ElasTraS: An Elastic, Scalable, and Self-Managing Transactional Database for the Cloud

SUDIPTO DAS, DIVYAKANT AGRAWAL, and AMR EL ABBADI, University of California Santa Barbara

A *database management system* (DBMS) serving a cloud platform must handle large numbers of application databases (or *tenants*) that are characterized by diverse schemas, varying footprints, and unpredictable load patterns. Scaling out using clusters of commodity servers and sharing resources among tenants (i.e., *multitenancy*) are important features of such systems. Moreover, when deployed on a pay-per-use infrastructure, minimizing the system's operating cost while ensuring good performance is also an important goal. Traditional DBMSs were not designed for such scenarios and hence do not possess the mentioned features critical for DBMSs in the cloud.

We present ElasTraS, which combines three design principles to build an elastically-scalable multitenant DBMS for transaction processing workloads. These design principles are gleaned from a careful analysis of the years of research in building scalable key-value stores and decades of research in high performance transaction processing systems. ElasTraS scales to thousands of tenants, effectively consolidates tenants with small footprints while scaling-out large tenants across multiple servers in a cluster. ElasTraS also supports *low-latency multistep ACID transactions*, is fault-tolerant, self-managing, and highly available to support mission critical applications. ElasTraS leverages Albatross, a low overhead on-demand *live database migration* technique, for elastic load balancing by adding more servers during high load and consolidating to fewer servers during usage troughs. This elastic scaling minimizes the operating cost and ensures good performance even in the presence of unpredictable changes to the workload.

We elucidate the design principles, explain the architecture, describe a prototype implementation, present the detailed design and implementation of Albatross, and experimentally evaluate the implementation using a variety of transaction processing workloads. On a cluster of 20 commodity servers, our prototype serves thousands of tenants and serves more than 1 billion transactions per day while migrating tenant databases with minimal overhead to allow lightweight elastic scaling. Using a cluster of 30 commodity servers, ElasTraS can scale-out a terabyte TPC-C database serving an aggregate throughput of approximately one quarter of a million TPC-C transactions per minute.

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

General Terms: Algorithms, Design, Performance.

Additional Key Words and Phrases: Cloud computing, elastic data management, transactions, ACID, scalability, fault-tolerance

5

This work is partially supported by NSF grants III 1018637 and CNS 1053594, NEC Labs America university relations award, and an Amazon AWS in Education grant. Any opinions, findings, and conclusions expressed in this article are those of the authors and do not necessarily reflect the views of the sponsors.
Author's addresses: S. Das, Microsoft Research, One Microsoft Way, Redmond, WA 98052 (corresponding author); email: sudiptod@microsoft.com; D. Agrawal and A. El Abbadi, Department of Computer Science, UC Santa Barbara, Santa Barbara, CA 93106-5110.
Permission to make digital or hard copies of part or all 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 show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2013 ACM 0362-5915/2013/04-ART5 $15.00
DOI: http://dx.doi.org/10.1145/2445583.2445588

ACM Reference Format:
Das, S., Agrawal, D., El Abadi, A. 2013. ElasTraS: An elastic, scalable, and self-managing transactional database for the cloud. ACM Trans. Datab. Syst. 38, 1, Article 5 (April 2013), 45 pages.
DOI: http://dx.doi.org/10.1145/2445583.2445588

1. INTRODUCTION

Cloud platforms serve a diverse class of data-driven applications that result in databases (or *tenants*) with very different schemas, workload types, data access patterns, and resource requirements. Moreover, many of these tenants start small and only a handful of them experience viral growth resulting in unpredictable changes in load; others observe sporadic loads followed by decreasing popularity [Yang et al. 2009; Jacobs and Aulbach 2007; von Eicken 2008]. A *database management system* (DBMS) serving the cloud application platforms must, therefore, efficiently consolidate these small tenants and share resources among them to ensure effective resource utilization: a model referred to as *multitenancy*. Consolidation in a multitenant system leverages the fact that peak resource usage is not frequent and peaks for different tenants are often temporally separated. While consolidating tenants to minimize the operating costs, a multitenant DBMS must also ensure good performance for the tenants.[1] Since tenants' requirements are independent, a multitenant DBMS must strike the right balance between resource sharing and good performance for tenants.

When such a multitenant DBMS is deployed on a pay-per-use cloud infrastructure, an added goal is to optimize the system's operating cost. *Elasticity*, that is, the ability to deal with load variations in a live system by adding more servers during high load and consolidating to fewer servers when the load decreases, is therefore critical for these systems. Most existing DBMSs do not consider elasticity as a first class feature and are performance centric. Elasticity is paramount to minimizing the operating cost of a system deployed on a pay-per-use cloud infrastructure. To ensure low impact on a tenant's performance, *elastic scaling* and *load balancing* must be *lightweight* with minimal performance overhead. Furthermore, managing and administering resources among large numbers of tenant databases with various requirements is hard and labor intensive. *Self-manageability* is, therefore, also important to minimize the system's operating cost by minimizing the need for human intervention in system management. Detecting failures and recovering from them, tenant placement for effective consolidation, and elastic load balancing are important responsibilities of such a self-managing system controller. Adding the requirements of scalability, fault-tolerance, and high availability common to any mission critical system, a multitenant DBMS for cloud platforms faces a unique set of challenges.

If we consider the spectrum of DBMSs, on one hand we have traditional *relational databases* (RDBMSs) that provide low latency and high throughput transaction processing at a single node, but lack features such as elasticity and scale-out. Some recently proposed "cloud enabled" relational databases, such as Cloud SQL Server [Bernstein et al. 2011a] and Relational Cloud [Curino et al. 2011], target scale-out but do not consider lightweight elastic scaling as a first class feature. On the other hand, key-value stores, such as Bigtable [Chang et al. 2006] PNUTS [Cooper et al. 2008], and Dynamo [DeCandia et al. 2007], are scalable, elastic, and highly available, but only support simple functionality such as key-based lookups and single-row transactions.

[1]Note that ideally a multitenant system should support some form of service level agreement (SLA) for performance. However, defining such a performance SLA is beyond the scope of this article. We assume that the tenant and provider agree upon an average response time that the provider tries to meet and uses as a trigger for elastic scaling and load balancing.

Current DBMSs, therefore, do not possess all the features of efficient scale-out transaction processing, lightweight elasticity, and effective multitenancy that are critical to the success of a DBMS for a cloud platform.

We present *ElasTraS*, an elastically scalable transaction processing DBMS for multitenant cloud platforms. ElasTraS embodies a combination of three design principles that provide effective resource sharing among tenants while ensuring good performance, low latency transaction processing, and low overhead live database migration. The key observation driving the design of ElasTraS is that even though classical RDBMSs and key-value stores form two different classes of data management systems that are architecturally different and target different types of applications, many of their design principles can be combined into a single system. ElasTraS is a culmination of two major design philosophies—traditional RDBMSs for efficient transaction execution on small databases and the key-value stores for scale, elasticity, and high availability—augmented with some novel features to efficiently and natively support multitenancy. ElasTraS differs from the key-value stores by supporting relational schemas and efficient nondistributed multistep transactions as a first class feature. It differs from classical RDBMSs by scaling out the system to a cluster of loosely coupled set of servers. Furthermore, ElasTraS differs from both classes of systems by supporting elasticity and lightweight live data migration in a multitenant DBMS targeting OLTP workloads. ElasTraS targets serving thousands of small tenants and processing billions of transactions per day while also handling tenants that grow *big*. We envision a typical ElasTraS installation as a cluster of a few hundreds of nodes co-located within a datacenter.

At the microscopic scale, ElasTraS consolidates multiple tenants within the same database process (*shared process* multitenancy [Jacobs and Aulbach 2007]) allowing effective resource sharing among small tenants. ElasTraS efficiently supports transactions while scaling-out to clusters of servers by limiting transactional accesses to a single node, thus obviating the need for distributed transactions. For large tenants that span multiple nodes, ElasTraS requires tenants to use a hierarchical schema pattern, such as the *tree schema*, and restricts a transaction to only access data items that refer to a single primary key at the root of the hierarchy. This limitation on transaction accesses obviates the need for distributed synchronization for transaction execution, a key requirement for high scalability and availability. Note that even though the current prototype supports only the tree schema, in principle, ElasTraS can support any schema that is amenable to partitioning such that the majority of transactions only access a single partition. For tenants small enough to be served from a single node, ElasTraS does not impose *any* restrictions on schema or transaction accesses, since all transactions are guaranteed to be nondistributed. However, designing applications based on the tree schema allows a tenant to scale-out across multiple nodes when its size and load grows and does not require rewriting the application or redesigning the database schema and the transactions.

Another important architectural feature of ElasTraS is that it uses *logically* separate clusters of nodes for transaction processing and for data storage. The storage layer in ElasTraS is abstracted as a scalable and fault-tolerant network-addressable storage. The transaction processing layer, executing on a cluster of nodes logically separate from the storage layer, focuses on efficient caching, transaction execution, and recovery of the partitions. Every transaction manager node has exclusive access to the data it is serving. ElasTraS, therefore, has a *decoupled storage* architecture, which is different from the classical *shared storage* architectures where the storage layer is used for transaction synchronization. By decoupled storage, we refer to an architecture (such as Bigtable) where persistent data is stored as a layer separate from the logic that executes operations on that data.

For tenants with sporadic changes in loads, ElasTraS leverages low-cost *live database migration* for elastic scaling and load balancing. We propose *Albatross*, a lightweight live database migration technique for decoupled storage architectures. In decoupled storage architectures, the persistent data does not need migration. However, to minimize the impact on transaction latency, Albatross focuses on migrating the database cache and the state of active transactions, which allows the tenant's database to start "warm" at the destination. To minimize the unavailability window during migration, this copying of the state is performed iteratively while the source continues to serve transactions on the partition being migrated. Copying the state of transactions active during migration allows them to resume execution at the destination. The lightweight nature of Albatross allows ElasTraS to aggressively consolidate tenants to a small set of servers while still being able to scale-out on-demand to meet tenant performance requirements in the event of unpredictable load bursts. Note that while many key-value stores or even classical RDBMSs might allow migrating data fragments, the known migration techniques either result in a period of unavailability, or even if they support live migration, little is publicly known about the techniques used or how they can be made efficient.

At the macroscopic scale, ElasTraS uses loose synchronization between the nodes for coordinating operations, fault-detection and recovery techniques that ensure safety during failures, and models that automate load balancing and elasticity. ElasTraS advances the state-of-the-art by *presenting a combination of multitenancy and elasticity in a single database system, while being self-managing, scalable, fault-tolerant, and highly available.*

We present a prototype implementation of ElasTraS with Albatross and demonstrate its effectiveness using two OLTP benchmarks: the industry standard TPC-C benchmark [Transaction Processing Performance Council 2009] and the Yahoo! cloud serving benchmark (YCSB) [Cooper et al. 2010] extended to add transactional support. We demonstrate that a 20 node ElasTraS cluster can serve thousands of small tenants processing more than 1 billion transactions a day. We demonstrate ElasTraS's ability to scale-out large tenants by serving a *large* partitioned TPC-C tenant with more than 1 TB data and spanning a 30 node cluster with an aggregate throughput of about a quarter of a million TPC-C transactions per minute (tpmC). We also demonstrate the effectiveness of Albatross for lightweight migration. Albatross can migrate individual tenant databases with *no aborted transactions*, negligible impact on transaction latency and throughput both during and after migration, and an unavailability window as low as 300 ms. We further demonstrate that ElasTraS can leverage Albatross for elastic scaling and consolidation in periods of high load as well as during usage troughs. Overall, our experiments demonstrate that ElasTraS can scale-out to clusters of commodity servers, support efficient low latency transaction execution, allow effective consolidation of small tenants, and perform lightweight elastic scaling, thus possessing the features critical to a multitenant DBMS for cloud platforms.

This article makes several contributions towards building a scalable and elastic OLTP DBMS for cloud platforms. In Das et al. [2009], we presented a brief preliminary overview of a preliminary version of ElasTraS's architecture and in Das et al. [2011], we presented the design and implementation of Albatross. This article presents the first detailed end-to-end design and implementation description of ElasTraS with a thorough experimental evaluation and an analysis of its correctness guarantees. Major contributions made by this article are as follows.

—We present the detailed design of ElasTraS and articulate its design principles that are important for lightweight elasticity, effective multitenancy, and large scale operation.

—We present a detailed prototype implementation for the design that explains the three critical facets of the design: transaction management, elastic load balancing, and serving large tenants.
—We present Albatross, a lightweight technique for live database migration. We develop the detailed design and its implementation in ElasTraS. Albatross is the first end-to-end solution for live database migration in decoupled storage architectures.
—We provide a detailed analysis of the guarantees provided by ElasTraS and Albatross during normal operation as well as during failures.
—We present a thorough evaluation of ElasTraS using a variety of transactional workloads, a detailed analysis of the results to understand the implications of some critical aspects of ElasTraS's design, and provide a discussion of the lessons learned.

