Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms

Aaron J. Elmore Sudipto Das Divyakant Agrawal Amr El Abbadi
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106-5110, USA
{aelmore, sudipto, agrawal, amr}@cs.ucsb.edu

ABSTRACT

Multitenant data infrastructures for large cloud platforms hosting hundreds of thousands of applications face the challenge of serving applications characterized by small data footprint and unpredictable load patterns. When such a platform is built on an elastic pay-per-use infrastructure, an added challenge is to minimize the system's operating cost while guaranteeing the tenants' service level agreements (SLA). *Elastic load balancing* is therefore an important feature to enable scale-up during high load while scaling down when the load is low. *Live migration*, a technique to migrate tenants with minimal service interruption and no downtime, is critical to allow lightweight elastic scaling. We focus on the problem of live migration in the database layer. We propose **Zephyr**, a technique to efficiently migrate a live database in a shared nothing transactional database architecture. Zephyr uses phases of on-demand pull and asynchronous push of data, requires minimal synchronization, results no service unavailability and few or no aborted transactions, minimizes the data transfer overhead, provides ACID guarantees during migration, and ensures correctness in the presence of failures. We outline a prototype implementation using an open source relational database engine and an present a thorough evaluation using various transactional workloads. Zephyr's efficiency is evident from the few tens of failed operations, 10-20% change in average transaction latency, minimal messaging, and no overhead during normal operation when migrating a live database.

Categories and Subject Descriptors

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

General Terms

Design, Experimentation, Performance, Reliability

Keywords

Cloud computing, multitenancy, elastic data management, database migration, shared nothing architectures

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'11, June 12–16, 2011, Athens, Greece.
Copyright 2011 ACM 978-1-4503-0661-4/11/06 ...$10.00.

1. INTRODUCTION

The increasing popularity of service oriented computing has seen hundreds of thousands of applications being deployed on various cloud platforms [15]. The sheer scale of the number of application databases, or **tenants**, and their small footprint (both in terms of size and load) mandate a shared infrastructure to minimize the operating cost [11, 19, 25, 26]. These applications often have unpredictable load patterns, such as flash crowds originating from a sudden and viral popularity, resulting in the tenants' resource requirements changing with little notice. Load balancing is therefore an important feature to minimize the impact of a heavily loaded tenant on the other co-located tenants. Furthermore, a platform deployed on a pay-per-use infrastructure (like Amazon EC2) provides the potential to minimize the system's operating cost. *Elasticity*, i.e. the ability to scale up to deal with high load while scaling down in periods of low load, is a critical feature to minimize the operating cost. **Elastic load balancing** is therefore a first class feature in the design of modern database management systems for the cloud [11, 12], and requires a low cost technique to migrate tenants between hosts, a feature referred to as **live migration** [8, 20].[1]

Our focus is the problem of live migration in the database layer supporting a multitenant cloud platform where the service provider manages the applications' databases. Force.com, Microsoft Azure, and Google AppEngine are examples of such multitenant cloud platforms. Even though a number of techniques are prevalent to scale the DBMS layer, elasticity is often ignored primarily due to static infrastructure provisioning. In a multitenant platform built on an infrastructure as a service (IaaS) abstraction, elastic scaling allows minimizing the system's operating cost leveraging the pay-per-use pricing. Most current DBMSs, however, only support heavyweight techniques for elastic scale-up where adding new nodes requires manual intervention or long service disruption to migrate a tenant's database to these newly added nodes. Therefore, to enable lightweight elasticity as a first class notion, live migration is a critical functionality.

We present **Zephyr**,[2] a technique for live migration in a shared nothing transactional database. Das et al. [13] proposed a solution for live database migration in a shared storage architecture while Curino et al. [10] outlined a possible solution for live migration in a shared nothing architecture. Zephyr is the first complete solution for live migration in a shared nothing database architecture.

Zephyr minimizes service interruption for the tenant being migrated by introducing a synchronized *dual mode* that allows both the source and destination to simultaneously execute transactions

[1] Our use of the term migration is different from migration between different database versions or different schema.

[2] Zephyr, meaning a gentle breeze, is symbolic of the lightweight nature of the proposed technique.

for the tenant. Migration starts with the transfer of the tenant's metadata to the destination which can then start serving new transactions, while the source completes the transactions that were active when migration started. Read/write access (called **ownership**) on database pages of the tenant is partitioned between the two nodes with the source node owning all pages at the start and the destination acquiring page ownership on-demand as transactions at the destination access those pages. The index structures are replicated at the source and destination and are immutable during migration. Lightweight synchronization between the source and the destination, only during the short dual mode, guarantees serializability, while obviating the need for two phase commit [16]. Once the source node completes execution of all active transactions, migration completes with the ownership transfer of all database pages owned by the source to the destination node. Zephyr thus allows migration of individual tenant databases that share a database process at a node and where live VM migration [8] cannot be used.

Zephyr guarantees no service disruption for other tenants, no system downtime, minimizes data transferred between the nodes, guarantees *safe* migration in the presence of failures, and ensures the strongest level of transaction isolation. Zephyr uses standard tree based indices and lock based concurrency control, thus allowing it to be used in a variety of DBMS implementations. Zephyr does not rely on replication in the database layer, thus providing greater flexibility in selecting the destination for migration, which might or might not have the tenant's replica. However, considerable performance improvement is possible in the presence of replication when a tenant is migrated to one of the replicas.

We implemented Zephyr in an open source RDBMS. Our evaluation using a variety of transactional workloads shows that Zephyr results in only a few tens of failed operations, compared to hundreds to thousands of failed transactions when using a simple heavyweight migration technique. Zephyr results in no operational overhead during normal operation, minimal messaging overhead during migration, and between 10-20% increase in average transaction latency compared to an execution where no migration was performed. These results demonstrate the lightweight nature of Zephyr allowing live migration with minimal service interruption.

The main contributions of this paper are as follows:

- We present Zephyr, the first complete end-to-end solution for live migration in a shared nothing database architecture.
- We present a detailed analysis of the guarantees provided by Zephyr, analyze the associated trade-offs, and prove safety and liveness guarantees in the presence of failures.
- We provide a detailed evaluation of our prototype evaluation using a variety of workloads that demonstrate interesting trade-offs in performance.

The rest of the paper is organized as: Section 2 provides background on multitenancy models, the system model used, migration cost measures, and describes some straightforward migration techniques. Section 3 describes Zephyr, Section 4 proves transaction correctness, and Section 5 discusses some extensions and optimizations. Section 6 describes the details of a prototype implementation and Section 7 provides a detailed evaluation. Section 8 provides a survey of related literature and Section 9 concludes the paper.

2. BACKGROUND

2.1 Multitenancy Models

The three most common multitenancy models are: *shared table*, *shared process*, and *shared machine*. The shared table model is used primarily in the context of Software as a Service (SaaS) such as in Saleforce.com [25]. It allows efficient resource sharing while providing row level isolation between tenants. All tenants' data is stored in a few big tables, with additional access structures or tables for metadata management and query processing. For instance, in the Saleforce.com architecture [25], a centralized table (called the *heap*) stores all tenants' data, while additional tables (called *pivot tables*) are used to access data in the heap. To allow schema flexibility, the heap table cannot use native types and hence custom query processing and indexing layers are needed. Such a design is suitable when most of the tenants have almost similar schema with minor customizations per tenant. However, this model leads to various challenges when application schemas differ considerably. Moreover, row level isolation between tenants complicates on-demand tenant migration.

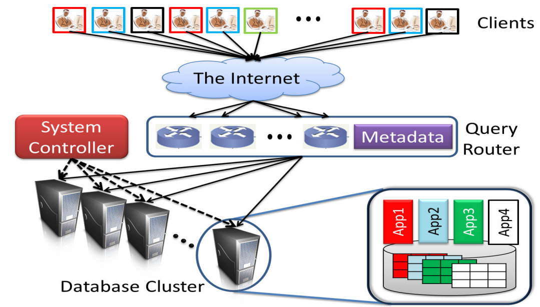


Figure 1: A shared nothing multitenant DBMS architecture.

The shared machine model [24] uses different virtual machines (VM) or database processes per tenant, thus allowing for stronger VM or OS level isolation. Using a VM per tenant can leverage VM migration techniques [8, 20] for elastic load balancing. However, experimental studies have shown that this stronger isolation comes at the cost of increased overhead during normal operation resulting in inefficient resource sharing between tenants [10]. This overhead due to redundant components can be significantly reduced by the shared process model.

The shared process model allows independent schemas for tenants while sharing the database process amongst multiple tenants [3, 10, 11, 26]. This model provides better isolation compared to the shared table model while allowing effective sharing and consolidation of multiple tenants in the same database process [10]. However, such a model cannot leverage VM migration to move individual tenants from a shared database process. This calls for efficient and low cost live database migration techniques, such as Zephyr, to provide a good mix of effective sharing and ease of migration.

2.2 System Architecture

We use a standard shared nothing database model for transaction processing (OLTP) systems executing short running transactions, with a two phase locking [14] based scheduler, and a page based model with a B+ tree index [4]. Figure 1 provides an overview of the architecture. Following are the salient features of the system. First, clients connect to the database through **query routers** that handle client connections and hide the physical location of the tenant's database. Routers store this mapping as metadata which is updated whenever there is a migration. Second, we use the **shared process** multitenancy model which strikes a balance between isolation and scale. Conceptually, each tenant has its own transaction manager and buffer pool. However, since most current systems do not support this, we use a design where co-located tenants share all resources within a database instance, but is shared nothing across

nodes. Finally, there exists a **system controller** that determines the tenant to be migrated, the initiation time, and the destination of migration. The system controller gathers usage statistics and builds a model to optimize the system's operating cost while guaranteeing the tenant's SLAs. The detailed design and implementation of the controller is orthogonal to the problem considered in this paper and is left for future work.

