have been proposed to perform such a migration efficiently in an online system. Sockut et al. [30] provide a detailed survey of the different approaches to this online reorganization of databases. We study the migration problem in a different context where migration is used for elasticity, and does not involve schema evolution or version upgrades.

A vast body of work also exists in scalable and distributed database management. Early work dates back two to three decades [15, 29]; and the evolution of these scalable data management systems has continued, with different alternative designs such as *Key-Value* stores [6, 8, 14] as well as other database systems for cloud computing environments [2, 5, 10, 11, 13, 25, 28]. Even though a large number of such systems exist, the focus of the majority of such



(a) Unavailability window.



(b) No. of failed operations.

Figure 6: Evaluating the cost of migration using the TPC-C benchmark.

systems is to scale single large databases to the cloud and the focus is on performance optimization. Kraska et al. [25] propose the use of varying consistency models to minimize the operating cost in a cloud DBMS. Our work is different from all these approaches since we propose the use of live database migration as a primitive for elastic load balancing, and rely on the peaks and troughs in the workload to minimize the operating cost of the system.

7. CONCLUSION

With the growing number of applications being deployed in different cloud platforms, the need for a scalable, fault-tolerant, and elastic multitenant DBMS will also increase. In such a large multitenant system, the ability to seamlessly migrate a tenants' database is an important feature that allows effective load balancing and elasticity for minimizing the operating cost and the efficient sharing of the system resources amongst the tenants. We presented the *Independent Copy* technique for live database migration that introduces minimal performance overhead and minimal disruption in service only for the tenant whose database is being migrated. This live migration technique decouples a *cell* from the DBMS node *owning* it, and allows the system to regularly use migration as a tool for elastic load balancing. Our evaluation using standard OLTP benchmarks shows that our proposed technique can migrate a *live* tenant database with as low as 70 ms service disruption which is orders of magnitude better compared to simple heavy weight techniques for migration of a database. In the future, we plan to extend the design by adding an intelligent system control that can model the cost of migration to predict its cost as well the behavior of the entire system. This model can be used to autonomously determine which *cells* to migrate, when to migrate, and to allow effective resource utilization in the system while maintaining the SLA's for the tenants of the system.

Acknowledgements

This work is partially supported by NSF Grant IIS-0847925 and an AWS in Education grant.

8. REFERENCES

[1] D. Agrawal, A. El Abbadi, S. Antony, and S. Das. Data Management Challenges in Cloud Computing Infrastructures. In *DNIS*, pages 1–10, 2010. 3

[2] M. Armbrust, A. Fox, D. A. Patterson, N. Lanham, B. Trushkowsky, J. Trutna, and H. Oh. SCADS: Scale Independent Storage for Social Computing Applications. In *CIDR Perspectives*, 2009. 11

[3] S. Aulbach, T. Grust, D. Jacobs, A. Kemper, and J. Rittinger. Multi-tenant databases for software as a service: schema-mapping techniques. In *SIGMOD*, pages 1195–1206, 2008. 1, 2, 11

[4] F. R. Bach and M. I. Jordan. Kernel independent component analysis. *J. Mach. Learn. Res.*, 3:1–48, 2003. 14

[5] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. Building a database on S3. In *SIGMOD*, pages 251–264, 2008. 11

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*, pages 205–218, 2006. 2, 3, 8, 11

[7] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *NSDI*, pages 273–286, 2005. 2, 4, 11, 14

[8] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, 2008. 2, 3, 11

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *ACM SoCC*, pages 143–154, 2010. 2, 9, 10

[10] C. Curino, E. Jones, Y. Zhang, E. Wu, and S. Madden. Relational Cloud: The Case for a Database Service. Technical Report 2010-14, CSAIL, MIT, 2010. 4, 11

[11] S. Das, S. Agarwal, D. Agrawal, and A. El Abbadi. ElasTraS: An Elastic, Scalable, and Self Managing Transactional Database for the Cloud. Technical Report 2010-04, CS, UCSB, 2010. http://www.cs.ucsb.edu/research/tech_reports/. 1, 2, 7, 8, 9, 11

[12] S. Das, D. Agrawal, and A. El Abbadi. ElasTraS: An Elastic Transactional Data Store in the Cloud. In *USENIX HotCloud*, 2009. 2

[13] S. Das, D. Agrawal, and A. El Abbadi. G-Store: A Scalable Data Store for Transactional Multi key Access in the Cloud. In *ACM SoCC*, pages 163–174, 2010. 3, 11

[14] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, pages 205–220, 2007. 1, 2, 3, 11

[15] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen. The Gamma Database Machine Project. *IEEE Trans. on Knowl. and Data Eng.*, 2(1):44–62, 1990. 11