Organization. Section 2 surveys related literature in scale-out transaction processing and multitenant DBMS designs. Section 3 articulates the design principles that form the cornerstone for ElasTraS's design. Section 4 presents the detailed design and rationalizes the design choices. Section 5 presents a prototype implementation. Section 6 presents the design and the detailed implementation of Albatross in ElasTraS. Section 7 evaluates ElasTraS's prototype using two standard OLTP benchmarks and analyzes the lessons learned. Section 8 concludes the article. The Appendices provide a detailed analysis of the correctness guarantees.

2. RELATED WORK

With the growing need for scalable data management systems, a number of systems have been proposed for managing large amounts of data. Key-value stores were designed with the target of large-scale operations—scaling to thousands of machines, spanning geographical regions, and serving hundreds of thousands of concurrent requests. Examples of such systems include Bigtable [Chang et al. 2006], PNUTS [Cooper et al. 2008], Dynamo [DeCandia et al. 2007], and their open source counterparts. These systems were designed primarily for high scalability and high availability and have demonstrated some interesting design alternatives for scaling large data management systems. However, in order to support large-scale operations and high availability in the presence of failures, these systems support simple key-based access where consistency guarantees are supported only on single rows. These systems do not natively support multistep transactions, secondary key accesses, and joins. ElasTraS demonstrates how multistep transactions and efficient multitenancy can be married with some of the design aspects of key-value stores to build a system that supports a richer set of operations while still allowing the system to scale out to large numbers of nodes.

The need for advanced functionality, such as transactional access on multiple rows, grew as more applications started using these key-value stores. This led to designs such as Megastore [Baker et al. 2011], Percolator [Peng and Dabek 2010], ecStore [Vo et al. 2010], and G-Store [Das et al. 2010] that build a layer to provide transactional access to a key-value store substrate. Megastore supports transactions on a static collection of keys, an *entity group*, while supporting cross-group transactions using queues. Megastore supports a transactional interface and entity groups as a layer on top of Bigtable while providing cross-datacenter replication for high availability. Megastore's design goal is to synchronously replicate updates to an entity group across multiple datacenters for high availability and disaster recovery. Little attention is given to efficient transaction execution within an entity group or in efficiently sharing resources among entity groups co-located on the same server. For instance, Megastore executes transactions serially on an entity group [Patterson et al. 2012], even though serializability allows concurrent execution, thus limiting the peak throughput of transactions on an entity group. ElasTraS, on the other hand, focuses on efficient

and concurrent transaction execution within a database partition, thus resulting in performance characteristics similar to RDBMSs for single-partition transactions. ElasTraS also addresses the problem of efficiently handling multitenancy and elasticity. However, unlike Megastore, ElasTraS does not target cross-datacenter replication; data replication in ElasTraS is only within a datacenter.

G-Store supports efficient multistep transactions on a dynamic collection of keys, a *key group*, while not supporting cross-group transactions. Both ElasTraS and G-Store focus on limiting distributed transactions. Percolator, on the other hand, allows flexible (and distributed) transactions, but targets the batched execution model where low transaction latency is not critical. Similar to Percolator, ecStore also uses distributed transactions to provide weak isolation guarantees and hence is expected to have high transaction execution latencies and reduced availability in the presence of failures. G-Store, ecStore, and Percolator do not target multitenancy. Similar to these systems, ElasTraS supports efficient scale-out transaction processing in addition to lightweight elastic scaling and native support for multitenancy.

Another class of systems include relational databases for the cloud, many of which were developed concurrently with ElasTraS. Such systems can be broadly divided into shared nothing architectures and decoupled storage architectures. The first category includes systems such as Microsoft Cloud SQL Server powering SQL Azure [Bernstein et al. 2011a], Relational Cloud [Curino et al. 2011], and a prototype from Yahoo! [Yang et al. 2009]. The goal of these systems is to adapt standard RDBMSs (such as SQL Server or MySQL) by using a shared-nothing database cluster. These systems use locally-attached disks for storage, replicate the databases for fault-tolerance, and rely on database partitioning for scale-out. On the other hand are systems with decoupled shared storage, which include Project Deuteronomy [Lomet et al. 2009], a database on S3 [Brantner et al. 2008], and Hyder [Bernstein et al. 2011b]. These systems leverage the rich literature [Weikum and Vossen 2001; Gray and Reuter 1992] to build transaction processing and advanced data processing capabilities on top of the storage layer. Irrespective of the architecture, these systems focus primarily on scalability. The use of elasticity to optimize a system's operating cost has remained unexplored.

Sinfonia [Aguilera et al. 2007] proposes a design that spans the middle ground between key-value access and transactional support by proposing the minitransaction primitive. The key idea is to piggyback some operations in the first phase of the two-phase commit (2PC) protocol [Gray 1978] to guarantee atomicity of operations in a distributed set of nodes. While the supported set of operations is limited to six, the system demonstrates how distributed synchronization, when used thoughtfully, can be used to build scalable systems. ElasTraS follows a similar principle by limiting distributed synchronization only for system management operations but differs by exploiting properties of the database schema to provide a richer transaction abstraction for accesses to a single node.

Even though little work has focused on elasticity and 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) [Clark et al. 2005; Liu et al. 2009]. The technique proposed by Clark et al. [2005] is conceptually similar to Albatross. The major difference is that Clark et al. use VM level memory page copying and OS and networking primitives to transfer live processes and network connections. On the other hand, Albatross leverages the semantics of a DBMS and its internal structures to migrate the cached database pages (analogous to VM pages), state of active transactions (analogous to active processes), and database connections (analogous to network connections). This allows Albatross to be used for live database migration in the shared process multitenancy model where live VM migration cannot be used.Liu et al. [2009] proposed an improvement over [Clark et al. 2005] to reduce

the downtime and the amount of data synchronized by using a log based copying approach. Albatross tracks changes to the database state, however, we do not use explicit log shipping for synchronization. Bradford et al. [2007] proposed a technique to consistently migrate a VM across a wide area network. Database migration in other contexts—such as migrating data as the database schema evolves, or between different versions of the database system—has also been studied; Sockut and Iyer [2009] provide a detailed survey of the different approaches. Our focus is migration for elastic scaling.

Elmore et al. [2011] proposed Zephyr, a live database migration technique for shared nothing database architectures such as Cloud SQL Server or Relational Cloud. In a shared nothing architecture, the persistent data is stored in disks locally attached to every node. Hence, the persistent data must also be migrated. Zephyr, therefore, focuses on minimizing the service interruption and uses a synchronized *dual mode* where the source and destination nodes are both executing transactions. However, since ElasTraS is a decoupled storage architecture where persistent data is not migrated, Albatross results in even lesser impact during migration. Albatross not only minimizes service interruption but also minimizes the impact on transaction latency by copying the database cache during migration. Therefore, unlike in Zephyr, the destination node starts warm in Albatross, thus resulting in lesser performance impact.

Much research has also focused on multitenancy in databases. Different multitenancy models have been proposed in the literature [Jacobs and Aulbach 2007]; the predominant models are *shared table* [Weissman and Bobrowski 2009; Baker et al. 2011], *shared process* [Bernstein et al. 2011a; Curino et al. 2011], and *shared machine*.

The shared table model co-locates different tenants' data in a set of shared tables for effective resource sharing when a majority of tenants have similar schema or minor variants of a common schema. However, this model provides weak isolation among tenants and requires custom query and transaction processing engines since the data is stored as raw bytes while type information is inferred from additional metadata.

The shared machine model is the other extreme with the strongest isolation where the tenants only share the physical servers while having their independent database processes and virtual machines (VM). In this model, live VM migration [Clark et al. 2005] techniques can be used for elastic scaling. However, as demonstrated in a recent study [Curino et al. 2011], the shared machine model introduces a high overhead resulting in inefficient resource sharing.

The shared process model strikes the middle ground by colocating multiple tenants within the same database process and allocating independent tables for tenants. This design supports more schema flexibility compared to the shared table model and allows using standard query and transaction processing techniques, while allowing consolidation of more tenants at the same node [Curino et al. 2011]. However, new live database migration techniques must be developed for elastic scalability in this model.

In summary, in spite of a rich literature in scalable transaction processing, no single system in the literature supports elasticity, low latency ACID transactions, large scale operation, efficient resource sharing amongst tenants, fault-tolerance, and high availability. In a brief and preliminary overview of ElasTraS's design [Das et al. 2009], we proposed a high-level design of a system that serves both OLTP and data analysis workloads within a single system. This article expands and extends the early overview by developing the detailed system design and implementation, a thorough experimental analysis using a wide variety of OLTP workloads, and a detailed analysis of the correctness guarantees. We developed Albatross [Das et al. 2011] as a stand-alone live database migration technique for decoupled storage architectures. In this article, we incorporate Albatross into ElasTraS to present a complete end-to-end elastic DBMS implementation targeting multitenant cloud platforms. To the best of our knowledge, this is the first work describing the detailed design, implementation, and a thorough

evaluation of a complete end-to-end system that supports scale-out transaction processing, elasticity, and multitenancy as first class DBMS features.

3. DESIGN PRINCIPLES

Even though RDBMSs and key-value stores are very different in their origins and architectures, a careful analysis of their design allows us to distill some design principles that can be carried over in designing database systems for cloud platforms. These principles form the cornerstone for the design of ElasTraS. Before delving into ElasTraS's design, we first articulate these design principles.

—*Separate system state from application state.* Abstractly, a distributed database system can be modeled as a composition of two different states: the *system state* and the *application state*. The system state is the meta data critical for the system's proper operation. Examples of system state are: node membership information in a distributed system, mapping of database partitions to the nodes serving the partition, and the location of the primary and secondary replicas in a system. This state requires stringent consistency guarantees, fault-tolerance, and high availability to ensure the proper functioning of the system in the presence of different types of failures. However, scalability is not a primary requirement for the system state since it is typically small and not frequently updated. On the other hand, the application state is the application's data that the system stores. Consistency, scalability, and availability of the application state is dependent on the application's requirements. Different systems provide varying trade-offs among the different guarantees provided for application state.
A clean separation between the two states is a common design pattern observed in different key-value stores, such as Bigtable and PNUTS, and allows the use of different protocols, with different guarantees and associated costs, to maintain the two types of states. For instance, the system state can be made highly available and fault-tolerant by synchronously replicating this state using distributed consensus protocols such as Paxos [Lamport 1998], while the application state might be maintained using less stringent protocols. This functional separation is a common feature in key-value stores that contributes to their scalability.

—*Decouple data storage from ownership. Ownership* refers to the read/write access rights to data items. Separating (or *decoupling*) the data ownership and transaction processing logic from that of data storage has multiple benefits: (i) it results in a simplified design allowing the storage layer to focus on fault-tolerance while the ownership layer can guarantee higher level guarantees such as transactional access without worrying about the need for replication; (ii) depending on the application's requirements it allows independent scaling of the ownership layer and the data storage layer; and (iii) it allows for lightweight control migration for elastic scaling and load balancing, it is enough to safely migrate only the ownership without the need to migrate data.

—*Limit transactions to a single node.* Limiting the frequently executed transactions to a single node allows efficient execution without the need for distributed synchronization. Additionally, it allows the system to horizontally partition and scale-out. It also limits the effect of a failure to only the data served by the failed component and does not affect the operation of the remaining components, thus allowing graceful performance degradation in the event of failures. As a rule of thumb, operations manipulating the application state must be limited to a single node in the database tier. Once transaction execution is limited to a single node, techniques from the rich literature on efficient transaction execution and performance optimization can be potentially applied. Distributed synchronization is used in a prudent manner

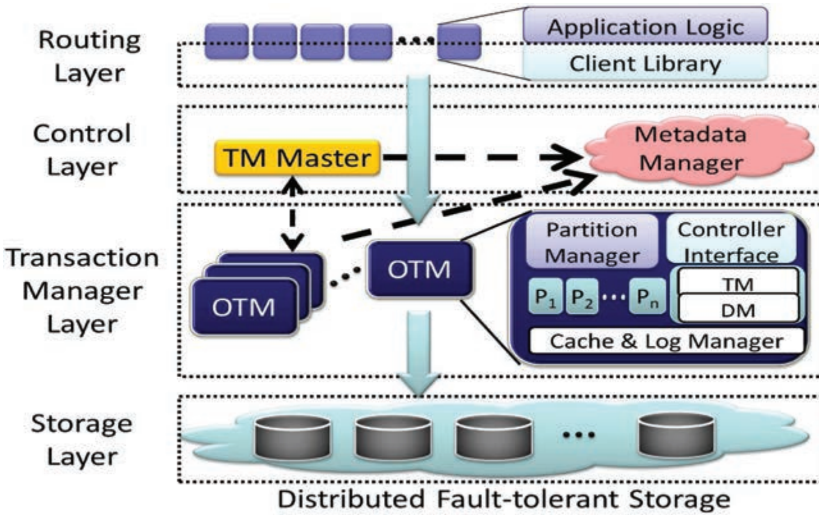


Fig. 1. ElasTraS architecture. Each box represents a server, a cloud represents a service abstraction, dashed arrows represent control flow, block arrows represent data flow, and dotted blocks represent conceptual layers in the system.

and is used only when needed. An example use of distributed synchronization is for updating the system state.