2.3 Migration Cost

The goal of any migration technique is to minimize migration cost. Das et al. [13] discuss some measures to quantify the cost of migration. Low migration cost allows the system controller to effectively use it for elastic load balancing.

- **Service interruption:** Live migration must ensure minimal service interruption for the tenant being migrated and should not result in downtime.[3] We use *downtime* for an entire system outage and *service interruption* for small interruption in service for some tenants. The number of transactions or operations aborted during migration is a measure of service interruption and is used to determine the impact of migration on the tenant's SLA.
- **Migration Overhead:** Migration overhead is the additional work done or resources consumed to enable and perform migration. This cost also includes performance impact as a result of migration, such as increase in transaction latency or reduction in throughput. This comprises of:
  - *Overhead during normal operation:* Additional work done during normal database operation to enable migration.
  - *Overhead during migration:* Performance impact on the tenant being migrated as well as other tenants co-located at the source or destination of migration.
  - *Overhead after migration:* Performance impact on transactions executing at the destination node after migration.
- **Additional data transferred:** Since the source and destination of migration do not share storage, the persistent image of the database must be moved from the source to the destination. This measure accounts for any data transfer that migration incurs, in addition to transferring the persistent database image.

2.4 Known Migration Techniques

Most enterprise database infrastructures are statically provisioned for the peak capacity. Migrating tenants on-demand for elasticity is therefore not a common operation. As a result, live migration is not a feature supported off-the-shelf by most database systems, resulting in the use of heavyweight techniques. We now discuss two known techniques for database migration.

Stop and copy: This is the simplest and arguably most heavy-handed approach to migrate a database. In this technique, the system stops serving updates for the tenant, checkpoints the state, moves the persistent image, and restarts the tenant at the destination. This technique incurs a long service interruption and a high post migration penalty to warm up the cache at the destination. The advantage is its simplicity and efficiency in terms of minimizing the amount of data transferred. However inefficient this technique might be, this is the only technique available in many current database systems (including RDBMSs like MySQL and *Key-Value* stores such as HBase) to migrate a tenant to a node which is not already running a replica.

Iterative State Replication: The long unavailability of *stop and copy* arises due to the time taken to create the checkpoint and to copy it to the destination. An optimization, Iterative State Replication (**ISR**), is to use an iterative approach, similar to [13], where the checkpoint is created and iteratively copied. The source checkpoints the tenant's database and starts migrating the checkpoint to the destination, while it continues serving requests. While the destination loads the checkpoint, the source maintains the differential changes which are iteratively copied until the amount of change to be transferred is small enough or a maximum iteration count is reached. At this point, a final stop and copy is performed. The iterative copy can be performed using either page level copying or shipping the transaction log and replaying it at the destination.

Consider applications such as shopping cart management or online games such as Farmville that represent workloads with a high percentage of reads followed by updates, and that require high availability for continued customer satisfaction. In ISR, the tenant's database is unavailable to updates during the final stop phase. Even though the system can potentially serve read-only transactions during this window, *all* transactions with at least one update will be aborted during this *small* window. On the other hand, Zephyr does not render the tenant unavailable by allowing concurrent transaction execution at both the source and the destination. However, during migration, Zephyr will abort a transaction in two cases: (i) if at the source it accesses an already migrated page, or (ii) if at either node, it issues an update operation that modifies the index structures. Hence, Zephyr may abort a fraction of update transactions during migration. The exact impact of either technique on transaction execution will depend on the workload and other tenant characteristics, and needs to be evaluated experimentally.

The iterative copying of differential updates in ISR can lead to more data being transferred during migration, especially for update heavy workloads that result in more changes to the database state. Zephyr, on the other hand, migrates a database page only once and hence is expected to have lower data transfer overhead.

Since ISR creates multiple checkpoints during migration, it will result in higher disk I/O at the source. Therefore, when migrating a tenant from a heavily loaded source node, this additional disk I/O can result in significant impact on co-located tenants which are potentially already disk I/O limited due to increased load. However, due to the log replay, the destination will start with a warm cache and hence will minimize the post migration overhead. On the other hand, Zephyr does not incur additional disk I/O at the source due to checkpointing, but the cold start at the destination results in higher post migration overhead and more I/O at the destination. Therefore, Zephyr results in less overhead at the source and is suitable for scale-out scenarios where the source is already heavily loaded, while ISR is attractive for consolidation during scale-down where it will result in lower impact on tenants co-located at the destination.

Finally, since ISR creates a replica of the tenant's state at another node, it can iteratively copy the updates to multiple nodes, thus creating replicas on the fly during migration. Zephyr however does not allow for this easy extension.

It is therefore evident that ISR and Zephyr are both viable techniques for live database migration; a detailed experimental comparison between the two is left for future work. This paper focusses on Zephyr since it is expected to have minimal service interruption which is critical to ensure high tenant availability.

3. ZEPHYR DESIGN

In this section, we provide an overview of Zephyr using some simplifying assumptions to ease presentation. We assume *no failures*, *small tenants* limited to a single node in the system, and *no replication*. Furthermore, the *index structures* are made *immutable* during migration. Failure handling and correctness is discussed in

[3] A longer interruption might result in a penalty. For instance, in platforms like Windows Azure, service availability below 99.9% results in a penalty. http://www.microsoft.com/windowsazure/sla/

Notation	Description
$\mathbb{D}_M$	The tenant database being migrated
$\mathbb{N}_S$	Source node for $\mathbb{D}_M$
$\mathbb{N}_D$	Destination node for $\mathbb{D}_M$
T_{Si}, T_{Di}	Transaction executing at nodes $\mathbb{N}_S$ and $\mathbb{N}_D$ respectively
P_k	Database page k

Table 1: Notational Conventions.

Section 4, while an extended design relaxing these assumptions is described in Section 5. The notational conventions used are summarized in Table 1.

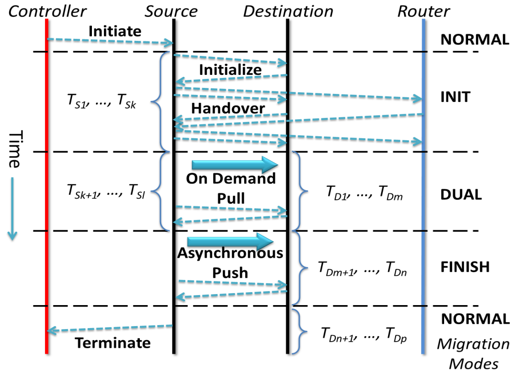
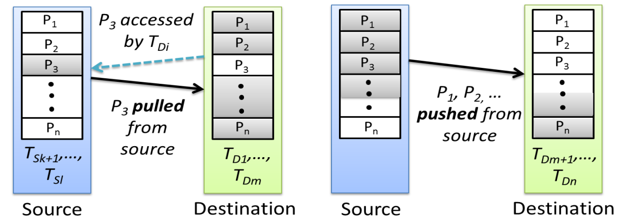


Figure 2: Timeline for different phases during migration. Vertical lines correspond to the nodes, the broken arrows represent control messages and the thick solid arrows represent data transfer. Time progresses from top towards the bottom.

3.1 Design Overview

Zephyr's main design goal is to minimize the service interruption resulting from migrating a tenant's database ($\mathbb{D}_M$). Zephyr does not incur a stop phase where $\mathbb{D}_M$ is unavailable for executing updates; it uses a sequence of three modes to allow the migration of $\mathbb{D}_M$ while transactions are executing on it. During normal operation (called the **Normal Mode**), $\mathbb{N}_S$ is the node serving $\mathbb{D}_M$ and executing all transactions $T_{S1}, \ldots, T_{Sk}$ on $\mathbb{D}_M$. A node that has the rights to execute update transactions on $\mathbb{D}_M$ is called an **owner** of $\mathbb{D}_M$. Once the system controller determines the destination for migration ($\mathbb{N}_D$), it notifies $\mathbb{N}_S$ which initiates migration to $\mathbb{N}_D$. Figure 2 shows the timeline of this migration algorithm and the control and data messages exchanged between the nodes. As time progresses from the top to the bottom, Figure 2 shows the progress of the different migration modes, starting from the **Init Mode** which initiates migration, the **Dual Mode** where both $\mathbb{N}_S$ and $\mathbb{N}_D$ share the ownership of $\mathbb{D}_M$ and simultaneously execute transactions on $\mathbb{D}_M$, and the **Finish Mode** which is the last step of migration before $\mathbb{N}_D$ assumes full ownership of $\mathbb{D}_M$. Figure 3 shows the transition of $\mathbb{D}_M$'s data through the three migration modes, depicted using ownership of database pages and executing transactions.