[16] A. Elmore, S. Das, D. Agrawal, and A. El Abbadi. Who's Driving this Cloud? Towards Efficient Migration for Elastic and Autonomic Multitenant Databases. Technical Report 2010-05, CS, UCSB, 2010. `http://www.cs.ucsb.edu/research/tech_reports/`. 9, 11

[17] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976. 14

[18] Facebook Statistics. `http://www.facebook.com/press/info.php?statistics`, Retreived July 23, 2010. 1

[19] A. Ganapathi, H. A. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. I. Jordan, and D. A. Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *ICDE*, pages 592–603, 2009. 14

[20] J. Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, 1978. Springer-Verlag. 5, 6

[21] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1992. 4

[22] P. Helland. Life beyond Distributed Transactions: An Apostate's Opinion. In *CIDR*, pages 132–141, 2007. 3

[23] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX Annual Technical Conference*, 2010. 8, 9

[24] D. Jacobs and S. Aulbach. Ruminations on multi-tenant databases. In *BTW*, pages 514–521, 2007. 1, 2, 3, 11

[25] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency Rationing in the Cloud: Pay only when it matters. *PVLDB*, 2(1):253–264, 2009. 11, 12

[26] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981. 4, 8, 14

[27] H. Liu, H. Jin, X. Liao, L. Hu, and C. Yu. Live migration of virtual machine based on full system trace and replay. In *HPDC*, pages 101–110, 2009. 11

[28] D. B. Lomet, A. Fekete, G. Weikum, and M. J. Zwilling. Unbundling transaction services in the cloud. In *CIDR Perspectives*, 2009. 3, 11

[29] J. B. Rothnie Jr., P. A. Bernstein, S. Fox, N. Goodman, M. Hammer, T. A. Landers, C. L. Reeve, D. W. Shipman, and E. Wong. Introduction to a System for Distributed Databases (SDD-1). *ACM Trans. Database Syst.*, 5(1):1–17, 1980. 11

[30] G. H. Sockut and B. R. Iyer. Online reorganization of databases. *ACM Comput. Surv.*, 41(3):1–136, 2009. 11

[31] A. A. Soror, U. F. Minhas, A. Aboulnaga, K. Salem, P. Kokosielis, and S. Kamath. Automatic virtual machine configuration for database workloads. In *SIGMOD*, pages 953–966, 2008. 1, 2, 11

[32] The Transaction Processing Performance Council. TPC-C benchmark (Version 5.10.1), 2009. 2, 9, 10

[33] C. D. Weissman and S. Bobrowski. The design of the force.com multitenant internet application development platform. In *SIGMOD*, pages 889–896, 2009. 1, 2, 11

[34] F. Yang, J. Shanmugasundaram, and R. Yerneni. A scalable data platform for a large number of small applications. In *CIDR*, 2009. 1, 3, 11

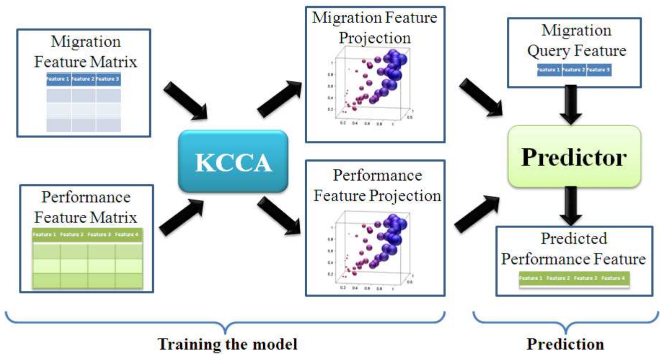

Migration Feature Matrix
Feature 1 Feature 2 Feature 3
Performance Feature Matrix
Feature 1 Feature 2 Feature 3 Feature 4
KCCA
Migration Feature Projection
Performance Feature Projection
Migration Query Feature
Feature 1 Feature 2 Feature 3
Predictor
Predicted Performance Feature
Feature 1 Feature 2 Feature 3 Feature 4
Training the model
Prediction

Figure 7: Illustration of KCCA for modeling and predicting the cost of migration. In the feature projection space, the spheres represent clusters.

APPENDIX

A. FUTURE EXTENSIONS

We now discuss some extensions as part of our future work. We first describe a new technique for synchronizing the state during migration. We then discuss a technique for modeling the cost migration.