4. THE DESIGN OF ELASTRAS

4.1. ElasTraS's Architecture

We explain ElasTraS's architecture in terms of the four layers shown in Figure 1 from the bottom up: the storage layer, the transaction management layer, the control layer, and the routing layer. Recall that ElasTraS views the database as a collection of partitions. For small tenants, its entire database is contained in a partition while large tenant databases span multiple partitions. The partitions form the granule of assignment, load balancing, and transactional access.

4.1.1. The Distributed Fault-tolerant Storage Layer. The storage layer, or the *Distributed Fault-Tolerant Storage* (DFS), is a network-addressable storage abstraction that stores the persistent data. This layer is a fault-tolerant replicated storage manager that guarantees durable writes and strong replica consistency while ensuring high data availability in the presence of failures. Such storage abstractions are common in current datacenters in the form of scalable distributed file systems (such as the Hadoop distributed file system [HDFS 2010]), or custom solutions (such as Amazon elastic block storage or the storage layer of Hyder [Bernstein et al. 2011b]). High throughput and low latency datacenter networks provide low cost reads from the storage layer; however, strong replica consistency make writes expensive. ElasTraS minimizes the number of DFS accesses to reduce network communication and improve the overall system performance. We use a multiversion append-only storage layout that supports more concurrency for reads and considerably simplifies live migration for elastic scaling. This choice of a decoupled storage architecture differentiates ElasTraS from the classical RDBMSs where the storage is either locally attached or used for data sharing (such as in Oracle RAC [Chandrasekaran and Bamford 2003] and Hyder [Bernstein et al. 2011b]).

4.1.2. Transaction Management Layer. This layer consists of a cluster of servers called *Owning Transaction Managers* (OTM). An OTM is analogous to the transaction manager in a classical RDBMS. Each OTM serves tens to hundreds of partitions for which it has unique *ownership*, that is, exclusive read/write access to these partitions. The number of partitions an OTM serves depends on the overall load. The exclusive ownership of a partition allows an OTM to cache the contents of a partition without violating data consistency while limiting transaction execution within a single OTM and allowing optimizations such as *fast commit* [Weikum and Vossen 2001]. Each partition has its own *transaction manager (*TM*)* and shared *data manager (*DM*)*. All partitions share the OTM's log manager, which maintains the transactions' commit log. This sharing of the log minimizes the number of competing accesses to the shared storage while allowing further optimizations such as group commit [Weikum and Vossen 2001; Bernstein and Newcomer 2009]. To allow fast recovery from OTM failures and to guarantee high availability, an OTM's commit log is stored in the DFS. This allows an OTM's state to be recovered even if it fails completely. The use of an OTM differentiates ElasTraS from the classical key-value stores by allowing efficient execution of nondistributed multioperation transactions as a first-class feature.

4.1.3. Control Layer. This layer consists of two components: the *TM Master* and the *Metadata Manager* (MM). The TM Master, a stateless process, monitors the status of the OTMs and maintains overall system load and usage statistics for performance modeling. The TM Master is responsible for assigning partitions to OTMs, detecting and recovering from OTM failures, and controlling elastic load balancing. On the other hand, the MM is responsible for maintaining the system state to ensure correct operation. This metadata consists of *leases*, granted to every OTM and the TM Master, *watches*, a mechanism to notify changes to a lease's state, and (a pointer to) the *system catalog*, an authoritative mapping of a partition to the OTM currently serving the partition. Leases are uniquely granted to a server for a fixed time period and must be periodically renewed. Since the control layer stores only metadata and performs system maintenance, it is not in the data path for the clients. The state of the MM is critical for ElasTraS's operation and is replicated for high availability.

4.1.4. Routing Layer. ElasTraS dynamically assigns partitions to OTMs. Moreover, for elastic load balancing, a database partition can be migrated on-demand in a live system. The routing layer, the *ElasTraS client library* which the applications link to, hides the logic of connection management and routing, and abstracts the system's dynamics from the application clients while maintaining uninterrupted connections to the tenant databases.

4.2. Design Rationale

Before we present the details of ElasTraS's implementation, we explain the rationale that has driven this design. We relate these design choices to the principles outlined in Section 3. Many of the design choices in ElasTraS are driven from a careful analysis of the design principles that resulted in scalability of key-value stores and efficient transaction processing in classical RDBMSs.

ElasTraS uses a decoupled storage architecture where the persistent data is stored on a cluster of servers that is logically separate from the servers executing the transactions. This architecture distinguishes ElasTraS from systems such as Cloud SQL Server and Relational Cloud that are shared-nothing architectures where the persistent data is stored in disks locally attached to the transaction managers. The rationale for decoupling the storage layer is that for high performance OLTP workloads, the active working set must fit in the cache or else the peak performance is limited by the disk I/O. OLTP workloads, therefore, result in infrequent disk accesses. Hence, decoupling

the storage from transaction execution will result in negligible performance impact since the storage layer is accessed infrequently. Moreover, since the TM layer is connected to the storage layer via low latency and high throughput datacenter networks, remote data access is fast. Furthermore, this decoupling of the storage layer allows independent scaling of the storage and transaction management layers, simplifies the design of the transaction managers by abstracting fault-tolerance in the storage layer, and allows lightweight live database migration without migrating the persistent data. ElasTraS's use of decoupled storage is, however, different from traditional shared-disk systems [Weikum and Vossen 2001], Hyder, or Megastore. Shared-disk systems and Hyder use the shared storage for synchronizing transactions executing on different servers while Megastore uses the log stored in the shared Bigtable layer to serialize transactions within an entity group. Each server in ElasTraS is allocated storage that does not overlap with any other server and the storage layer is not used for any form of synchronization.

Colocating a tenant's data into a single partition, serving a partition within a single database process, and limiting transactions to a partition allows ElasTraS to limit transactions to a single node, thus limiting the most frequent operations to a single node. This allows efficient transaction processing and ensures high availability during failures. For small tenants contained in a single partition, ElasTraS supports flexible multistep transactions. The size and peak loads of a single-partition tenant is limited by the capacity of a single server.

ElasTraS separates the application state, stored in the storage layer and served by the transaction management layer, from the system state, managed by the control layer. ElasTraS replicates the system state, using protocols guaranteeing strong replica consistency, to ensure high availability. The clean separation between the two types of states allows ElasTraS to limit distributed synchronization to only the critical system state while continuing to use nondistributed transactions to manipulate application state.

ElasTraS's use of the shared process multitenancy model is in contrast to the shared table model used by systems such as Salesforce.com [Weissman and Bobrowski 2009] and Megastore. In addition to providing effective consolidation (scaling to large numbers of small tenants) and low cost live migration (elastic scaling), the shared process model provides good isolation between tenants and provides for more flexible tenant schema.

5. IMPLEMENTATION DETAILS

In this section, we present a detailed prototype implementation of ElasTraS that conceptually stores and serves a set of partitions. Table I summarizes the major aspects in the design and implementation of ElasTraS.

5.1. Storage Layer

Many existing solutions provide the guarantees required by the ElasTraS storage layer; in our prototype, we use the Hadoop Distributed File System (HDFS) [HDFS 2010]. Our use of HDFS is primarily driven by the fact that it is one of the more stable freely available distributed file system and avoids reliance on any custom commercial network attached storage solution. However, in principle, any network filesystem or storage system can be used.

HDFS is a replicated append-only file system providing low-cost block-level replication with strong replica consistency when only one process is appending to a file. ElasTraS ensures that at any point in time, only a single OTM appends to an HDFS file. A write is acknowledged only when it has been replicated in memory at a

Table I. Summary of ElasTraS's Design and Implementation

Problem	Technique	Advantages
System state maintenance	Metadata manager with Paxos-based replicated state machine	Guaranteed safety in the presence of failures, no single-point of failure.
Scaling out large tenants	Schema-level partitioning	Co-locates data items frequently accessed within a transaction.
Membership management	Lease-based loose synchronization	Minimal synchronization overhead, fault-tolerance. Guarantees unique partition ownership.
Request routing	Thick client library	Simplifies client interactions with the system, transparent connection migration.
Efficient transaction execution	Unique ownership of database partitions. Classical RDBMS-like concurrency control and recovery	Obviates distributed synchronization during transaction execution. Proven and tested techniques for efficient transaction execution.
Fault tolerance	Replication in decoupled storage	Replication transparent to TM, thus simplifying its design.
Lightweight elasticity	Live database migration with no failed transactions	Low impact on tenants in terms of availability and response times.
Multitenancy	Shared process multitenancy	Good consolidation ratios and effective resource sharing.

configurable number of servers (called data nodes in HDFS). ElasTraS optimizes the number of writes to HDFS by batching DFS writes to amortize the cost.

5.2. Transaction Manager Layer

The transaction manager layer consists of a cluster of OTMs serving the partitions. Even though each ElasTraS OTM is analogous to a relational database, we implemented our own custom OTM for the following reasons. First, most existing open-source RDBMSs do not support dynamic partition reassignment, which is essential for live migration. Second, since traditional RDBMSs were not designed for multitenancy, they use a common transaction and data manager for all partitions sharing a database process. This makes tenant isolation more challenging. Third, ElasTraS uses advanced schema level partitioning for large tenants while most open-source databases only support simple table-level partitioning mechanisms off-the-shelf.

An ElasTraS OTM has two main components: the transaction manager responsible for concurrency control and recovery and the data manager responsible for storage and cache management.

5.2.1. Concurrency Control and Recovery. In our prototype, we implement Optimistic Concurrency Control (*OCC*) [Kung and Robinson 1981] to ensure serializability. However, note that any classical concurrency control technique, such as two-phase locking [Eswaran et al. 1976] or Snapshot Isolation [Berenson et al. 1995], can be used at an OTM. We implemented parallel validation in OCC, which results in a very short critical section for validation, thus allowing more concurrency [Kung and Robinson 1981].

Transaction logging is critical to ensure the durability of committed transactions in case of OTM failures. Every OTM maintains a *transaction log* (or log) where updates from transactions executing at the OTM are appended. All partitions at an OTM share a common log that is stored in the DFS to allow quick recovery from OTM failures. This sharing minimizes the number of DFS accesses and allows effective batching. Each log entry has a partition identifier to allow a log entry to be associated to the corresponding partition during recovery. Log entries are buffered during transaction execution. Once a transaction has been successfully validated, and before it can commit, a `COMMIT` record

for this transaction is appended to the log, and the log is *forced* (or flushed) to ensure durability. The following optimizations minimize the number of DFS accesses: (i) no log entry is written to record the start of a transaction, (ii) a `COMMIT` entry is appended only for update transactions, (iii) no log entry is made for aborted transactions, so the absence of a `COMMIT` record implies an abort, and (iv) group commits are implemented to commit transactions in groups and batch their log writes [Weikum and Vossen 2001]. (i) and (ii) ensure that there are no unnecessary log writes for read-only transactions. Buffering and group commits allow the batching of updates and are optimizations to improve throughput.

Log records corresponding to a transaction's updates are forced to the disk before the transaction commits. This ensures durability and transaction recoverability. In the event of a failure, an OTM recovers the state of failed partitions. The OTM replays the log to recreate the state prior to failure and recovers updates from all the committed transactions. We use a standard two-pass REDO recovery algorithm [Weikum and Vossen 2001]. Version information in the storage layer is used to guarantee idempotence of updates to guarantee safety during repeated OTM failures.

5.2.2. Storage and Cache management. The storage layer in ElasTraS is append-only, and the DM is designed to leverage this append-only nature. Persistent data for a partition consists of a collection of *immutable pages* that store the rows of a table in sorted order of their primary keys. Internally, a page is a collection of *blocks* with an index to map blocks to key ranges. This page index is used to read directly from the block containing the requested row and thus avoiding an unnecessary scan through the entire page.

Each OTM caches recently accessed data; the cache at an OTM is shared by all partitions being served by the OTM. Due to the append-only nature of the storage layout, a DM in ElasTraS uses separate read and write caches. Updates are maintained in the write cache that is periodically flushed to the DFS as new pages; a flush of the write cache is asynchronous and does not block new updates. As in-memory updates are flushed to the DFS, the corresponding entries in the transaction log are marked for garbage collection. The append-only semantics of the storage layer adds two new design challenges: (*i*) fragmentation of the pages due to updates and deletions, and (*ii*) ensuring that reads see the updates from the latest writes in spite of the two separate caches. A separate garbage collection process addresses the first challenge. The second challenge is addressed by answering queries from a merged view of the read and write caches. A least recently used policy is used for page eviction; since a page is immutable, its eviction does not result in a DFS write.

The immutable pages get fragmented internally with updates and deletes. A periodic garbage collection process recovers unused space and reduces the number of pages. In the same vein as Bigtable [Chang et al. 2006], we refer to this process as *compaction*. This compaction process executes in the background and merges multiple small pages into one large page, removing deleted or updated entries during this process. Old pages can still be used to process reads while the new compacted pages are being produced. Similarly, the log is also garbage collected as the write cache is flushed, thus reducing an OTM's recovery time.