Init Mode: In the Init Mode, $\mathbb{N}_S$ bootstraps $\mathbb{N}_D$ by sending the minimal information (the **wireframe** of $\mathbb{D}_M$) such that $\mathbb{N}_D$ can execute transactions on $\mathbb{D}_M$. The wireframe consists of the schema and data definitions of $\mathbb{D}_M$, index structures, and user authentication information. Indices migrated include the internal nodes of the clustered index storing the database and all secondary indices. Non-indexed attributes are accessed through the clustered index. In this mode, $\mathbb{N}_S$ is still the unique owner of $\mathbb{D}_M$ and executes transactions ($T_{S1}, \ldots, T_{Sk}$) without synchronizing with any other node. Therefore, there is no service interruption for $\mathbb{D}_M$ while $\mathbb{N}_D$ initializes the necessary resources for $\mathbb{D}_M$. We assume a B+ tree index, where the internal nodes of the index contain only the keys while the actual data pages are in the leaves. The wireframe therefore only includes these internal nodes of the indices for the database tables. Figure 4 illustrates this, where the part of the tree enclosed in a rectangular box is the *index wireframe*. At $\mathbb{N}_S$, the wireframe is constructed with minimal impact on concurrent operations using shared multi-granularity intention locks on the indices. When $\mathbb{N}_D$ receives the wireframe, it has $\mathbb{D}_M$'s metadata, but the data is still owned by $\mathbb{N}_S$. Since migration involves a gradual transfer of page level ownership, both $\mathbb{N}_S$ and $\mathbb{N}_D$ must maintain a list of owned pages. We use the B+ tree index for tracking page ownership. A valid pointer to a database page implies unique page ownership, while a sentinel value (NULL) indicates a missing page. In the init mode, $\mathbb{N}_D$ therefore initializes all the pointers to the leaf nodes of the index to the sentinel value. Once $\mathbb{N}_D$ completes initialization of $\mathbb{D}_M$, it notifies $\mathbb{N}_S$, which then initiates the transition to the dual mode. $\mathbb{N}_S$ then executes the **Atomic Handover** protocol which notifies the *query router* to direct all new transactions to $\mathbb{N}_D$.



(a) Dual Mode. (b) Finish Mode.

Figure 3: Ownership transfer of the database pages during migration. P_i represents a database page and a white box around P_i represents that the node currently *owns* the page.

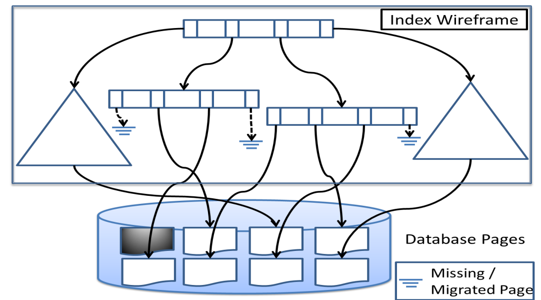


Figure 4: B+ tree index structure with page ownership information. A sentinel marks missing pages. An allocated database page without ownership is represented as a grayed page.

Dual Mode: In the dual mode, both $\mathbb{N}_S$ and $\mathbb{N}_D$ execute transactions on $\mathbb{D}_M$, and database pages are migrated to $\mathbb{N}_D$ *on-demand*. All new transactions ($T_{D1}, \ldots, T_{Dm}$) arrive at $\mathbb{N}_D$, while $\mathbb{N}_S$ continues executing transactions that were active at the start of this mode ($T_{Sk+1}, \ldots, T_{Sl}$). Since $\mathbb{N}_S$ and $\mathbb{N}_D$ share ownership of $\mathbb{D}_M$, they synchronize to ensure transaction correctness. Zephyr however requires minimal synchronization between these nodes.

At $\mathbb{N}_S$, transactions execute normally using local index and page level locking, until a transaction T_{Sj} accesses a page P_j which has already been migrated. In our simplistic design, a database page is

migrated only once. Therefore, such an access fails and the transaction is aborted. When a transaction T_{Di} executing at $\mathbb{N}_D$ accesses a page P_i that is not owned by $\mathbb{N}_D$, it **pulls** P_i from $\mathbb{N}_S$ *on demand* (pull phase as shown in Figure 3(a)); this pull request is serviced only if P_i is not locked at $\mathbb{N}_S$, in which case the request is blocked. As the pages are migrated, both $\mathbb{N}_S$ and $\mathbb{N}_D$ update their ownership mapping. Once $\mathbb{N}_D$ receives P_i, it proceeds to execute T_{Di}. Apart from fetching missing pages from $\mathbb{N}_S$, transactions at $\mathbb{N}_S$ and $\mathbb{N}_D$ do not need to synchronize. Due to our assumption that the index structure cannot change at $\mathbb{N}_S$, local locking of the index structure and pages is enough. This ensures minimal synchronization between $\mathbb{N}_S$ and $\mathbb{N}_D$ only during this short dual mode, while ensuring serializable transaction execution.

When $\mathbb{N}_S$ has finished executing all transactions $T_{Sk+1}, \ldots, T_{Sl}$ that were active at the start of dual mode (i.e. $\mathbb{T}(\mathbb{N}_S) = \phi$), it initiates transfer of exclusive ownership to $\mathbb{N}_D$. This transfer is achieved through a handshake between $\mathbb{N}_S$ and $\mathbb{N}_D$ after which both nodes enter the finish mode for $\mathbb{D}_M$.

Finish Mode: In the finish mode, $\mathbb{N}_D$ is the only node executing transactions on $\mathbb{D}_M$ ($T_{Dm+1}, \ldots, T_{Dn}$), but does not yet have ownership of all the database pages (Figure 3(b)). In this phase, $\mathbb{N}_S$ **pushes** the remaining database pages to $\mathbb{N}_D$. While the pages are migrated from $\mathbb{N}_S$, if a transaction T_{Di} accesses a page that is not yet owned by $\mathbb{N}_D$, the page is requested as a *pull* from $\mathbb{N}_S$ in a way similar to that in the dual mode. Ideally, $\mathbb{N}_S$ must migrate the pages at the highest possible transfer rate such that the delays resulting from $\mathbb{N}_D$ fetching missing pages is minimized. However, such a high throughput push can impact other tenants co-located at $\mathbb{N}_S$ and $\mathbb{N}_D$. Therefore, the rate of transfer is a trade-off between the tenant SLAs and migration overhead. The page ownership information is also updated during this bulk transfer. When all the database pages have been moved to $\mathbb{N}_D$, $\mathbb{N}_S$ initiates the termination of migration so that operation switches back to the normal mode. This again involves a handshake between $\mathbb{N}_S$ and $\mathbb{N}_D$. On successful completion of this handshake, it is guaranteed that $\mathbb{N}_D$ has a persistent image of $\mathbb{D}_M$, and so $\mathbb{N}_S$ can safely release all of $\mathbb{D}_M$'s resources. $\mathbb{N}_D$ executes transactions on $\mathbb{D}_M$ without any interaction with $\mathbb{N}_S$. Once migration terminates, $\mathbb{N}_S$ notifies the system controller.

3.2 Migration Cost Analysis

Migration cost in Zephyr results from copying the initial wireframe, operation overhead during migration, and transactions or operations aborted during migration. In the wireframe transferred, the schema and authentication information is typically small. The indices for the tables however have a non-trivial size. A simple analysis provides an estimate of index sizes. Assuming 4KB pages, 8 byte keys (integers or double precision floating point numbers), and 4 byte pointers, each internal node in the tree can hold about $4096/12 \approx 340$ keys. Therefore, a three-level B+ tree can have up to $340^2 = 115600$ leaf nodes, which can index a $(115600 \times 4096 \times 0.8)/10^6 \approx 400$ MB database, assuming 80% page utilization. Similarly, a four-level tree can index a 125 GB database. For a three level tree, the size of the wireframe is a mere $340 \times 4096/10^6 \approx 1.4$ MB while for a 4-level tree, it is about 400 MB. For most multitenant databases whose representative sizes are in the range of hundreds of megabytes to a few gigabytes, an index size of the order of tens of megabytes is a realistic conservative estimate [25,26]. These index sizes add up for the multiple tables and indices maintained for the database.

Overhead during migration stems from creating the wireframe and fetching pages over the network. $\mathbb{N}_S$ uses standard *multi-granularity* locking [17] of the index to construct the index wireframe. This scan to create the wireframe needs intention read locks at the internal nodes which only conflict with write locks [4] on the internal node. Therefore, this scan can execute in parallel with any transaction T_{Si} executing at $\mathbb{N}_S$, only blocking update transactions that result in an update in the index structure that requires a conflicting write lock on an internal node. On the other hand, on-demand pull of a page from $\mathbb{N}_S$ over the network is also not very expensive compared to fetches from the disk – disks have an access latency of about a millisecond while most data center networks have round trip latencies of less than a millisecond. The cost incurred by this remote pull is therefore of the same order as a cache miss during normal operation resulting in a disk access. Assuming an OLTP workload with predominantly small transactions, the period for which $\mathbb{D}_M$ remains in the dual mode is expected to be small. Therefore, the cost incurred in this short period in the dual mode is expected to be small.

Another contributor to the migration cost is failed transactions at $\mathbb{N}_S$ resulting from accesses to pages that have been migrated. In its simplest form as described, Zephyr does not guarantee zero transaction failure; this however can be guaranteed by an extended design as shown later in Section 5.

4. CORRECTNESS AND FAULT TOLERANCE

Any migration technique should guarantee transaction correctness and migration safety in the presence of arbitrary failures. We first prove that Zephyr guarantees serializable isolation even during migration. We then prove the atomicity and durability properties of both transaction execution as well the migration protocol.