A.1 Replication based copy

The iterative copy technique assumes the database state is a black box and performs copying at the level of database pages and other main memory structures, similar to the memory page copying technique used in VM migration [7]. The replication based technique, on the other hand, uses the operations of the transactions for synchronizing the state of C_{migr}. In addition to taking a snapshot of the initial state of C_{migr} during migration initialization (Phase 1a), the query router is also notified of the impending migration – i.e., the destination node N_{dst} and the cell (C_{migr}) being migrated. All subsequent operations for C_{migr} are then replicated and also sent to N_{dst}, in addition to sending them to N_{src}. Even though the operations of the transactions are replicated, N_{src} still remains the *owner* of C_{migr} and is responsible for executing the transactions; N_{dst} batches the operations and applies them in the commit order determined by N_{src}. Referring to Figure 2, transactions $T_{11}, T_{12} \ldots$ which are active at the start of the migration phase are sent to both N_{src} and N_{dst}. N_{src} executes the transactions and commits them in a serializable order. In the mean while, the transactions are queued at N_{dst}. The serialization order of transactions at N_{src} is copied to N_{dst}, and the batched transactions are then executed as per the equivalent serial order notified by N_{src}. The intuition behind this approach is that given the initial state of C_{migr} which was captured at N_{src}, the state of the cell before the atomic handover is an application of the operations of the transactions in the same order as it was applied at N_{src}. Note that even though the operations between two concurrently executing transactions can be interleaved, serializability allows us to apply operations one transaction at a time in the equivalent serial order. This approach allows replication without the use of expensive distributed commit or synchronous replication protocols. As earlier, once the amount of state (in this case the equivalent serial order of committed transactions) converges in subsequent iterations, we initiate the atomic handover phase (Phase 1c).

This technique relies on the equivalent serializable order of the execution history. Determining this order can vary depending on the concurrency control used for scheduling by the transaction managers. For instance, in an OCC based scheduler [26], committed transactions are assigned transaction numbers by the system, which determines the equivalent serial order. On the other hand, in a 2PL scheduler [17], transactions are not assigned numbers, but the order in which transactions reach their lock point (i.e. the instant a transaction releases its first lock and hence cannot reacquire another lock) dictates the serializable order. Transactions can then be numbered in the order in which they reach the lock point. In the future, we would like to extend this technique and compare the trade-offs when compared to the iterative copy technique.

A.2 Cost Model

Live migration of a *cell* in a multitenant DBMS is one of the critical features for load balancing and elasticity, and the controller in the system (see Figure 3) decides which *cell* to migrate, when to migrate, and the destination of migration. In order for the controller to model the behavior of the system before, during, and after migration, an estimate of the cost of migration is important. In this section, we discuss a technique to estimate and predict the cost of migration. Note that for a system, the cost of migration can have multiple dimensions, representative examples include: the time taken to complete migration (Phase 1), the amount of data transferred during migration, number of transactions aborted due to migration, percentage increase in latency of transactions during and after migration. In a system trying to optimize cost of operation, the cost can also be measured as the percentage change in operating cost. Our goal is to develop a cost model that is extensible in terms of predicting the cost(s) associated with migration as per the requirement of the system.

We use a statistical machine learning technique for multiple metric performance predictions, a technique also used in a recent study by Ganapathi et al. [19]. The authors propose the use of a technique known as Kernel Canonical Correlation Analysis (KCCA) [4]. We use a set of input features that describe the migration task, and the corresponding set of performance features that describe the cost of migration, and kernel functions are used to project the vectors onto dimensions of maximum correlation. These projected vectors can then be used for prediction for new workloads. Figure 7 provides an illustration of the modeling technique. The model is first trained using the input migration feature matrix and the corresponding performance feature matrix obtained by performing the migration and using the observed cost. More details about the internals of KCCA, and the use of KCCA for prediction can in found in [4, 19]. In this section, we focus on identifying the features that describe the migration task. Recall that during migration, the database buffer is being migrated. The size of the buffer, and the number of updates to the buffer while the iterative copy is in progress would impact the cost of migration. In addition, the load on the system would also determine the impact of migration for transactions on C_{migr} as well as other *cells* on N_{src} and N_{dst}. Using this understanding, we use the following features for describing the migration task: (*i*) size of the database cache that needs migration; (*ii*) expected transactional load (in transactions per second) during migration; (*iii*) approximate read/write workload distribution; and (*iv*) average load on N_{src} and N_{dst}. Selecting performance feature is straightforward, we use: (*i*) amount of data transferred during migration; (*ii*) percentage increase in latency of transactions to C_{migr}; (*iii*) percentage increase in latency of transactions to other *cells* at N_{src} and N_{dst}. (*iv*) time taken for migration; (*v*) time for which C_{migr} is unavailable; and (*vi*) number of aborted transactions on C_{migr} as a result of migration. In the future, we plan to evaluate the effectiveness of this technique in predicting the cost of migration.