5.3. Control Layer

The control layer has two components: the metadata manager (MM) and the TM Master. We use Zookeeper [Hunt et al. 2010] as the MM. The Zookeeper service provides low overhead distributed coordination and exposes an interface similar to a filesystem where each file is referred to as a *znode*. A Zookeeper client can request exclusive access to a znode, which is used to implement timed *leases* in the MM. Zookeeper

also supports *watches* on the leases that notify the watchers when the state of a lease changes. ElasTraS uses the Zookeeper primitives—leases and watches—to implement timed *leases* granted to the OTMs and the TM Master. Each OTM and the TM Master acquires exclusive write access to a znode corresponding to the server; an OTM or the TM Master is operational only if it owns its znode.

A system catalog maintains the mapping of a partition to the OTM serving the partition. Each partition is identified by a unique partition identifier; the catalog stores the partition-id to OTM mapping as well as the metadata corresponding to partitions, such as schema, authentication information, etc. The catalog is stored as a database within ElasTraS and is served by one of the live OTMs. The MM maintains the address of the OTM serving the catalog database.

The TM Master automates failure detection using periodic heart-beat messages and the MM's notification mechanism. Every OTM in the system periodically sends a heart-beat to the TM Master. In case of an OTM failure, the TM Master times-out on the heart beats from the OTM and the OTM's lease with the MM also expires. The master then atomically deletes the znode corresponding to the failed OTM. This deletion makes the TM Master the *owner* of the partitions that the failed OTM was serving and ensures safety in the presence of a network partition.

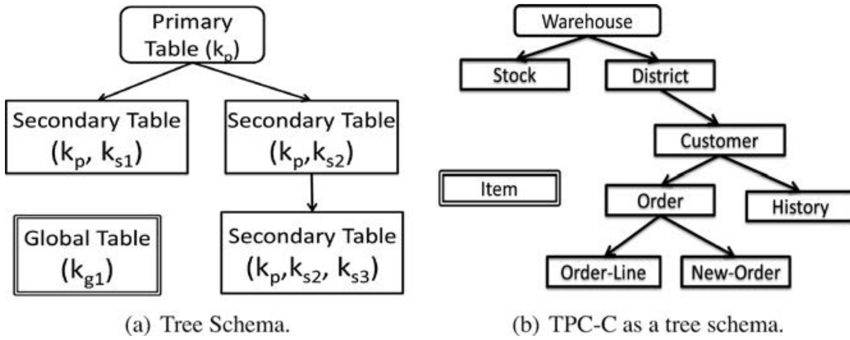
Once the znode for an OTM is deleted, the TM Master reassigns the failed partitions to another OTM. In an infrastructure supporting dynamic provisioning, the TM Master can spawn a new node to replace the failed OTM. In statically provisioned settings, the TM Master assigns the partitions of the failed OTM to the set of live OTMs. After reassignment, a partition is first recovered before it comes online. Since the persistent database and the transaction log are stored in the DFS, partitions can be recovered without the failed OTM recovering. An OTM failure affects only the partitions that OTM was serving; the rest of the system remains available.

ElasTraS also tolerates failures of the TM Master. A deployment can have a standby TM Master that registers a watch on the active TM Master's znode. When the acting TM Master fails, its lease on the znode expires, and the standby TM Master is notified. The standby TM Master then acquires the master lease to become the acting TM Master. The MM's notification mechanism notifies all the OTMs about the new master, which then initiates a connection with the new TM Master. This TM Master fail-over does not affect client transactions since the master is not in the data path of the clients. Therefore, there is no single point of failure in an ElasTraS installation.

5.4. Routing Layer

The routing layer comprises the ElasTraS client library, which abstracts the complexity of looking up and initiating a connection with the OTM serving the partition being accessed. Application clients link directly to the ElasTraS client library and submit transactions using the client interface. The ElasTraS client is responsible for: (i) looking up the catalog information and routing the request to the appropriate OTM, (ii) retrying the requests in case of a failure, and (iii) rerouting requests to a new OTM when a partition is migrated. When compared to the traditional Web-services architecture, the client library is equivalent to a proxy/load balancer, except that it is linked with the application.

The ElasTraS client library determines the location of the OTM serving a specific partition by querying the system catalog; the catalog information is cached at the clients after the initial lookup. A client accesses again the catalog tables only if there is a cache miss or an attempt to connect to a cached location fails. In the steady state, the clients directly connect to the OTM using cached catalog information. The distributed nature of the routing layer allows the system to scale to large numbers of tenants and clients.



(a) Tree Schema. (b) TPC-C as a tree schema.

Fig. 2. Schema level database partitioning.

5.5. Advanced Implementation Aspects

5.5.1. Schema Level Partitioning for Large Tenants. Our discussion so far has focused on small tenants that fit in a single ElasTraS partition; ElasTraS allows such small tenants to have any database schema as required by their application. However, ElasTraS can also serve tenants that grow big, both in terms of data sizes as well as loads, by using *schema-level partitioning* of the tenant databases for scale-out. Since the inefficiencies resulting from distributed transactions are well known [Curino et al. 2010], the choice of a good partitioning technique is critical to support flexible functionality while limiting transactions to a single partition. Transactions accessing a single partition can be executed efficiently without any dependency and synchronization between OTMs, thus allowing high scalability and availability. Moreover, live database migration and elastic load balancing is considerably simpler with single partition transactions. Therefore, even though the design has nothing inherent to disallow distributed transactions, leaving them out was a conscious decision in favor of other features, such as scalability and elasticity, that are more important for the target space of ElasTraS.

We now explain a particular instance of schema level partitioning. ElasTraS supports a tree-based schema for partitioned databases. Figure 2(a) provides an illustration of such a schema type. This schema supports three types of tables: the *Primary Table*, *Secondary Tables*, and *Global Tables*. The primary table forms the root of the tree; a schema has one primary table whose primary key acts as the partitioning key. A schema can however have multiple secondary and global tables. Every secondary table in a database schema will have the primary table's key as a foreign key. Referring to Figure 2(a), the key k_p of the primary table appears as a foreign key in each of the secondary tables.

This tree structure implies that corresponding to every row in the primary table, there are a group of related rows in the secondary tables, a structure similar to row groups in Cloud SQL Server [Bernstein et al. 2011a]. All rows in the same row group are guaranteed to be co-located. A transaction can only access rows in a particular row group. A database partition is a collection of such row groups. This schema structure also allows efficient dynamic splitting and merging of partitions. In contrast to the two table types, global tables are look-up tables that are mostly read-only. Since global tables are not updated frequently, these tables are replicated on all the nodes; decoupled storage allows ElasTraS to cache the global tables at the OTMs without actually replicating the data. In addition to accessing only one row group, an operation in a transaction can also read from a global table. Figure 2(b) shows a representation of the TPC-C schema [Transaction Processing Performance Council 2009] as a tree

schema. The TATP benchmark [Neuvonen et al. 2009] is another example of a schema conforming to the tree schema.

ElasTraS requires databases to conform to the tree schema to enable partitioning; small tenant databases contained within a single partition do not require to have a tree schema. For such a configuration, a transaction can access any data item within the partition and the partition is the granule of transactional access. This configuration is similar to table groups in Cloud SQL Server.

The “tree schema” has been demonstrated to be amenable to partitioning using a partition key shared by all the tables in the schema. Even though such a schema does not encompass the entire spectrum of OLTP applications, a survey of real applications within a commercial enterprise shows that a large number of applications either have such an inherent schema pattern or can be easily adapted to it [Bernstein et al. 2011a]. A similar schema pattern is also observed in two other concurrently developed systems, such as Cloud SQL Server and Megastore. Since ElasTraS operates at the granularity of the partitions, the architecture can also leverage other partitioning techniques, such as Schism [Curino et al. 2010].

5.5.2. Dynamic Partitioning. In addition to limiting transactions to a single node, an added benefit of a tree schema is that it lends ElasTraS the ability to dynamically split or merge partitions. In a tree schema, a partition is a collection of row groups. A partition can therefore be dynamically split into two sets of row groups. To split a partition, an OTM first splits the primary table and then uses the primary table’s key to split the secondary tables. Global tables are replicated, and hence are not split. The primary table is physically organized using its key order and the rows of the secondary tables are first ordered by the primary table’s key and then by the keys of the secondary. This storage organization allows efficient splitting of partitions, since the key used to split the primary key can also be used to split the secondary tables into two parts. To ensure a quick split, an OTM performs a logical split; the two child partitions reference the top and bottom halves of the old parent partition. The OTM performing the split updates the system state to reflect the logical split. Thus, splitting does not involve data shuffling and is efficient. To ensure correctness of the split in the presence of a failure, this split is executed as a transaction, thus providing an atomic split. Once a partition is split, the TM Master can reassign the two subpartitions to different OTMs. Data is moved to the new partitions asynchronously and the references to the old partition are eventually cleaned up during garbage collection.

5.5.3. Multiversion Data. The storage layer of ElasTraS is append-only and hence multiversion. This allows for a number of implementation optimizations. First, read-only queries that do not need the latest data can run on snapshots of the database. Second, for large tenants spanning multiple partitions, read-only analysis queries can be executed in parallel without the need for distributed transactions. Third, if a tenant experiences high transaction load, most of which is read-only, the read-only transactions can be processed by another OTM that executes transactions on a snapshot of the tenant’s data read from the storage layer. Since such transactions do not need validation, they can execute without any coordination with the OTM currently owning the tenant’s partition. The use of snapshot OTMs in ElasTraS is only on-demand to deal with high load. The snapshot OTMs also allow the coexistence of transaction processing and analysis workloads.

6. LIVE DATABASE MIGRATION

Elasticity and pay-per-use pricing are the key features of cloud computing that obviate the need for static resource provisioning for peak loads and allows on-demand resource provisioning based on the workload. However, DBMSs are typically not as elastic as the

other tiers of the web-application stack. In order to effectively leverage the underlying elastic infrastructure, the DBMSs must also be elastic, that is, when the load increases, they can add more servers to the database tier and migrate some database partitions to the newly added servers to distribute the load, and vice versa. Elastic scaling and load balancing therefore calls for lightweight techniques to migrate database partitions in a live system.

We now present Albatross, a live database migration protocol. We consider migration of both database partitions, and in the case of small tenants, a tenant's entire database. Hence, the terms partition and tenant are used interchangeably in this section.

Albatross is similar to the iterative phase-by-phase copying proposed by Clark et al. [2005]. The major difference is that Clark et al. use VM-level memory page copying and OS and networking primitives to transfer live processes and network connections. On the other hand, Albatross leverages the semantics of a DBMS and its internal structures to migrate the cached database pages, state of active transactions, and the database connections. Another major difference is that Albatross must ensure that in the event of a failure during migration, the transactional guarantees of the database are not violated. Thus, failure handling and recovery form an integral component of Albatross. The details of how Albatross leverages the internal semantics of a DBMS is explained in detail later in this section.

6.1. Migration Cost Metrics

Migrating database partitions in a live system is a hard problem and comes at a cost. We discuss four cost metrics to evaluate the effectiveness of a live database migration technique, both from the user and the system perspective.

—*Service unavailability.* The duration of time for which a database partition is unavailable during migration. Unavailability is defined as the period of time when all requests to the database fail.

—*Number of failed requests.* The number of well-formed requests that fail due to migration. Failed requests include both aborted transactions and failed operations; a transaction consists of one or more operations. Aborted transactions signify failed interactions with the system while failed operations signify the amount of work wasted as a result of an aborted transaction. The failed operations account for transaction complexity; when a transaction with more operations aborts, more work is wasted for the tenant. We therefore quantify both types of failures in this cost metric.

—*Impact on response time.* The change in transaction latency (or response time) observed as a result of migration. This metric factors any overhead introduced in order to facilitate migration as well as the impact observed during and after migration.

—*Data transfer overhead.* Any additional data transferred during migration. Database migration involves the transfer of data corresponding to the partition being migrated. This metric captures the messaging overhead during migration as well any data transferred in addition to the minimum required to migrate the partition.

The first three cost metrics measure the external impact on the tenants while the last metric measures the internal performance impact. In a cloud data platform, a long unavailability window or a large number of failed requests resulting from migration might violate a tenant's availability requirements, thus resulting in a penalty. For instance, in Google AppEngine, if the availability drops below 99.95%, then tenants receive a service credit.[2] Similarly, low transaction latency is critical for good tenant performance. Similar to availability, a response time higher than a threshold can incur

[2]http://code.google.com/appengine/sla.html.