4.1 Isolation guarantees

Transactions executing with serializable isolation, use two phase locking (2PL) [14] with multi-granularity [17]. In the init mode and finish mode, only one of $\mathbb{N}_S$ and $\mathbb{N}_D$ is executing transactions on $\mathbb{D}_M$. The init mode is equivalent to normal operation while in finish mode, $\mathbb{N}_S$ acts as the storage node for the database serving pages on demand. Guaranteeing serializability is straightforward in these modes. We only need to prove correctness in the dual mode where both $\mathbb{N}_S$ and $\mathbb{N}_D$ are executing transactions on $\mathbb{D}_M$. In the dual mode, $\mathbb{N}_S$ and $\mathbb{N}_D$ share the internal nodes of the index which are immutable in our design, while the leaf nodes (i.e. the data pages) are still uniquely owned by one of the two nodes. To guarantee serializability, we first prove that the phantom problem [14] is impossible, and then prove general serializability of transactions executing in the dual mode. The phantom problem arises from predicate based accesses where a transaction inserts or deletes an item that matches the predicate of a concurrently executing transaction.

LEMMA 1. ***Phantom problem:*** *Local predicate locking at the internal index nodes and exclusive page level locking between nodes is enough to ensure impossibility of phantoms.*

PROOF. **Proof by contradiction:** Assume for contradiction that a phantom is possible resulting in predicate instability. Let T_1 and T_2 be two transactions such that T_1 has a predicate and T_2 is inserting (or deleting) at least one element that matches T_1's predicate. T_1 and T_2 cannot be executing at the same node, since local predicate locking would prevent such a behavior. Therefore, these transactions must be executing on different nodes. Without loss of generality, assume that T_1 is executing at $\mathbb{N}_S$ and T_2 is executing at $\mathbb{N}_D$. Let T_1's predicate match pages $P_i, P_{i+1}, \ldots, P_j$ representing a range of keys. Since Zephyr does not allow an update that changes the index during migration, therefore, T_2 cannot insert to a newly created page at $\mathbb{N}_D$. Therefore, if T_2 was inserting to (or

deleting from) one of the pages $P_i, P_{i+1}, \ldots, P_j$ while T_1 was executing, then it implies that both $\mathbb{N}_S$ and $\mathbb{N}_D$ have ownership of the page. This results in a contradiction. Hence the proof. □

LEMMA 2. ***Serializability at a node:*** *Transactions executing at the same node (either $\mathbb{N}_S$ or $\mathbb{N}_D$) cannot have a cycle in the conflict graph involving these transactions.*

The proof of Lemma 2 follows directly from the correctness of 2PL [14], since all transactions executing at the same node use 2PL for concurrency control.

LEMMA 3. *Let T_{Sj} be a transaction executing at $\mathbb{N}_S$ and T_{Di} be a transaction executing at $\mathbb{N}_D$, it is impossible to have a conflict dependency $T_{Di} \rightarrow T_{Sj}$.*

PROOF. **Proof by contradiction:** Assume for contradiction that there exists a dependency of the form $T_{Di} \rightarrow T_{Sj}$. This implies that T_{Sj} makes a conflicting access to an item in page P_i after T_{Di} accessed P_i. Due to the two phase locking rule, the conflict $T_{Di} \rightarrow T_{Sj}$ implies that commit of T_{Di} precedes the conflicting access by T_{Sj}, which in turn implies that T_{Sj} accesses P_i after it was migrated to $\mathbb{N}_D$ as a result of an access by T_{Di}. This leads to a contradiction since in Zephyr, once P_i is migrated from $\mathbb{N}_S$ to $\mathbb{N}_D$, all subsequent accesses to P_i at $\mathbb{N}_S$ fail. Hence the proof. □

Corollary 4 follows by applying induction on Lemma 3.

COROLLARY 4. *It is impossible to have a path $T_{Di} \rightarrow \ldots \rightarrow T_{Sj}$ in the conflict graph.*

THEOREM 5. ***Serializability in dual mode.*** *It is impossible to have a cycle in the conflict graph of transactions executing in the dual mode.*

PROOF. **Proof by contradiction:** Assume for contradiction that there exists a set of transactions $T_1, T_2, \ldots, T_k$ such that there is a cycle $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_k \rightarrow T_1$ in the conflict graph. If all transactions are executing at the same node, then this is a contradiction to Lemma 2. Consider the case where some transactions are executing at $\mathbb{N}_S$ and some at $\mathbb{N}_D$. Let us first assume that T_1 executed at $\mathbb{N}_S$. Let T_i be the first transaction in the sequence which executed at $\mathbb{N}_D$. The above cycle implies that there exists a path of the form $T_i \rightarrow \ldots \rightarrow T_1$ where T_i executed at $\mathbb{N}_D$ and T_1 executed at $\mathbb{N}_S$. This is a contradiction to Corollary 4. Similarly, if T_1 executed at $\mathbb{N}_D$, then there exists at least one transaction T_j which executed at $\mathbb{N}_S$, which implies a path of the form $T_1 \rightarrow \ldots \rightarrow T_j$, again a contradiction to Corollary 4. Hence the proof. □

Snapshot Isolation (SI) [2], arguably the most commonly used isolation level, can also be guaranteed in Zephyr. A transaction T_i writing to a page P_i must have unique ownership of P_i, while a read can be performed from a snapshot shared by both nodes. This condition of unique page ownership is sufficient to ensure that during validation of transactions in SI, the transaction manager can detect two concurrent transactions writing to the same page and abort one. Zephyr therefore guarantees transactional isolation with minimal synchronization and without much migration overhead.

4.2 Fault tolerance

Our failure model assumes that all message transfers use reliable communication channels that guarantee in-order, at most once delivery. We consider node crash failures and network partitions; we however do not consider malicious node behavior. We assume that a node failure does not lead to loss of the persistent disk image. In case of a failure during migration, our design first recovers state of the committed transactions and then recovers the state of migration.

4.2.1 Transaction State Recovery

Transactions executing during migration use write ahead logging for transaction state recovery [4,23]. Updates made by a transaction are *forced* to the log before it commits, thus resulting in a total order on transactions executing at the node. After a crash, a node recovers its transaction state using standard log replay techniques, ARIES [23] being an example.

In the dual mode, $\mathbb{N}_S$ and $\mathbb{N}_D$ append transactions to their respective node's local transaction log. Log entries in a single log file have a local order. However, since the log for $\mathbb{D}_M$ is spread over $\mathbb{N}_S$ and $\mathbb{N}_D$, a logical global order of transactions on $\mathbb{D}_M$ is needed to ensure that the transactions from the two logs are applied in the correct order to recover from a failure during migration. The ordering of transactions is important only when there is a conflict between two transactions. If two transactions, T_S and T_D, executing on $\mathbb{N}_S$ and $\mathbb{N}_D$, conflict on item i, they must access the same database page P_i. Since at any instant of time only one of $\mathbb{N}_S$ and $\mathbb{N}_D$ is the owner of P_i, the two nodes must synchronize to arbitrate on P_i. This synchronization forms the basis for establishing a total order between the transactions. During migration, a **commit sequence number** (CSN) is assigned to every transaction at commit time, and is appended along with the commit record of the transaction. This CSN is a monotonically increasing sequence number maintained locally at the nodes and determines the order in which transactions commit. If P_i was owned by $\mathbb{N}_S$ and T_S was the last committed transaction before the migration request for P_i was made, then $CSN(T_S)$ is piggy-backed with P_i. On receipt of a page P_i, $\mathbb{N}_D$ sets its CSN as the maximum of its local CSN and that received with P_i such that at $\mathbb{N}_D$, $CSN(T_D) > CSN(T_S)$. This causal conflict ordering creates a global order per database page, where all transactions at $\mathbb{N}_S$ accessing P_i are ordered before all transactions at $\mathbb{N}_D$ that access P_i. We formally state this property as Theorem 6:

THEOREM 6. *The transaction recovery and the conflict ordering protocol ensures that for every database page, conflicting transactions are replayed in the same order in which they committed.*

4.2.2 Migration State Recovery

Migration progress is logged to guarantee atomicity and consistency in the presence of failures. Migration safety is ensured by using rigorous recovery protocols. A failure of either $\mathbb{N}_S$ or $\mathbb{N}_D$ in the dual mode or the finish mode requires coordinated recovery between the two nodes. We first discuss recovering from a failure during transition of migration modes and discuss recovery after failure in different migration modes.

Transitions of Migration Modes: During migration, a transition from one state to another is logged. Except for the transition from the init mode to dual mode, which involves the query router metadata in addition to $\mathbb{N}_S$ and $\mathbb{N}_D$, all other transitions involve only $\mathbb{N}_S$ and $\mathbb{N}_D$. Such transitions occur through a one-phase handshake between $\mathbb{N}_S$ and $\mathbb{N}_D$ (as shown in Figure 2). At the occurrence of an event triggering a state transition, $\mathbb{N}_S$ initiates the transition by sending a message to $\mathbb{N}_D$. On receipt of the message, $\mathbb{N}_D$ moves to the next migration mode, forces a log entry for this change, and sends an acknowledgment to $\mathbb{N}_S$. Receipt of this acknowledgment completes this transition and $\mathbb{N}_S$ forces another entry to its log.

If $\mathbb{N}_S$ fails before sending the message to $\mathbb{N}_D$, the mode remains unchanged when $\mathbb{N}_S$ recovers, and $\mathbb{N}_S$ re-initiates the transition. If $\mathbb{N}_S$ fails after sending the message, then it knows about the message after it recovers and establishes contact with $\mathbb{N}_D$. Therefore, a state transition results in two messages and two writes to the log. Logging of messages at $\mathbb{N}_S$ and $\mathbb{N}_D$ provides message idempotence,

detects and rejects duplicate messages resulting from failure of $\mathbb{N}_S$ or $\mathbb{N}_D$, and guarantees safety with repeating failures.