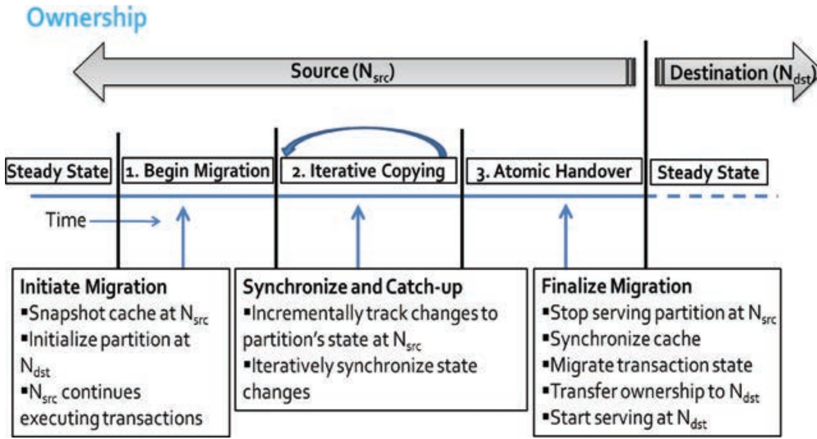


Fig. 3. Migration timeline for Albatross (times not drawn to scale).

Table II. Notational conventions

Notation	Description
$\mathbb{P}_M$	The tenant database being migrated
$\mathbb{N}_S$	Source node for $\mathbb{P}_M$
$\mathbb{N}_D$	Destination node for $\mathbb{P}_M$

a penalty in some service models. To be effectively used elastic load balancing, a live migration technique must have minimal impact on the tenants' performance.

6.2. Design Overview

Albatross aims to have minimal impact on tenant's performance while leveraging the semantics of the database structures for efficient database migration. This is achieved by iteratively transferring the database cache and the state of active transactions. For an Optimistic Concurrency Control (OCC) [Kung and Robinson 1981] scheduler, this state consists of the read-sets and write-sets of active transactions and a subset of committed transactions. Figure 3 depicts the timeline of Albatross when migrating a partition ($\mathbb{P}_M$) from the source node ($\mathbb{N}_S$) to the destination node ($\mathbb{N}_D$). Table II lists the notational conventions.

Phase 1: Begin Migration. Migration is initiated by creating a snapshot of the database cache at $\mathbb{N}_S$. This snapshot is then copied to $\mathbb{N}_D$. $\mathbb{N}_S$ continues processing transactions while this copying is in progress.

Phase 2: Iterative Copying. Since $\mathbb{N}_S$ continues serving transactions for $\mathbb{P}_M$ while $\mathbb{N}_D$ is initialized with the snapshot, the cached state of $\mathbb{P}_M$ at $\mathbb{N}_D$ will lag that of $\mathbb{N}_S$. In this iterative phase, at every iteration, $\mathbb{N}_D$ tries to "catch-up" and synchronize the state of $\mathbb{P}_M$ at $\mathbb{N}_S$ and $\mathbb{N}_D$. $\mathbb{N}_S$ tracks changes made to the database cache between two consecutive iterations. In iteration i, Albatross copies to $\mathbb{N}_D$ the changes made to $\mathbb{P}_M$'s cache since the snapshot of iteration $i - 1$. This phase terminates when approximately the same amount of state is transferred in consecutive iterations or a configurable maximum number of iterations have completed.

Phase 3: Atomic Handover. In this phase, the ownership of $\mathbb{P}_M$ is transferred from $\mathbb{N}_S$ to $\mathbb{N}_D$. $\mathbb{N}_S$ stops serving $\mathbb{P}_M$, copies the final un-synchronized database state and the state of active transactions to $\mathbb{N}_D$, flushes changes from *committed* transactions to the persistent storage, transfers control of $\mathbb{P}_M$ to $\mathbb{N}_D$, and metadata partitions to point to the new location of $\mathbb{P}_M$. To ensure safety in the presence of failures,

this operation is guaranteed to be *atomic*. The successful completion of this phase makes $\mathbb{N}_D$ the owner of $\mathbb{P}_M$ and completes the migration.

The iterative phase minimizes the amount of $\mathbb{P}_M$'s state to be copied and flushed in the handover phase, thus minimizing the unavailability window. In the case where the transaction logic executes at the client, transactions are seamlessly transferred from $\mathbb{N}_S$ to $\mathbb{N}_D$ without any loss of work. The handover phase copies the state of active transaction along with the database cache. For an OCC scheduler, the handover phase copies the read/write sets of the active transactions and that of a subset of committed transactions whose state is needed to validate new transactions. For transactions executed as stored procedures, $\mathbb{N}_S$ tracks the invocation parameters of transactions active during migration. Any such transactions active at the start of the handover phase are aborted at $\mathbb{N}_S$ and are automatically restarted at $\mathbb{N}_D$. This allows migrating these transactions without process state migration [Theimer et al. 1985]. Durability of transactions that committed at $\mathbb{N}_S$ is ensured by synchronizing the commit logs of the two nodes.

Iterative copying in Albatross is reminiscent of migration techniques used in other domains, such as in VM migration [Clark et al. 2005] and process migration [Theimer et al. 1985]. The major difference is that Albatross leverages the semantics of the database internals to copy only the information needed to re-create the state of the partition being migrated.

6.3. Failure Handling

In the event of a failure, data safety is primary while progress towards successful completion of migration is secondary. Our failure model assumes node failures and a connection oriented network protocol, such as TCP, that guarantees message delivery in order within a connection. We do not consider byzantine failures or malicious node behavior. We further assume that node failures do not lead to complete loss of the persistent data: either the node recovers or the data is recovered from the DFS where data persists beyond DBMS node failures. If either $\mathbb{N}_S$ or $\mathbb{N}_D$ fails prior to Phase 3, migration of $\mathbb{P}_M$ is aborted. Progress made in migration is not logged until Phase 3. If $\mathbb{N}_S$ fails during Phases 1 or 2, its state is recovered, but since there is no persistent information of migration in the commit log of $\mathbb{N}_S$, the progress made in $\mathbb{P}_M$'s migration is lost during this recovery. $\mathbb{N}_D$ eventually detects this failure and in turn aborts this migration. If $\mathbb{N}_D$ fails, migration is again aborted since $\mathbb{N}_D$ does not have any log entries for a migration in 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.

The atomic handover phase (Phase 3) consists of the following major steps: (i) flushing changes from all committed transactions at $\mathbb{N}_S$; (ii) synchronizing the remaining state of $\mathbb{P}_M$ between $\mathbb{N}_S$ and $\mathbb{N}_D$; (iii) transferring ownership of $\mathbb{P}_M$ from $\mathbb{N}_S$ to $\mathbb{N}_D$; and (iv) notifying the query router that all future transactions must be routed to $\mathbb{N}_D$. Steps (iii) and (iv) can only be performed after the Steps (i) and (ii) complete. Ownership transfer involves three participants—$\mathbb{N}_S$, $\mathbb{N}_D$, and the query router—and must be atomic. We perform this handover as a *transfer transaction* and a Two Phase Commit (2PC) protocol [Gray 1978] with $\mathbb{N}_S$ as the coordinator guarantees atomicity in the presence of node failures. In the first phase, $\mathbb{N}_S$ executes steps (i) and (ii) in parallel, and solicits a vote from the participants. Once all the nodes acknowledge the operations and vote *yes*, the transfer transaction enters the second phase where $\mathbb{N}_S$ relinquishes control of $\mathbb{P}_M$ and transfers it to $\mathbb{N}_D$. If one of the participants votes "no", this transfer transaction is aborted and $\mathbb{N}_S$ remains the owner of $\mathbb{P}_M$. This second step completes the transfer transaction at $\mathbb{N}_S$, which, after logging the outcome, notifies the participants about the decision. If the handover was successful, $\mathbb{N}_D$ assumes ownership of $\mathbb{P}_M$ once

it receives the notification from $\mathbb{N}_S$. Every protocol action is logged in the commit log of the respective nodes.

6.4. Implementing Albatross

Creating a database snapshot. In the first step of migration, $\mathbb{N}_S$ creates a snapshot of the tenant's database cache. Albatross does not require a transactionally consistent snapshot of the cache. $\mathbb{N}_S$'s cache snapshot is a list of identifiers for the immutable page blocks that are cached. This list of block identifiers is obtained by scanning the read cache using a read lock. It is passed to $\mathbb{N}_D$, which then reads the blocks directly from the DFS and populates its cache. This results in minimal work at $\mathbb{N}_S$ and delegates all the work of warming up the cache to $\mathbb{N}_D$. The rationale is that during migration, $\mathbb{N}_D$ is expected to have less load compared to $\mathbb{N}_S$. Therefore, transactions at $\mathbb{N}_S$ observe minimal impact during snapshot creation and copying. After $\mathbb{N}_D$ has loaded all the blocks into its cache, it notifies $\mathbb{N}_S$ of the amount of data transferred (Δ_0); both nodes now enter the next iteration. No transaction state is copied in this phase.

Iterative copying phase. In every iteration, changes made to the read cache at $\mathbb{N}_S$ are copied to $\mathbb{N}_D$. After a first snapshot is created, the data manager of $\mathbb{P}_M$ at $\mathbb{N}_S$ tracks changes to the read cache (both insertions and evictions) and incrementally maintains the list identifiers for the blocks that were evicted from or loaded in to the read cache since the previous iteration, which is copied to $\mathbb{N}_D$ in subsequent iterations. Again, only the block identifiers are passed; $\mathbb{N}_D$ populates its cache using the identifiers and notifies $\mathbb{N}_S$ of the amount of data transferred (Δ_i). This iterative phase continues until the amount of data transferred in successive iterations is approximately the same, that is, $\Delta_i \approx \Delta_{i-1}$. The rationale behind this termination condition is that when $\Delta_i \approx \Delta_{i-1}$, irrespective of the magnitude of Δ_i, little gain is expected from subsequent iterations. Δ_i is small for most cases, except when the working set of the database does not fit into the cache, thus resulting in frequent changes to the cache due to pages being evicted and new pages added to the cache. A maximum bound on the number of iterations ensures termination when Δ_i fluctuates between iterations.

The write cache is periodically flushed during the iterative copying phase when its size exceeds a specified threshold. A write-cache flush creates a new block whose identifier is passed to $\mathbb{N}_D$, which loads the new block into its read cache. After the handover, $\mathbb{N}_D$ starts serving $\mathbb{P}_M$ with an empty write cache, but the combination of the read and write cache contains the same state of data as in $\mathbb{N}_S$. Since the data manager hides this cache separation, transactions on $\mathbb{P}_M$ continue execution at $\mathbb{N}_D$ unaware of the migration.

Copying the transaction state. The transaction state consists of the read and write sets of the active transactions and a subset of committed transactions needed to validate new transactions. The read/write sets of active transactions and committed transactions are maintained in separate main-memory structures. Two counters are used to assign transaction numbers and commit sequence numbers. In Albatross, the transaction state is copied only in the final handover phase. Writes of an active transaction, stored locally with the transaction's state, are also copied to $\mathbb{N}_D$ during handover along with the counters maintained by the transaction manager. The state of a subset of committed transactions (ones that committed after any one of the current set of active transactions started) is copied to $\mathbb{N}_D$ to validate the active transactions at $\mathbb{N}_D$. The small size of transaction states allows efficient serialization. After handover, $\mathbb{N}_D$ has the exact same transaction state of $\mathbb{P}_M$ as $\mathbb{N}_S$, thus allowing it to continue executing the transactions that were active at the start of the handover phase.

Handover phase. The handover phase flushes changes from committed transactions. After the transaction state and the final changes to the read cache have been copied, the atomic handover protocol makes $\mathbb{N}_D$ the unique owner of $\mathbb{P}_M$ and updates the mapping in the catalog used by the query router. The ElasTraS clients cache the routing information. After handover, $\mathbb{N}_S$ rejects any request to $\mathbb{P}_M$, which invalidates the catalog information cached at the clients.

Recall that the system catalog is served by one of the live OTMs. The OTM serving the catalog participates in the transfer transaction of the atomic handover phase. The TM Master can be a participant of the transfer transaction so that it is aware of the outcome of migration; however, it is not needed for correctness. In our implementation, the TM Master is notified by $\mathbb{N}_S$ after handover completes. Clients that have open connections with $\mathbb{P}_M$ at $\mathbb{N}_S$ are notified directly about the new address of $\mathbb{N}_D$. This prevents an additional network round-trip.

For a transaction accessing $\mathbb{P}_M$ during the atomic handover phase, the ElasTraS client library transparently retries the operation; once the handover completes, this retried operation is routed to $\mathbb{N}_D$. Since an OTM's commit log is stored in the DFS, it is not migrated. $\mathbb{P}_M$'s transaction log at $\mathbb{N}_S$ is garbage collected once the transactions active at the start of the handover phase have completed at $\mathbb{N}_D$, though the entries for transactions that committed at $\mathbb{N}_S$ can be purged after the handover completes.

6.5. System Modeling for Elastic Load Balancing

Modeling the system's performance, behavior, and workloads plays an important role in determining when to scale up the system, when to consolidate to a fewer number of nodes, and how to assign partitions to the OTMs. Every live OTM periodically sends heartbeats to the TM Master that include statistics about the load on the OTM, that is, the number of partitions, the number of requests per partition, CPU, IO and other resource utilization metrics. The TM Master aggregates these statistics across all the OTMs to create a model for the overall system performance and incrementally maintains it to create a history of the system's behavior. This model is then used to load balance partitions and to decide on elastic scaling of the system. Our approach to system modeling and control is similar to many closed-loop controllers used in many systems, such as Trushkowsky et al. [2011].