Atomic Handover: A transition from the init mode to the dual mode involves three participants ($\mathbb{N}_S$, $\mathbb{N}_D$, and the query router metadata) that must together change the state. A one-phase handshake is therefore not enough. We use the two-phase commit (2PC) [16] protocol, a standard protocol for atomic commitment over multiple sites. Once $\mathbb{N}_D$ has acknowledged the initialization of $\mathbb{D}_M$, $\mathbb{N}_S$ initiates the transition and sends a message to the router to direct all future transactions accessing $\mathbb{D}_M$ to $\mathbb{N}_D$, and a message to $\mathbb{N}_D$ to start accepting new transactions for $\mathbb{D}_M$ whose ownership is shared with $\mathbb{N}_S$. On receipt of the messages, both $\mathbb{N}_D$ and the router log their messages and reply back to $\mathbb{N}_S$. Once $\mathbb{N}_S$ has received messages from both $\mathbb{N}_D$ and the router, it logs the successful handover in its own log, changes its state to dual mode and sends acknowledgments to $\mathbb{N}_D$ and the router which update their respective states. Atomicity of this handover process follows directly from the atomicity proof of 2PC [16]. This protocol also exhibits the blocking behavior of 2PC when $\mathbb{N}_S$ (the coordinator) fails. This blocking however only affects $\mathbb{D}_M$ which is anyways unavailable as a result of $\mathbb{N}_S$'s failure. Atomic handover therefore does not introduce any additional blocking when compared to traditional 2PC where a coordinator failure blocks any other conflicting transaction.

Recovering Migration Progress: The page ownership information is critical for migration progress as well as safety. A simple fault-tolerant design is to make this ownership information durable – any page (P_i) transferred from $\mathbb{N}_S$ is immediately flushed to the disk at $\mathbb{N}_D$. $\mathbb{N}_S$ also makes this transfer persistent, either by logging the transfer or by updating P_i's parent page in the index, and flushing it to the disk. This simple solution will guarantee resilience to failure but introduces a lot of disk I/O which considerably increases migration cost and impacts other co-located tenants.

An optimized solution uses the semantics of the operation that resulted in P_i's on-demand migration. When P_i is migrated, $\mathbb{N}_S$ has its persistent (or at least recoverable) image. After the migration of P_i, if a committed transaction at $\mathbb{N}_D$ updated P_i, then the update will be in $\mathbb{N}_D$'s transaction log. Therefore, after a failure, $\mathbb{N}_D$ recovers P_i from its log and the persistent image at $\mathbb{N}_S$. The presence of a log entry accessing P_i at $\mathbb{N}_D$ implies that $\mathbb{N}_D$ owns P_i, thus preserving the ownership information after recovery. In case P_i was migrated only for a read operation or if an update transaction at $\mathbb{N}_D$ did not commit, then this migration is not persistent at $\mathbb{N}_D$. When $\mathbb{N}_D$ recovers, it synchronizes its knowledge of page ownership with that of $\mathbb{N}_S$, any missing page P_i is detected during this synchronization. For these missing pages, either $\mathbb{N}_S$ or $\mathbb{N}_D$ can be assigned P_i's ownership; assigning it to $\mathbb{N}_D$ will need copying P_i to $\mathbb{N}_D$ yet again.

On the other hand, if $\mathbb{N}_S$ fails after migrating P_i, it recovers and synchronizes its page ownership information with $\mathbb{N}_D$ when the missing P_i is detected, and $\mathbb{N}_S$ updates its ownership mapping. Failure of both $\mathbb{N}_S$ and $\mathbb{N}_D$ immediately following P_i's transfer is equivalent to the failure of $\mathbb{N}_D$ without P_i making it to the disk at $\mathbb{N}_D$, and all undecided pages can be assigned ownership as described earlier. Logging the pages at $\mathbb{N}_D$ guarantees idempotence of page transfers, thus allowing migration to deal with repeated failures and prevent lost updates at $\mathbb{N}_D$. These optimizations considerably reduces the disk I/O during the dual mode. However, in the finish mode, since pages are transferred in bulk, the pages transferred can be immediately flushed to the disk; the large number of pages per flush amortizes the disk I/O.

Since the transfer of pages to $\mathbb{N}_D$ does not force an immediate flush, after migration terminates, $\mathbb{N}_D$ must ensure a flush before the state of $\mathbb{D}_M$ can be purged at $\mathbb{N}_S$. This is achieved through a fuzzy checkpoint [4] at $\mathbb{N}_D$. A fuzzy checkpoint is used by a DBMS during normal operation to reduce the recovery time after a failure. It causes minimal disruption to transaction processing, as a background thread scans through the database cache and flushes modified pages, while the database can continue to process updates. As part of the final state transition, $\mathbb{N}_D$ initiates a fuzzy checkpoint and acknowledges $\mathbb{N}_S$ only after the checkpoint completes. After the checkpoint, $\mathbb{N}_D$ can independently recover and $\mathbb{N}_S$ can safely purge $\mathbb{D}_M$'s state. This recovery protocol guarantees that in the presence of a failure, migration recovers to a consistent point before the crash. Theorem 7 formalizes this recovery guarantee.

THEOREM 7. ***Migration recovery:*** *At any instant during migration, its progress is recoverable, i.e. after transaction state recovery is complete, database page ownership information is restored to a consistent state and every page has exactly one owner.*

Failure and Availability: A failure in any migration mode results in partial or complete unavailability. In the init mode, $\mathbb{N}_S$ is still the exclusive owner of $\mathbb{D}_M$. If $\mathbb{N}_D$ fails, $\mathbb{N}_S$ can single-handedly abort the migration or continue processing new transactions until $\mathbb{N}_D$ recovers and migration is resumed. In case this migration is aborted in the init mode, $\mathbb{N}_S$ notifies the controller which might select a new destination. A failure of $\mathbb{N}_S$ however makes $\mathbb{D}_M$ unavailable and is equivalent to $\mathbb{N}_S$'s failure during normal operation. In this case, $\mathbb{N}_D$ can abort migration at its discretion. If $\mathbb{N}_S$ fails in the dual mode or the finish mode, then $\mathbb{N}_D$ can only process transactions that access pages whose ownership was migrated to $\mathbb{N}_D$ before $\mathbb{N}_S$ failed. This is equivalent to a disk failing, making parts of the database unavailable. When $\mathbb{N}_D$ fails, $\mathbb{N}_S$ can only process transactions that do not access the migrated pages. A failure of $\mathbb{N}_D$ in the finish mode however makes $\mathbb{D}_M$ unavailable since $\mathbb{N}_D$ is now the exclusive owner of $\mathbb{D}_M$. This failure is equivalent to $\mathbb{N}_D$'s failure during normal operation.

4.3 Migration Safety and Liveness

Migration safety ensures correctness in the presence of a failure, while liveness ensures that 'something good' will eventually happen. We first establish formal definitions for safety and liveness, and then show how Zephyr guarantees these properties.

DEFINITION 1. ***Safety*** *of migration requires the following conditions:* (*i*) Transactional correctness: *serializability is guaranteed for transactions executing during migration;* (*ii*) Transaction durability: *updates from committed transactions are never lost; and* (*iii*) Migration consistency: *a failure during migration does not leave the system's state and data inconsistent.*

DEFINITION 2. ***Liveness*** *of migration requires the following conditions to be met:* (*i*) Termination: *if $\mathbb{N}_S$ and $\mathbb{N}_D$ are not faulty and can communicate with each other for a sufficiently long period during migration, this process will terminate; and* (*ii*) Starvation Freedom: *in the presence of one or more failures, $\mathbb{D}_M$ will eventually have at least one node that can execute its transactions.*

Transaction correctness follows from Theorem 5. We now prove transaction durability and migration consistency.

THEOREM 8. ***Transaction durability:*** *Changes made by a committed transaction are never lost, even in the presence of an arbitrary sequence of failure.*

PROOF. The proof follows from the following two conditions: (*i*) during normal operation, transactions force their updates to the log before commit, making them durable; and (*ii*) on successful

termination of migration, $\mathbb{N}_S$ purges its transaction log and the database image only after the fuzzy checkpoint at $\mathbb{N}_D$ completes, ensuring that changes at $\mathbb{N}_S$ and $\mathbb{N}_D$ during migration are durable. □

THEOREM 9. ***Migration consistency:*** *In the presence of arbitrary or repeated failures, Zephyr ensures:* (*i*) *updates made to data pages are consistent even in the presence of failures;* (*ii*) *a failure does not leave a page P_i of $\mathbb{D}_M$ without an owner; and* (*iii*) *both $\mathbb{N}_S$ and $\mathbb{N}_D$ are in the same migration mode.*

The condition for exclusive page ownership along with Theorem 5 and 6 ensures that updates to the database pages are always consistent, both during normal operation and after a failure. Theorem 7 guarantees that no database page is without an owner, while the atomicity of the atomic handover and other state transition protocols discussed in Section 4.2.2 guarantee that both $\mathbb{N}_S$ and $\mathbb{N}_D$ are in the same migration mode. Theorem 5, 8, and 9 therefore guarantee migration safety.

THEOREM 10. ***Migration termination:*** *If $\mathbb{N}_S$ and $\mathbb{N}_D$ are not faulty and can communicate for a long enough period, Zephyr guarantees progress and termination.*

PROOF. Zephyr successfully terminates if: (*i*) the set of active transactions ($\mathbb{T}$) at $\mathbb{N}_S$ at the start of the dual mode have completed, i.e. $\mathbb{T} = \phi$; and (*ii*) the persistent image of $\mathbb{D}_M$ is migrated to $\mathbb{N}_D$ and is recoverable. If $\mathbb{N}_S$ is not faulty in the dual mode, all transactions in $\mathbb{T}$ will eventually complete, irrespective of whether $\mathbb{N}_D$ has failed or not. If there is a failure of $\mathbb{N}_S$ at any point during migration, after recovery, it is guaranteed that $\mathbb{T} = \phi$. Therefore, the first condition is guaranteed to be satisfied eventually. After the condition $\mathbb{T} = \phi$, if $\mathbb{N}_S$ and $\mathbb{N}_D$ can communicate long enough, all the pages of $\mathbb{D}_M$ at $\mathbb{N}_S$ will be migrated and recoverable at $\mathbb{N}_D$. □

THEOREM 11. ***Starvation freedom:*** *Even after an arbitrary sequence of failures, there will be at least one node that can execute transactions on $\mathbb{D}_M$.*

The proof of Theorem 11 follows from Theorem 9 which ensures that $\mathbb{N}_S$ and $\mathbb{N}_D$ are in the same migration mode, and hence have a consistent view of $\mathbb{D}_M$'s ownership.

Theorem 10 and 11 together guarantee liveness. Zephyr guarantees safety in the presence of repeated failures or a network partition between $\mathbb{N}_S$ and $\mathbb{N}_D$, though progress is not guaranteed. Even though such failures are rare, proven guarantees in such scenarios improves the users' reliance on the system.

5. OPTIMIZATIONS AND EXTENSIONS

We now discuss some extensions that relax some of the assumptions made to simplify our initial description of Zephyr.

5.1 Replicated Tenants

In our discussion so far, we assume that the destination of migration does not have any prior information about $\mathbb{D}_M$. Many production database installations however use some form of replication for fault-tolerance and availability. In such a scenario, $\mathbb{D}_M$ can be migrated to a node which already has its replica. Since most DBMS implementations use lazy replication techniques to circumvent the high cost of synchronous replication [4], replicas often lag behind the master. Zephyr can be adapted to leverage this form of replication. Since $\mathbb{N}_D$ already has a replica, there is no need for the init mode. When $\mathbb{N}_S$ is notified to initiate migration, it executes the atomic handover protocol to enter the dual mode. Since $\mathbb{N}_D$'s copy of the database is potentially stale, when a transaction T_{Di} accesses a page P_i, similar to the original design, $\mathbb{N}_D$ synchronizes with $\mathbb{N}_S$ to transfer ownership. $\mathbb{N}_D$ sends the sequence number associated with its version of P_i to determine if it has the latest version of P_i; P_i is transferred only if $\mathbb{N}_D$'s version is stale. Furthermore, in the finish mode, $\mathbb{N}_S$ only needs to send a small number of pages that were not replicated to $\mathbb{N}_D$ due to a lag in replication. Replication can therefore considerably improve the performance of Zephyr.

5.2 Sharded Tenants

Our initial description assumes that a tenant is small and is served from a single node, i.e. a single partition tenant. However, Zephyr can also handle a large tenant that is sharded across multiple nodes, primarily due to the fact that $\mathbb{N}_S$ completes the execution of all transactions that were active when migration was initiated. Let $\mathbb{D}_M$ consist of partitions $\mathbb{D}_{M1}, \ldots, \mathbb{D}_{Mp}$ and assume that we are migrating $\mathbb{D}_{Mi}$ from $\mathbb{N}_S$ to $\mathbb{N}_D$. Transactions accessing only $\mathbb{D}_{Mi}$ are handled similar to the case of a single partition tenant. Let T_i be a multi-partition transaction where $\mathbb{D}_{Mi}$ is a participant. If T_i was active at the start of migration, then $\mathbb{N}_S$ is the node that executes T_i, and $\mathbb{D}_{Mi}$ will transition to finish mode only when all such T_i's have completed. On the other hand, if T_i started after $\mathbb{D}_{Mi}$ had transitioned to the dual mode, then $\mathbb{N}_D$ is the node executing T_i. At any given node, T_i is executed in the same way as in a small single partition tenant.

5.3 Data Sharing in Dual Mode

In Dual Mode, both $\mathbb{N}_S$ and $\mathbb{N}_D$ are executing update transactions on $\mathbb{D}_M$. This design is reminiscent of data sharing systems [7], the difference being that our design does not use a shared lock manager. However, our design can be augmented to use a shared lock manager to support a larger set of operations during migration, including arbitrary updates and minimizing transaction aborts at $\mathbb{N}_S$.

In the modified design, we replace the concept of page ownership with page level locking, allowing the locks to be shared when both $\mathbb{N}_S$ and $\mathbb{N}_D$ are reading a page. Every node in the system has a **Local Lock Manager (LLM)** and a **Global Lock Manager (GLM)**. The LLM is responsible for the local locking of pages while the GLM is responsible for arbitrating locks for remote pages. In all migration modes except dual mode, locks are local and hence serviced by the LLM. However, in the dual mode, $\mathbb{N}_S$ and $\mathbb{N}_D$ must synchronize through the GLMs. The only change needed is in the page ownership transfer, with the rest of the algorithm remains unchanged. Note that scalability limitations of a shared lock manager is not significant in our case since any instance of the lock manager is shared only between two nodes. We now describe how this extended design can remove some limitations of the original design. Details have been omitted due to space constraints.

In the original design of Zephyr, when a transaction T_{Di} requests access for a page P_i, $\mathbb{N}_D$ transfers ownership from $\mathbb{N}_S$. Therefore, future accesses to P_i (even reads) must fail to ensure serializable isolation. In this extended design, if $\mathbb{N}_D$ only needs a shared lock on P_i to service reads, then $\mathbb{N}_S$ can also continue processing reads from $T_{Sk+1}, \ldots, T_{Sl}$ that access P_i. Furthermore, even if $\mathbb{N}_D$ had acquired an exclusive lock, $\mathbb{N}_S$ can request a lock to $\mathbb{N}_D$'s GLM for the desired lock on P_i. This allows processing transactions at $\mathbb{N}_S$ that access a migrated page; the request to migrate the page back to $\mathbb{N}_S$ might be blocked in case it is locked at $\mathbb{N}_D$. The trade-off associated with this flexibility is the cost of additional synchronization between $\mathbb{N}_S$ and $\mathbb{N}_D$ to arbitrate shared locks, and the higher network overhead arising from the need to potentially copy P_i multiple times, while in the initial design, P_i was migrated exactly once. The original design made the index structure at both $\mathbb{N}_S$ and $\mathbb{N}_D$

immutable during migration and did not allow insertions or deletions that required a change in the index structure. The shared lock manager in the modified design circumvents this limitation by sharing locks at the index level as well, such that normal index traversal will use shared intention locks while an update to the index will acquire an exclusive lock on the index nodes being updated.

Zephyr, adapted to the data sharing architecture, allows more flexibility by allowing arbitrary updates and minimizing transactions or operations aborted due to migration. The implication on the correctness is straightforward. Since page ownership can be transferred back to $\mathbb{N}_S$, Lemma 3 does not hold any longer. However, Theorem 5 still holds since page level locking is done in a two phase manner using the shared lock managers, which ensures that a cycle in the conflict graph is impossible. The detailed proof is omitted for space limitations. Similarly, the proof for Lemma 1 has to be augmented with the case for index changes. However, since index changes will need the transaction inserting an item (T_2 in Lemma 1) to acquire an exclusive on the index page being modified, it will be blocked by the predicate lock acquired by the transaction with the predicate (T_1 in Lemma 1) on the index pages. Therefore, transactional correctness is still satisfied in the modified design; the other correctness arguments remain unchanged.

In summary, all these optimizations provide interesting trade-offs between minimizing the service disruption resulting from migration and the additional migration overhead manifested as higher network traffic and increased synchronization between $\mathbb{N}_S$ and $\mathbb{N}_D$. A detailed analysis and evaluation is left for future work.

6. IMPLEMENTATION DETAILS

Our prototype implementation of Zephyr extends an open source OLTP database H2 [18]. H2 is a lightweight relational database with a small footprint built entirely in Java supporting both embedded and server mode operation. Though primarily designed for embedded operation, one of the major applications of H2 is as a replacement of commercial RDBMS servers for development and testing. It supports a standard SQL/JDBC API, serializable and repeatable reads isolation levels [2], tree indices, and a relational data model with foreign keys and referential integrity constraints.

H2's architecture resembles the shared process multitenancy model where an H2 instance can have multiple independent databases with different schemas. Each database maintains its independent database cache, transaction manager, transaction log, and recovery manager. In H2, a database is stored as a file on disk which is internally organized as a collection of fixed size database pages. The first four pages store the database's metadata. The data definitions and user authentication information is stored as a metadata table (called `INFORMATION_SCHEMA`) which is part of the database. Every table in H2 is organized as a tree index. If a table is defined with a primary key which is of type integer or real number, then the primary key index stores data for the table. In case the primary key has other types (such as varchar) or if the primary key was not specified at table creation, the table's data are stored in a tree index whose key is auto-generated by the system. A table can have multiple indices which are maintained separate from the primary key index. The fourth page in the database file stores a pointer to the root of the `INFORMATION_SCHEMA` table, which in turn stores pointers to the other user tables. H2 supports classic multi-step transactions with serializable and read committed isolation level.