The current implementation uses a rule-based model. If the resource utilization at a configurable percentage of OTMs is above a configurable threshold, then new OTM instances are added. On the other hand, if utilization is below a min-threshold, partitions are consolidated into a smaller number of nodes. The threshold values are based on performance measurements of utilization levels above which a considerable impact on tenant transactions is observed, or under which the system's resources are being under-utilized. Historical information is used to smooth out transient spikes in load. In the future, we plan to explore model advanced system modeling and prediction techniques based on control theory, machine learning, and time series analysis.

The partition assignment algorithm determines an assignment of partitions to OTMs ensuring that the tenant performance requirements are met while consolidating the tenants into as few servers as possible. In our implementation, we assume that a tenant's acceptable performance requirements, in terms of average latency per request, is provided upfront. The tenant and the service provider agree on a mutually acceptable average response time before the provider starts serving the tenant. Since our focus is OLTP scenarios, response times are critical for such applications. The problem of automatically determining the appropriate performance service level agreements (SLAs) for a given tenant workload is beyond the scope of this system. We model the partition assignment problem as an optimization problem. In our current prototype, we use a greedy heuristic where the historical load and resource utilization statistics

per partition are used to assign partitions to an OTM without overloading it. A newly created partition for which no historical information is available is assigned to the OTM least loaded in terms of resource utilization. While the system is serving the tenant transactions, the average response times are monitored and if they appear to be higher than what was agreed upon, the tenant is migrated to a more lightly loaded node. Therefore, our implementation provides best-effort quality of service.

7. EXPERIMENTAL EVALUATION

In this section, we provide a detailed experimental evaluation of our prototype implementation of ElasTraS to analyze the impact of the three major design choices of ElasTraS: the shared storage layer, shared process multitenancy, and colocating tenant's data into a single partition and limiting transactions to a single partition. We also evaluate Albatross's effectiveness for lightweight elastic scaling.

In this section, we first explain the cluster setup for our experiments (Section 7.1) and then explain the two benchmarks, TPC-C and YCSB, used in our experiments (Section 7.2). In the first set of experiments, we evaluate ElasTraS with static cluster configurations. We evaluate the performance of a single tenant in isolation (Section 7.3), the effect of multitenancy at a single OTM (Section 7.4), aggregate system performance for clusters of different sizes (Section 7.5), and finally for scaling large tenant databases using schema level partitioning (Section 7.6). The second set of experiments evaluate the effectiveness of Albatross for a wide variety of workloads (Section 7.7). We then evaluate the elastic load balancing feature of ElasTraS where the workloads change dynamically resulting in servers being added to or removed from the cluster (Section 7.8). Finally, we discuss some lessons learned from our prototype implementation and evaluation (Section 7.9).

7.1. Experimental Setup

Our experiments were performed on a local cluster and on Amazon Web Services Elastic Compute Cloud (*EC2*). Servers in the local cluster had a quad core processor, 8 GB RAM, and 1 TB disk. On EC2, we use the `c1.xlarge` instances each of which is a virtual machine with eight virtual cores, 7 GB RAM, and 1.7 TB local storage. ElasTraS is implemented in Java and uses HDFS v0.21.0 [HDFS 2010] as the storage layer and Zookeeper v3.2.2 [Hunt et al. 2010] as the metadata manager. We use a three node Zookeeper ensemble in our experiments. The nodes in the Zookeeper ensemble also host the ElasTraS TM Master and the HDFS name node and secondary name node. A separate ensemble of nodes host the slave processes of ElasTraS and HDFS: that is, the ElasTraS OTMs that serve the partitions, and the data nodes of HDFS, which store the filesystem data. In the local cluster, the OTMs and the data nodes were executed on different nodes, while in EC2, each node executed the data node and the OTM processes.

The performance of the storage layer (HDFS) impacts ElasTraS's performance. Several studies have reported on HDFS performance [Tankel 2010]. In our experimental configuration using three replicas, we observed an average read bandwidth of 40MBps when reading sequentially from a 64MB block in HDFS. When reading small amounts of data, the typical read latency was on the order of 2 to 4 ms. A latency on the order of 5 ms was observed for appends. In our implementation, log appends are the only synchronous writes, a flush of the cache is performed asynchronously. Group commit amortizes the cost of a log append over multiple transactions. The cost of accessing data from the storage layer is therefore comparable to that of a locally attached storage.

7.2. Workload Specification

Our evaluation uses two different OLTP benchmarks that have been appropriately adapted to a multitenant setting: (i) the Yahoo! cloud serving benchmark

Table III. Default values for YCSB parameters

Parameter	Default value
Transaction size	10 operations
Read/Write distribution	9 reads 1 write
Database size	4 GB
Transaction load	200 transactions per second (TPS)
Hotspot distribution	90% operations accessing 20% of the database

(YCSB) [Cooper et al. 2010] adapted for transactional workloads to evaluate performance under different read/write loads and access patterns; and (ii) the TPC-C benchmark [Transaction Processing Performance Council 2009] representing a complex transactional workload for typical relational databases.

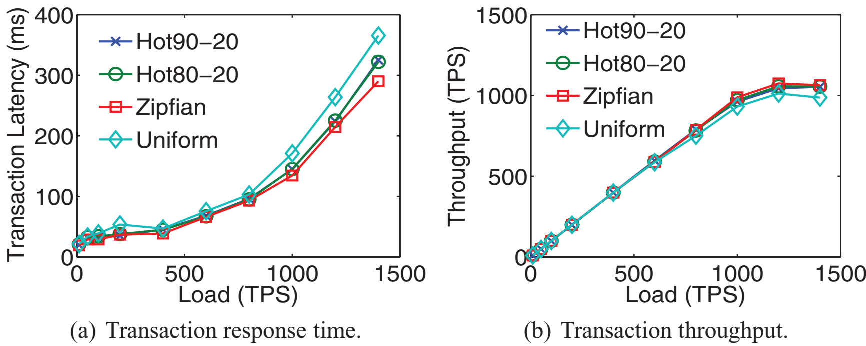
7.2.1. Yahoo! Cloud Serving Benchmark. YCSB [Cooper et al. 2010] is a benchmark to evaluate DBMSs serving web applications. The benchmark was initially designed for key-value stores and hence did not support transactions. We extended the benchmark to add support for multistep transactions that access data only from a single database partition. We choose a tenant schema with three tables where each table has ten columns of type VARCHAR and 100 byte data per column; one of the tables is the primary and the two remaining tables are secondary. The workload consists of a set of multistep transactions parameterized by the number of operations, percentage of reads and updates, and the distribution (uniform, Zipfian, and hotspot distributions) used to select the data items accessed by a transaction. We also vary the transaction loads, database sizes, and cache sizes. To simulate a multitenant workload, we run one benchmark instance for each tenant. YCSB comprises small transactions whose logic executes at the clients.

7.2.2. TPC-C Benchmark. The TPC-C benchmark is an industry standard benchmark for evaluating the performance of OLTP systems [Transaction Processing Performance Council 2009]. The TPC-C benchmark suite consists of nine tables and five transactions that portray a wholesale supplier. The five transactions represent various business needs and workloads: (i) the NewOrder transaction, which models the placing of a new order; (ii) the Payment transaction, which simulates the payment of an order by a customer; (iii) the OrderStatus transaction representing a customer query for checking the status of the customer's last order; (iv) the Delivery transaction representing deferred batched processing of orders for delivery; and (v) the StockLevel transaction, which queries for the stock level of some recently sold items. A typical transaction mix consists of approximately 45% NewOrder transactions, 43% Payment transactions, and 4% each of the remaining three transaction types, representing a good mix of read/write transactions. The number of warehouses in the database determines the scale of the system. Since more than 90% of transactions have at least one write operation (insert, update, or delete), TPC-C represents a write heavy workload. Each tenant is an instance of the benchmark. The TPC-C benchmark represents a class of complex transactions executed as stored procedures at an OTM.

7.3. Single Tenant Behavior

We first evaluate the performance of ElasTraS for a single tenant in isolation, that is, an OTM serving only one tenant whose database is contained within a partition. This allows us to analyze the performance of the different components in the design, that is, the transaction manager, cache, and interactions with the storage layer.

7.3.1. Yahoo! Cloud Serving Benchmark. Table III summarizes the default values of some YCSB parameters. To evaluate the performance of a single tenant in isolation, we vary



(a) Transaction response time. (b) Transaction throughput.

Fig. 4. Evaluating the impact of offered load on throughput and latency. This experiment uses YCSB and runs a single tenant database in isolation at an OTM.

different YCSB parameters while using the default values for the rest of the parameters. The YCSB parameters varied in these experiments are: number of operations in a transaction or *transaction size*, number of read operations in a transaction or *read/write ratio*, database size, number of transactions issued per second (TPS) or *offered load*, and *cache size*. The data items accessed by a transaction are chosen from a distribution; we used Zipfian (coefficient 1.0), uniform, and hotspot. A hotspot distribution simulates a hot spot where $x\%$ of the operations access $y\%$ of the data items. We used a hot set comprising 20% of the database and two workload variants with 80% and 90% of the operations accessing the hot set.

Figure 4 plots the transaction latency and throughput as the load on the tenant varied from 20 to 1400 TPS. Until the peak capacity is reached, the throughput increases linearly with the load and is accompanied by a gradual increase in transaction latency; limiting transactions to a single partition allows this linear scaling. The throughput plateaus at about 1100 TPS and a sharp increase in latency is observed beyond this point. The system thrashes for a uniform distribution due to high contention in the network arising from high cache miss percentage. Thrashing is observed in OCC only in the presence of heavy resource contention. This behavior is confirmed for the skewed distributions where the working set is mostly served from the database cache. A higher cache miss rate also results in higher latency for the uniform distribution.

As observed from Figure 4, the behavior of all skewed workloads is similar as long as the working set fits in the database cache. We therefore choose a hotspot distribution with 90% operations accessing 20% data items (denoted by Hot90-20) as a representative skewed distribution for the rest of the experiments. Since our applied load for the remaining experiments (200 TPS) is much lower than the peak, the throughput is equal to the load. We therefore focus on transaction latency where interesting trends are observed.

Figure 5 plots the impact of transaction size; we vary the number of operations from 5 to 30. As expected, the transaction latency increases linearly with a linear increase in the transaction size. Transaction latency also increases as the percentage of update operations in a transaction increases from 10% to 50% (see Figure 6). This increase in latency is caused by two factors: first, due to the append-only nature of the storage layer, more updates results in higher fragmentation of the storage layer, resulting in subsequent read operations becoming more expensive. The latency increase is less significant for a skewed distribution since the more recent updates can often be found in the cache thus mitigating the impact of fragmentation. Second, a higher fraction of write operations implies a more expensive validation and commit phase for OCC, since

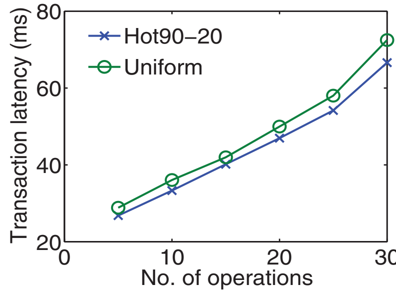


Fig. 5. Impact of transaction size.

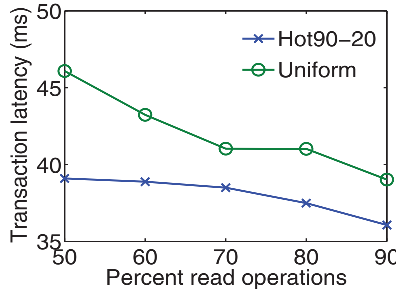


Fig. 6. Impact of read percent.

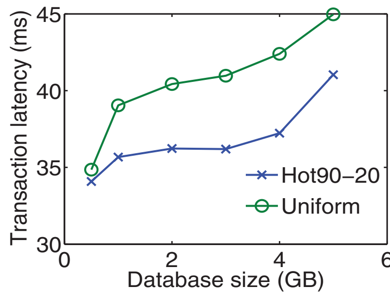


Fig. 7. Impact of database size.

a transaction must be validated against all write operations of concurrently executing transactions.

Figure 7 plots the effect of database size on transaction latency; we varied the database size from 500MB to 5GB. For a database size of 500MB, the entire database fits in the cache resulting in comparable transaction latencies for both uniform and skewed distributions. As the database size increases, a uniform access distribution results in more cache misses and hence higher transaction latency. When the database size is 5GB, the working set of the skewed distribution also does not fit in the result, resulting in a steep increase in latency. Similarly, when varying the cache size, transaction latency decreases with an increase in cache size resulting from higher cache hit ratios (see Figure 8 where we varied the cache size from 200MB to 1GB).