We use SQL Router[4], an open source package, to implement the query router. It is a JDBC wrapper that transparently migrates JDBC connections from $\mathbb{N}_S$ to $\mathbb{N}_D$. This SQL router runs a server listener that is notified when $\mathbb{D}_M$'s location changes. When migration is initiated, $\mathbb{N}_S$ spawns a migration thread T. In *init mode*, T transfers the database metadata pages, the entire `INFORMATION_SCHEMA` table of H2, and the internal nodes of the indices. Conceptually, this wireframe can be constructed by traversing the index trees to determine the internal index nodes. This however might incur a large number of random disk accesses for infrequently accessed parts of the index, which can considerably increase the migration overhead. We therefore use an optimization in the implementation where T sequentially scans through the database file and transfers only the internal nodes of the indices. When processing a database index page, it synchronizes with any concurrent transactions and obtains the latest version from the cache, if needed. Since the index structure is frozen during migration, this scan uses shared locking, allowing other update transactions to proceed. T notifies $\mathbb{N}_D$ of the number of pages skipped, which is used to update page ownership information at $\mathbb{N}_D$.

[4] `http://www.continuent.com/community/tungsten-sql-router`

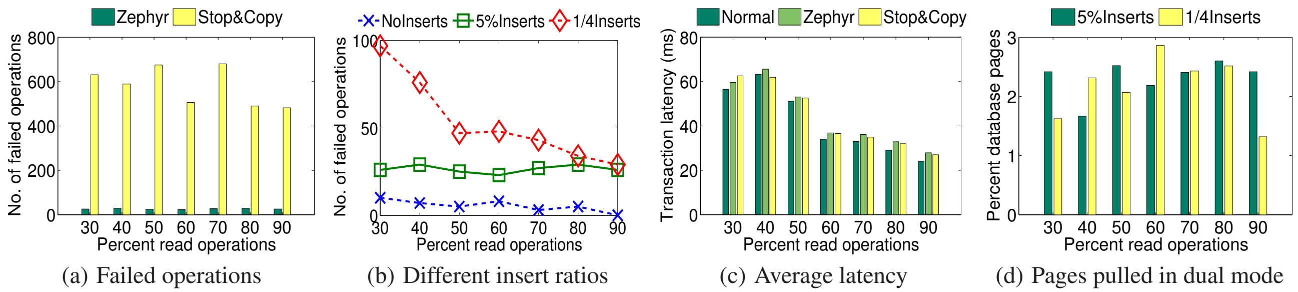
In the *dual mode*, $\mathbb{N}_D$ pulls pages from $\mathbb{N}_S$ on-demand while $\mathbb{N}_S$ continues transaction execution. Before a page is migrated, $\mathbb{N}_S$ obtains an exclusive lock on the page, updates the ownership mapping, and then sends it to $\mathbb{N}_D$. This ensures that the page is migrated only if it is not locked by any concurrent transaction. In the *finish mode*, $\mathbb{N}_S$ pushes all remaining pages that were not migrated in the dual mode, while serving any page fetch request from $\mathbb{N}_D$; pages transferred twice as a result of both the push from $\mathbb{N}_S$ and pull from $\mathbb{N}_D$ are detected at $\mathbb{N}_D$ and duplicate pages are rejected. Since $\mathbb{N}_S$ does not execute any transactions in finish mode, this push does not require any synchronization at $\mathbb{N}_S$.

7. EXPERIMENTAL EVALUATION

We now present a thorough experimental evaluation of Zephyr for live database migration using our prototype implementation. We compare Zephyr with the off-the-shelf **stop and copy** technique that stops the database at $\mathbb{N}_S$, flushes all changes, copies over the persistent image, and restarts the database at $\mathbb{N}_D$. Our evaluation uses two server nodes that run the database instances and a separate set of client machines that generate load on the database. Each server node has a 2.40GHz Intel Core 2 Quad processor, 8 GB RAM, a 7200 RPM SATA hard drive with 32MB Cache, and runs a 64-bit Ubuntu Server Edition with Java 1.6. The nodes are connected via a gigabit switch. Workload is generated from a different set of client machines. Since migration only involves $\mathbb{N}_S$ and $\mathbb{N}_D$, our evaluation focusses only on these two nodes and is oblivious of other nodes. We measure the migration cost as the number of failed operations, the amount of data transferred during migration, and the impact on transaction latency during and after migration.

7.1 Benchmark Description

We use the Yahoo! cloud serving benchmark (YCSB) [9] in our evaluation. YCSB emulates a synthetic workload generator that can be parameterized to vary the read/write ratio, access distributions, etc. Since the underlying database layer is multitenant, we run one benchmark instance for each tenant database. YCSB was originally designed to evaluate *Key-Value* stores, and hence primarily designed for single key operations or scans. We augmented this workload model and added multi-step transactions, where each transaction consists of multiple operations, the number of operations in a transaction (called **transaction size**) is another workload parameter. The number of read operations in a transaction is another parameter and so is the access distribution to select the rows accessed by a transaction. These parameters allow us to evaluate the behavior of the migration cost for different workloads and access patterns. We use the cost measures discussed in Section 2.3.



(a) Failed operations (b) Different insert ratios (c) Average latency (d) Pages pulled in dual mode

Figure 5: Impact of the distribution of reads, updates, and inserts on migration cost; default configurations used for rest of the parameters. We also vary the different insert ratios – 5% inserts correspond to a fixed percentage of inserts, while $1/4$ inserts correspond to a distribution where a fourth of the write operations are inserts. The benchmark executes 60,000 operations.

The workload emulates multiple user sessions where a user connects to a tenant's database, executes hundred transactions and then disconnects. A workload consists of sixty such sessions, i.e. a total of $6,000$ transactions. The default configurations use transactions with ten operations, 80% being read operations, 15% update operations and 5% new rows inserted. Each tenant's database consists of a single table with an integer primary key and ten columns of type varchar. Keys accessed by a transaction are chosen from a Zipfian distribution over a database with $100,000$ rows ($\sim$ 250 MB on disk); the Zipfian co-efficient is set to 1.0. The workload generator is multi-threaded with target throughput of 50 transactions per second (TPS). The default database page size is set to 16 KB and the cache size is set to 32 MB. These default configurations are representative of medium sized tenants [25, 26]. We vary these parameters, one at a time, to analyze their impact on migration cost.

7.2 Migration Cost

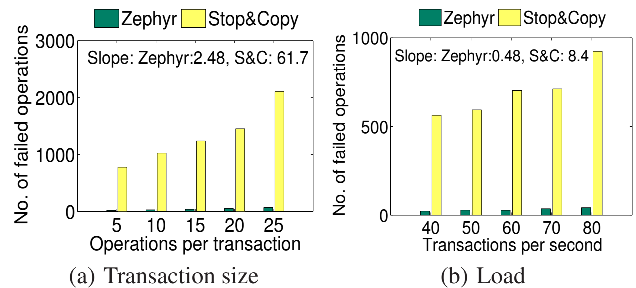
Our first experiment analyzes the impact on migration cost when varying the percentage read operations in a transaction. Figure 5(a) plots the number of failed operations during migration; clients continue issuing operations on the tenant even during migration. A client thread sequentially issues the operations of a transaction. All operations are well-formed, and any error reported by the database server after an operation has been issued account for a failed operation. As is evident from Figure 5(a), the number of failed operations in Zephyr is one to two orders of magnitude lesser when compared to stop and copy. Two reasons contribute to more failed operations in stop and copy: (i) abortion of all transactions active at the start of migration, and (ii) abortion of all new transactions that access the tenant when it is unavailable during migration. Zephyr does not incur any unavailability; operations fail only when they result in a change to the index structure during migration.

Figure 5(b) plots the number of failed operations when using Zephyr for workloads with different insert ratios. Zephyr results in only few tens of failed operations when the workload does not have a high percentage of inserts, even for cases with a high update proportion. As the workload becomes predominantly read-only, the probability of an operation resulting in a change in the index structure decreases. This results in a decrease in the number of failed operations in Zephyr. Stop and copy also results in fewer failed operations for higher values of read percentages, the reason being the smaller unavailability window resulting from fewer updates that need to be flushed before migration.

Figure 5(c) plots the average transaction latency as observed by a client during normal operation (i.e. when no migration is performed) and that with a migration occurring midway; the two bars correspond to the two migration techniques used. We report latency averaged over all the $6,000$ transactions that constitute the workload. We only report latency of committed transactions; aborted transactions are ignored. When compared to normal operation, the increased latency in stop and copy results from the cost of warming up the cache at $\mathbb{N}_D$ and the cost of clients re-establishing the database connections after migration. In addition to the aforementioned costs, Zephyr fetches pages from $\mathbb{N}_S$ on-demand during migration; the page can be fetched from $\mathbb{N}_S$'s cache or from its disk. This results in additional latency overhead in Zephyr when compared to stop and copy.

Figure 5(d) shows the percentage of database pages pulled during the dual mode of Zephyr. Since the dual mode runs for a very short period, only a small fraction of pages are pulled on demand.

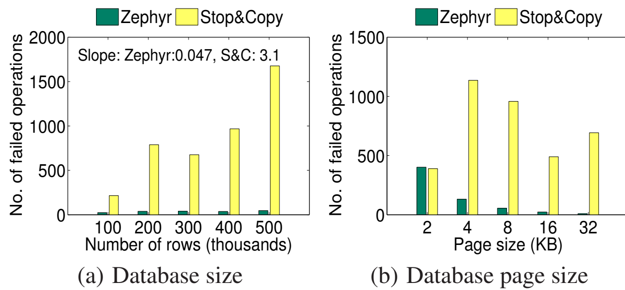
In our experiments, stop and copy took 3 to 8 seconds to migrate a tenant. Since all transactions in the workload have at least one update operation, when using stop and copy, all transactions issued during migration are aborted. On the other hand, even though Zephyr requires about 10 to 18 seconds to migrate the tenant, there is no downtime. As a result, the tenants observe few failed operations. Zephyr also incurs minimal messaging overhead beyond that needed to migrate the persistent database image. Every page transferred is preceded with its unique identifier; a pull request in the dual mode requires one round trip of messaging to fetch the page from $\mathbb{N}_S$. Stop and copy only requires the persistent image of the database to be migrated and does not incur any additional data transfer/messaging overhead.