7.3.2. Evaluating Alternative Architectures. An alternative architecture for designing a system with goals similar to ElasTraS is to use a key-value store and represent a tenant

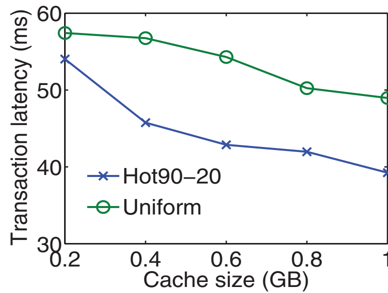


Fig. 8. Impact of cache size.

as a row in the key-value store. Most key-value stores provide the functionality of executing single-row transactions. Therefore, in principle, such a design can provide transactional access to a tenant's data. Multiple rows, or tenants, can be served at a node, thus providing support for multitenancy.

As a concrete instance, we use HBase as a key-value store implementation to prototype the aforementioned design. HBase's data model is a sparse multidimensional map where a data item is keyed by the row key, the column family name, column, and timestamp. Since a tenant's entire database is encapsulated in a row, the tenant id is used as the row key. For each table in the tenant's schema, the table name combined with the attribute name forms a column family, the value of the primary key corresponding to a row in the tenant's table forms the column name, and finally, the remaining attributes are stored as the value. Each table in the tenant's schema in HBase supports an atomic compare and swap operation. However, the granularity of this compare and swap is a single value, which is not enough to support multioperation transactions on a single tenant similar to ElasTraS. Therefore, we use the *row lock* API of HBase to provide serializable transactions on a row. However, since a row lock serializes all access to a row, transactions end up executing serially instead of executing serializably. Using the YCSB benchmark configurations from Section 7.3.1, a peak throughput for a single tenant was about 20 TPS compared to about 1100 TPS for ElasTraS. Further, as the number of concurrent transactions increased, the waits introduced in obtaining the row lock increased considerable, which resulted in a significant increase in latencies with many requests timing out even at offered load less than 100 TPS.

Other key-value store implementations might provide an API to directly execute transactions on a row. However, since such systems were not designed to efficiently execute transactions, such API implementations often resort to coarse-grained concurrency control, which limits the peak transaction throughput per tenant. For instance, Megastore's implementation of transactions as a layer on top of Bigtable also results in the serial execution of transactions within an entity group [Baker et al. 2011]. One of the key insights in our design of ElasTraS is that once transactions are limited to a single node, we can leverage techniques similar to those used in classical RDBMSs to efficiently execute transactions. ElasTraS therefore uses an OTM to efficiently execute transactions on a tenant and is expected (as demonstrated in this experiment) to have considerably better throughput and latency. As reported in Section 7.3, ElasTraS supports a peak throughput of around 1100 TPS.

Since an ElasTraS OTM is similar to a single node RDBMS engine, it should therefore exhibit similar behavior. To compare the behavior of ElasTraS with that of a traditional RDBMS, we repeated the above experiment using two other open-source RDBMSs, MySQL v5.1 and H2 v1.3.148, running on the same hardware with the same cache

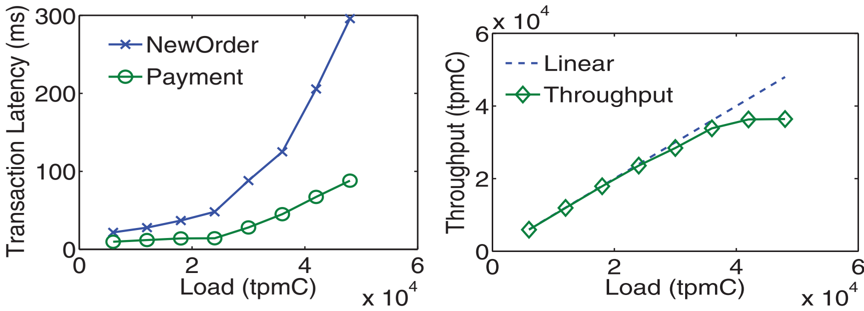


Fig. 9. Evaluating the performance of a single tenant in isolation using TPC-C.

settings.[3] A behavior similar to that shown in Figure 4 was observed; the only difference was the peak transaction throughput: MySQL's peak was about 2400 TPS and H2's peak was about 600 TPS.

7.3.3. TPC-C Benchmark. In this experiment, we use a TPC-C database with ten warehouses resulting in a data size of 2.5GB. The cache size is set to 1GB. Figure 9 plots the average transaction latency and throughput as the TPC-C metric of transactions per minute (tpmC), where a behavior similar to that in YCSB is observed. We report the latency of NewOrder and Payment transactions that represent 90% of the TPC-C workload. The NewOrder is a long read/write transaction accessing multiple tables while Payment is a short read/write transaction accessing a small number of tables. Hence the payment transaction has a lower latency compared to NewOrder. The throughput is measured for the entire benchmark workload. The gains from executing transactions as stored procedures is evident from the lower latency of the more complex NewOrder transaction (executing close to a hundred operations per transaction) compared to that of the simpler transactions in YCSB (executing 10–20 operations).

7.4. Multitenancy at a Single OTM

We now evaluate performance of multiple tenants hosted at the same OTM, thus evaluating the tenant isolation provided by the shared process multitenancy model. We use YCSB for these experiments to emulate a multitenant setting with large numbers of small tenants executing short read-write transactions.

Figure 10 reports transaction latency and throughput at one OTM. In the plot for transaction latency, the data points plotted correspond to the average latency across all tenants, while the error bars correspond to the minimum and maximum average latency observed for the tenants. Each tenant database is about 500MB in size and the total cache size is set to 2.4GB. The load on each tenant is set to 100TPS. We increase the number of tenants per OTM, thus increasing the load on the OTM. For clarity of presentation, we use a hot spot distribution where 90% of the operations access 20% of the data items as a representative skewed distribution, and a uniform distribution. For skewed workloads when the working set fits in the cache, even though different tenants compete for the OTM resources, the performance is similar to that of a single tenant with the same load. The only observable difference is for uniform distribution at high loads where high contention for the network and storage results in a higher variance in transaction latency. Similarly, the peak throughput is also similar to that observed in the experiment for a single tenant (see Figure 4). This experiment demonstrates the good isolation provided by ElasTraS.

[3]MySQL: www.mysql.com/, H2: www.h2database.com.

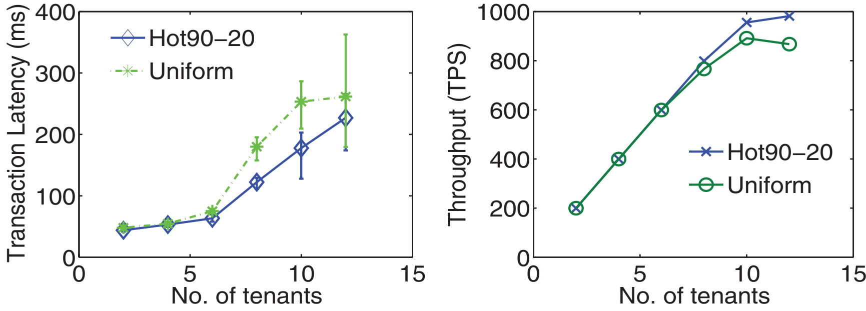


Fig. 10. Evaluating performance when multiple tenants shared an OTM. Each tenant's workload is an instance of YCSB; multiple concurrent YCSB instances simulate a multitenant workload.

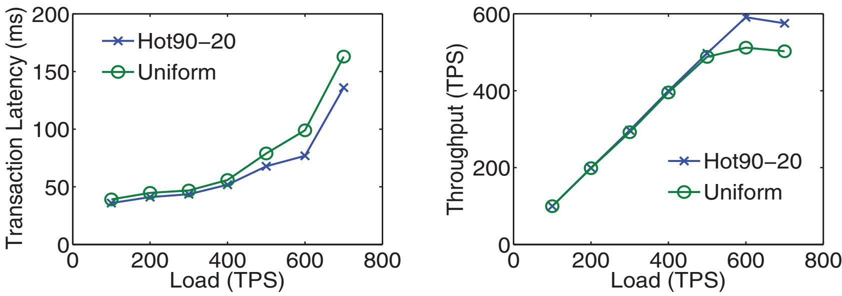


Fig. 11. Performance evaluation using YCSB for large numbers of small tenants sharing resources at an OTM. In this experiment, each tenant's workload is an instance of YCSB. Multiple concurrent YCSB instances simulate a multitenant workload.

We now analyze the effectiveness of ElasTraS in handling large numbers of tenants with small data footprints. In this experiment, each OTM serves an average of 100 tenants each between 50MB to 100MB in size. We evaluate the impact on transaction latency and aggregate throughput per OTM as the load increases (see Figure 11). Even when serving large numbers of tenants that contend for the shared resources at the OTM, a similar behavior is observed when compared to the experiment with a single isolated tenant or one with a small number of tenants. Even though the peak throughput of an OTM is lower when compared to the other scenarios, transaction latencies remain comparable to the experiments with a much smaller number of tenants. The reduction in throughput arises from the overhead due to the large number of network connections, large number of transaction and data managers, and higher contention for OTM resources. This experiment demonstrates that ElasTraS allows considerable consolidation where hundreds of tenants, each with small resource requirement, can be consolidated at a single OTM.

When colocating multiple tenants at an OTM, it is also important that the system effectively isolates a heavily loaded tenant from impacting other co-located tenant. Figure 12 shows the impact of an overloaded tenant on other co-located tenants. In this experiment, we increase the load on a tenant by 5× and 10× the load during normal operation and analyze the impact on the transaction latency of other co-located tenants. The OTM under consideration is serving 10 tenants each with an average load of 25 TPS during normal operation (an aggregate load of 250 TPS at the OTM). The first group of bars (labeled 'Normal') plots the minimum, average, and maximum transaction latency for the tenants during normal operation. The next set of bars plots the latencies when the load on one of the tenants is increased by 5× from 25 TPS to

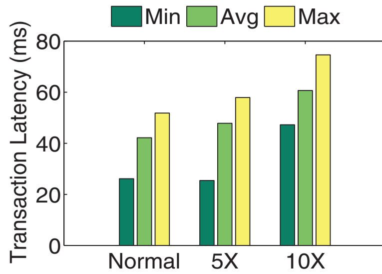
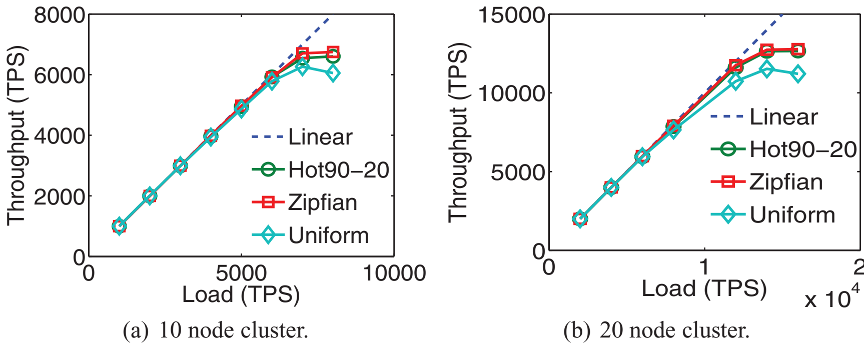


Fig. 12. Impact of a heavily loaded tenant on other co-located tenants.



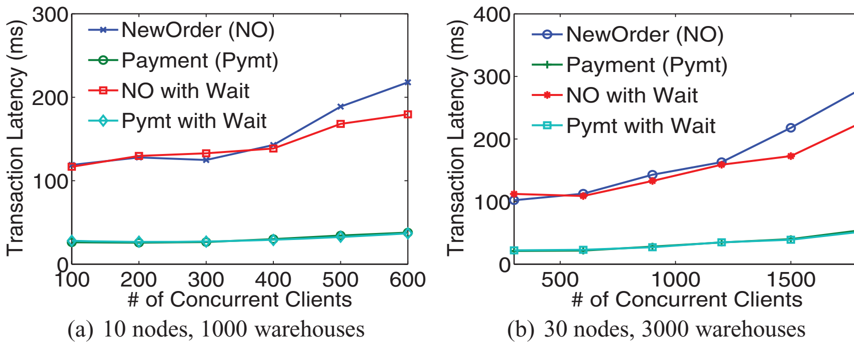
(a) 10 node cluster. (b) 20 node cluster.

Fig. 13. Aggregate system throughput using YCSB for different cluster sizes.

125 TPS. As is evident from Figure 12, this sudden increase in load results in less than 10% increase in the average transaction latency of other tenants co-located at the same OTM. A significant impact on other tenants is observed only when the load on a tenant increases by 10× from 25 TPS to 250 TPS; even in this case, the increase in transaction latency is only about 40%. This experiment demonstrates the benefits of having an independent transaction and data manager per tenant; the impact on co-located tenants arise from contention for some shared resources like the CPU and the cache. ElasTraS's use of the shared process multitenancy model therefore provides a good balance between effective resource sharing and tenant isolation. The experiment also demonstrates that while the provider can place the tenants based on a certain historical load (aggregating to 250 TPS in this case), a sudden increase in workload of one tenant (by 10×) does not have an adverse effect on other co-located tenants. Such spikes in load are hard to predict and the provider will react to such spikes by migrating some tenants out of the node to balance the load; in this experiment we did not migrate the tenants.