(a) Transaction size (b) Load

Figure 6: Impact of varying the transaction size and load on number of failed transactions. We also report the slope of an approximate linear fit of the points in a series.

We now evaluate the impact of transaction sizes and load (see Figure 6). Varying the transaction size implies varying the number of operations in a transaction. Since the load is kept constant at 50 TPS, a higher number of operations per transaction implies more operations issued per unit time. Varying the load implies varying



(a) Database size (b) Database page size

Figure 7: Impact of the database page size and database size on number of failed operations.

the number of transactions issued. Therefore, higher load also implies more operations issued per unit time. Moreover, since the percentage of updates is kept constant, more operations result in more updates. For stop and copy, more updates result in more data to be flushed before migration. This results in a longer unavailability window which in turn results in more operations failing. On the other hand, for Zephyr, more updates imply a higher probability of changes to the index structure during migration, resulting in more failed operations. However, the rate of increase in failed operations is lower in Zephyr when compared to stop and copy. This is evident from the slope of an approximate linear fit of the data points in Figure 6; the linear fit for Zephyr has a considerably smaller slope than that for stop and copy. This shows that Zephyr is more robust to the use for a variety of workloads. The effect on transaction latency is similar and hence is omitted. We also varied the cache size allocated to the tenants, however, the impact of cache size on service interruption was not significant. Even though a large cache size will result in potentially more changes to be flushed to the disk, the Zipfian access distribution coupled with a high percentage of read operations result in very few changed objects in the cache.

Figure 7(a) plots the impact of the database size on failed operations. In this experiment, we increase the database size up to 500K rows in the database (about 1.3 GB). As the database size increases, more time is needed to copy the database's persistent image, resulting in a longer unavailability window for stop and copy. On the other hand, for Zephyr, a larger database implies a longer finish mode. However, since Zephyr does not result in any unavailability, the database size has almost no impact on the number of failed operations. This is again evident from the slope of the linear fit of the data points; the slope is considerably higher for stop and copy, while that of Zephyr is negligible. Therefore, Zephyr is more robust for larger databases when compared to stop and copy.

Figure 7(b) shows an interesting interplay of the database page size on the number of operations failing. As the database page size increases, the number of failed operations decreases considerably for Zephyr, while that of stop and copy remains almost un-affected. When the page size is small, each page could fit only a few rows. For instance, in our setting, each row is close to a kilobyte, and a 2K page is already full with two rows. As a result, a majority of inserts result in structural changes to the index, which result in a lot of these inserts failing during migration. If we consider the experiment with 2K page size, more than 95% of the failed operations were inserts. However, as the page size increases, the leaf pages have more unused capacity. Therefore, only a few inserts result in a change to the index structure. Since stop and copy is oblivious of the page size and transfers the raw bytes of the database file, its performance is almost unchanged as a result of a change in the page size. However, when the page size is increased beyond the block size of the underlying filesystem, reading a page from the disk becomes more expensive resulting in an increase in the transaction latency when the page size is larger than the file system block size.

In summary, Zephyr results in minimal service interruption. In a cloud platform, high availability is extremely critical for customer satisfaction, thus making Zephyr more attractive. In spite of Zephyr not allowing changes to the index structure during migration, it resulted in very few operations failing. A significant failure rate was observed only with a high row size to page size ratio. Zephyr is therefore more robust to variances in read-write ratios, database sizes, and transaction sizes when compared to stop and copy, thus making it suitable for a variety of workloads and applications.

8. RELATED WORK

Multitenant database systems have been used for many years in large enterprises. Traditionally, multitenancy in the database layer has been in the form of the shared table model with major proponents including Force.com [25], Google AppEngine [1] etc. A detailed analysis of the different models, associated trade-offs, and representative designs was presented in Section 2.1. *Key-Value* stores, such as Bigtable, PNUTS, Dynamo etc., have also been used for multitenant application deployment. They provide a unified namespace over a cluster of machines and multiple tenants store their databases in this shared namespace. The data store however manages data placement and co-location of tenants. Hence this sharing cannot be directly mapped to any of the common multitenancy models. On the other hand, a large number of database systems have been designed and proposed for the cloud [6, 10, 22], but the focus for most of them have been to scale-out using the cloud infrastructure.

Elasticity is a crucial factor to the success of a system in the cloud, and live migration is important for lightweight elasticity. The history of live migration can be traced back to more than two decades and relates to migration of live processes [27]. These process migration techniques have been extended in the context of virtual machine (VM) migration, and a number of techniques have been proposed for Live VM migration [8, 20], which is now a standard feature supported by most VMs. These techniques use a combination of stop-and-copy, pull, and push phases to migrate memory pages and process states, but rely on a shared persistent storage abstraction between the source and the destination. Bradford et al. [5] propose a technique to consistently migrate a VM image and persistent local state across a WAN. The authors use a technique based on disk checkpointing followed by checkpoint migration and copying over the differential changes. Our technique differs from the VM migration techniques in that we use the semantics of the DBMS such as database pages, locking during transaction execution, and proven database and distributed systems recovery protocols to migrate a live database while guaranteeing correctness in the presence of arbitrary failures.

Recently, techniques for live migration in the database layer have also been proposed. Most of these techniques draw inspiration from VM migration techniques and use database operation semantics. Das et al. [13] propose a technique, called *Iterative Copy* (IC), for a shared storage database architecture. Since the storage is shared, the authors focus on copying the cache during migration such that the destination starts with a hot cache. Our technique, on the other hand, focuses on the shared nothing architecture and involves transfer of the entire persistent image of the database. Our goal, therefore, is to minimize service interruption. In another closely related work, Curino et al. [10] also identify the need for live migration in a shared nothing database architecture. Our technique is similar in spirit to that of Curino et al. where they suggest starting execution

of transactions at the destination and the destination fetching pages on demand.

Lomet [21] proposes the use of replicated and partitioned indices for scaling out a database. Zephyr uses a similar concept where the index wireframe is replicated at the source and the destination nodes while page ownership is partitioned between them. However, Zephyr maintains the same structure of the index at both nodes while [21] allows the indices to have different structures while propagating updates asynchronously.

9. CONCLUSION

Live migration is an important feature to enable elasticity as a first class feature in multitenant databases for cloud platforms. We presented Zephyr, a technique to efficiently migrate a tenant's live database in a shared nothing architecture. Our technique uses a combination of on-demand pull and asynchronous push to migrate a tenant with minimal service interruption. Using light weight synchronization, we minimize the number of failed operations during migration, while also reducing the amount of data transferred during migration. We also provided a detailed analysis of the guarantees provided and proved the safety and liveness of Zephyr. Our technique relies on generic structures such as lock managers, standard B+ tree indices, and minimal changes to write ahead logging, thus making it suitable to be used in a variety of standard database engines with minimal changes to the existing code base. Our implementation in a standard lightweight open source RDBMS implementation shows that Zephyr allows lightweight migration of a live tenant database with minimal service interruption, thus allowing migration to be effectively used for elastic load balancing.

In the future, we plan to augment this technique with the control logic that determines which tenant to migrate and where to migrate. This control logic along with the ability to migrate live tenants, together form the basis for autonomous elasticity in multitenant databases for cloud platforms.

Acknowledgments

The authors would like to thank Shoji Nishimura, the anonymous reviewers, and our anonymous shepherd for providing useful feedback to improve this paper. This work is partly funded by NSF grants III 1018637 and CNS 1053594, and NEC Labs America.

10. REFERENCES

[1] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *CIDR*, pages 223–234, 2011.

[2] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, pages 1–10, 1995.

[3] P. A. Bernstein, I. Cseri, N. Dani, N. Ellis, A. Kalhan, G. Kakivaya, D. B. Lomet, R. Manner, L. Novik, and T. Talius. Adapting Microsoft SQL Server for Cloud Computing. In *ICDE*, 2011.

[4] P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann Publishers Inc., second edition, 2009.

[5] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *VEE*, pages 169–179, 2007.

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

[7] S. Chandrasekaran and R. Bamford. Shared cache - the future of parallel databases. In *ICDE*, pages 840–850, 2003.

[8] 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.

[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.

[10] C. Curino, E. Jones, R. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, and N. Zeldovich. Relational Cloud: A Database Service for the Cloud. In *CIDR*, pages 235–240, 2011.

[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.

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

[13] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi. Live Database Migration for Elasticity in a Multitenant Database for Cloud Platforms. Technical Report 2010-09, CS, UCSB, 2010.

[14] 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.

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

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

[17] J. N. Gray, R. A. Lorie, and G. R. Putzolu. Granularity of locks in a shared data base. In *VLDB*, pages 428–451, 1975.

[18] H2 Database Engine. `http://www.h2database.com/html/main.html`, 2011.

[19] D. Jacobs and S. Aulbach. Ruminations on multi-tenant databases. In *BTW*, pages 514–521, 2007.

[20] 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.

[21] D. B. Lomet. Replicated indexes for distributed data. In *PDIS*, pages 108–119. IEEE Computer Society, 1996.

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

[23] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM TODS*, 17(1):94–162, 1992.

[24] 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.

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

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

[27] E. Zayas. Attacking the process migration bottleneck. *SIGOPS Oper. Syst. Rev.*, 21(5):13–24, 1987.