7.5. Overall System Performance

Figure 13 plots the aggregate throughput for different cluster sizes and number of tenants served, as the load on the system increases. We use YCSB for this experiment to simulate a multitenant workload. Each OTM on average serves 50 tenants; each tenant database is about 200MB and the cache size is set to 2.4GB. Experiments were performed on EC2 using `c1.xlarge` instances. Since there is no interaction between OTMs and there is no bottleneck, the system is expected to scale linearly with the number of nodes. This (almost) linear scaling is evident from Figure 13 where the peak capacity of the twenty node cluster is almost double that of the 10 node cluster; the 20 node



(a) 10 nodes, 1000 warehouses (b) 30 nodes, 3000 warehouses

Fig. 14. Scaling-out a single tenant by partitioning and scaling to different cluster sizes and varying number of clients. This experiment reports the impact on transaction latency as the load on the system increases.

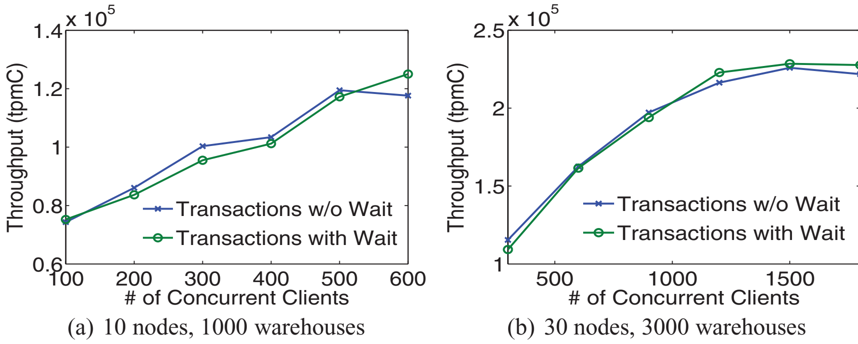
cluster serves about 12K TPS compared to about 6.5K TPS for the 10 node cluster. The choice of colocating a tenant's data within a single partition allows ElasTraS to scale linearly. The reason for the small reduction in per-node throughput in the 20 node cluster compared to the 10 node cluster is, in part, due to the storage layer being co-located with the OTMs, thus sharing some of the resources available at the servers.

The peak throughput of 12K TPS in an ElasTraS cluster of 20 OTMs serving 1000 tenants corresponds to about a billion transactions per day. Though not directly comparable, putting it in perspective, Database.com (the "cloud database" serving the Salesforce.com application platform) serves 25 billion transactions per quarter hosting ∼80,000 tenants [Rao 2010].

7.6. Scaling Large Tenants

In all our previous experiments, we assume that a tenant is small enough to be contained within a single partition and served by a single OTM. In this experiment, we evaluate the scale-out aspect of ElasTraS for a single tenant that grows big. We leverage schema level partitioning to partition a large tenant. The partitions are then distributed over a cluster of OTMs. In this experiment, we vary the number of OTMs from 10 to 30. As the number of OTMs is increased, the size of the database is also increased proportionally and so is the peak offered load. We use the TPC-C benchmark for this experiment and leverage the inherent tree schema for partitioning the database. As per the original TPC-C benchmark specification, about 15% transactions are not guaranteed to be limited to a single partition. In our evaluation, all TPC-C transactions access a single partition. For a cluster with 10 OTMs, the database is populated with 1000 warehouses and the number of clients is increased from 100 to 600 concurrent clients in steps of 100. For a cluster with 30 OTMs, the database size is set to 3000 warehouses (about 2TB on disk), and the number of concurrent clients is varied from 300 to 1800 in steps of 300.

Figure 14 plots the transaction latencies of the NewOrder and Payment transactions and the aggregate system throughput. Figure 15 plots the aggregate throughput of the system in terms of the TPC-C metric of transactions per minute C (tpmC). We used two variants of transactional workload: (i) the clients do not wait between submitting transaction requests, and (ii) the clients wait for a configurable amount of time (10–50ms) between two consecutive transactions. As per TPC-C specifications for generating the workload, each client is associated with a warehouse and issues requests only for that warehouse. As expected, transaction latency increases as the load increases, a behavior similar to the previous experiments. These experiments also demonstrate that in addition to serving large numbers of small tenants, ElasTraS



(a) 10 nodes, 1000 warehouses (b) 30 nodes, 3000 warehouses

Fig. 15. Scaling-out a single tenant by partitioning and scaling to different cluster sizes and varying number of clients. This experiment reports the impact on transaction throughput as the load on the system increases.

can efficiently scale-out large tenants serving thousands of concurrent clients and sustaining throughput of about quarter of a million transactions per minute. Putting it in perspective, the performance of ElasTraS is comparable to many commercial systems that have published TPC-C results (http://www.tpc.org/tpcc/results/tpcc_results.asp).

7.7. Evaluating Albatross

We evaluate the effectiveness of Albatross using the YCSB and TPC-C benchmarks used in the evaluation of ElasTraS. We measure migration cost using the four cost measures we defined earlier: tenant unavailability window, number of failed requests (aborted transactions or failed operations), impact on transaction latency (or response time), and additional data transfer during migration. We compare performance with *stop and migrate* (S&M), a representative off-the-shelf technique, implemented in ElasTraS. In S&M, when the system controller initiates migration, $\mathbb{N}_S$ stops serving $\mathbb{P}_M$, flushes all the cached updates to the DFS, transfers ownership to $\mathbb{N}_D$, and restarts $\mathbb{P}_M$ at $\mathbb{N}_D$. S&M often results in a long unavailability window due to flushing cached updates from committed transactions. An optimization, called *flush and migrate* (F&M), performs a flush while continuing to serve transactions, followed by the final stop and migrate step. We also compare Albatross against an implementation of F&M in ElasTraS.

7.7.1. Experimental Methodology. For the experiments using YCSB, we vary different YCSB parameters to cover a wide spectrum of workloads. These parameters include the percentage of read operations in a transaction, number of operations in a transaction, size of a partition, load offered on a partition, cache size, and the distribution from which the keys accessed are selected. For experiments using the Zipfian distribution, the coefficient is set to 1.0. In a specific experiment, we vary one of these parameters while using the default values for the rest of the parameters; we used the default values provided in Table III. In every experiment, we execute about 12, 000 transactions (about 240 seconds at 50 TPS) to warm up the cache, after which migration is initiated. Clients continue to issue transactions while migration is in progress. We only report the latency for committed transactions; latency measurements from aborted transactions are ignored. Each instance of YCSB corresponds to a single tenant served by one of the live OTMs. As earlier, multiple YCSB instances emulate a multitenant workload.

For the experiments using TPC-C, each tenant database size is about 1 GB and contains 4 TPC-C warehouses. The cache per tenant is set to 500MB. We vary the load on each tenant from 500 tpmC (transactions per minute TPC-C) to 2500 tpmC. Again, multiple TPC-C benchmark instances simulate a multitenant workload.

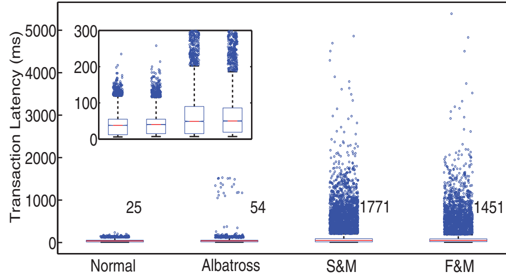


Fig. 16. Transaction latency distribution (box and whisker plot). Inset shows the same graph but with a limited value of the y-axis to show the box corresponding to the 25th and 75th percentiles.

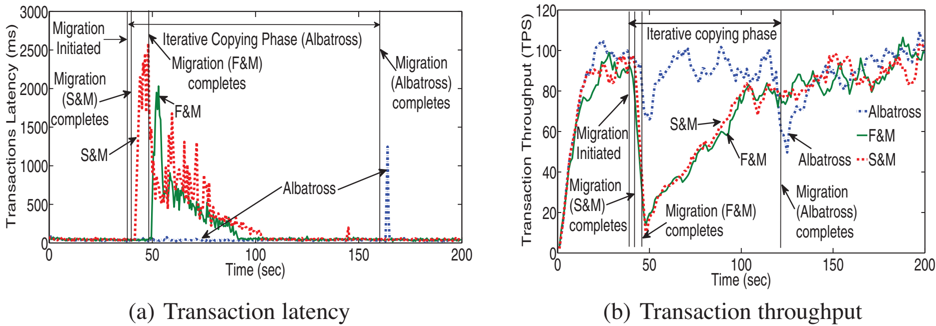
7.7.2. Evaluation Using Yahoo! Cloud Serving Benchmark

Impact on response times and throughput. In the first experiment, we analyze the impact of migration on transaction latency using the default workload parameters just described. We ran a workload of 10,000 transactions, after warming up the cache with another workload of 10,000 transactions; Figure 16 plots the distribution of latency (or response time) of each individual transaction as a box and whisker plot. The four series correspond to the observed transaction latency of an experiment when migration was not initiated (Normal) and that observed when migration was initiated using each of the three different techniques. The inset shows the same plot, but with a restricted range of the y-axis. The box in each series encloses the 25th and 75th percentile of the distribution with the median shown as a horizontal line within each box. The whiskers (the dashed line extending beyond the box) extend to the most extreme data points not considered outliers; the outliers are plotted individually as circles (in blue).[4] The number beside each series denotes the number of outlier data points that lie beyond the whiskers.

As is evident from Figure 16, when migrating a tenant using Albatross, the transaction latencies are almost similar to that in the experiment without migration. A cluster of data points with latency about 1000–1500 ms correspond to transactions that were active during the handover phase that were stalled during the handover and resumed at $\mathbb{N}_D$. On the other hand, both S&M and F&M result in a high impact on transaction latency with about 1500 or more transactions having a latency higher than that observed during normal operation. The high impact on latency for S&M and F&M is due to cache misses at $\mathbb{N}_D$ and contention for the NAS. Since all transactions active at the start of migration are aborted in F&M and S&M, they do not contribute to the increase in latency.

The low impact of Albatross on transaction latency is further strengthened by the experiment reported in Figure 17(a), which plots the average latency observed by the tenants as time progresses; latencies were averaged in disjoint 500 ms windows. The different series correspond to the different migration techniques and are aligned based on the migration start time (about 38 seconds). Different techniques complete migration at different time instances as is shown by the vertical lines; S&M completes at about 40 seconds, F&M completes at around 45 seconds, while Albatross completes at around 160 seconds. The iterative phase for Albatross is also marked in the figure. As is evident, both F&M and S&M result in an increase in latency immediately after migration completes, with the latency gradually decreasing as the cache at $\mathbb{N}_D$ warms up. On the other hand, even though Albatross takes longer to finish, it has negligible impact on latency while migration is in progress. This is because in Albatross, most

[4]The whiskers denote the sampled minimum and sampled maximum (http://en.wikipedia.org/wiki/Sample_minimum).



(a) Transaction latency (b) Transaction throughput

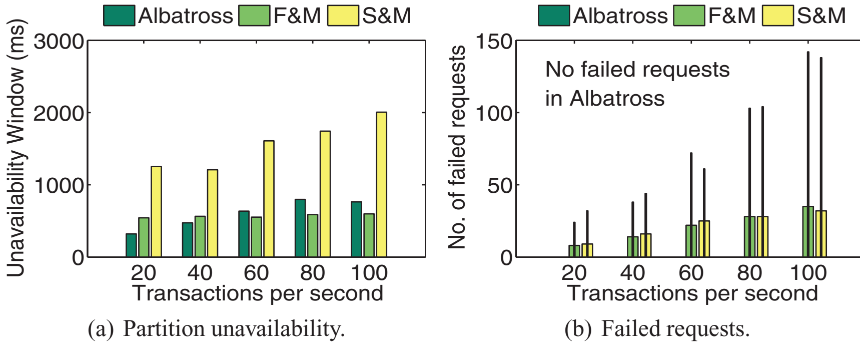
Fig. 17. Transaction latency and throughput observed for different migration techniques. There are 3 different series and each correspond to an execution using one of the discussed migration techniques. All series are aligned at the time at which migration was initiated (about 38 seconds).

of the heavy lifting for copying the state is done by $\mathbb{N}_D$, thus having minimal impact on the transactions executing at $\mathbb{N}_S$. A small spike in latency is observed for Albatross immediately after migration completes, which corresponds to active transactions being stalled temporarily during the final handover phase.

The low impact on latency ensures that there is also a low impact on transaction throughput. Figure 17(b) plots the impact of migration on throughput as time progresses (plotted along the x-axis). The y-axis plots the throughput measured for a second long window. The load is generated by four clients threads, that issue transactions immediately after the previous transaction completes. The different series correspond to different migration techniques. As is evident from the figure, both S&M and F&M result in a high impact on the client throughput due to increased transaction latency after migration, coupled with throughput reduction during the unavailability window. On the other hand, Albatross results in minor throughput fluctuations, once during the first snapshot creation phase and once during the unavailability window in the handover phase; Albatross results in negligible impact during migration since the list of block identifiers in the cache snapshot is maintained incrementally and $\mathbb{N}_D$ performs most of the work done during the synchronization phase.

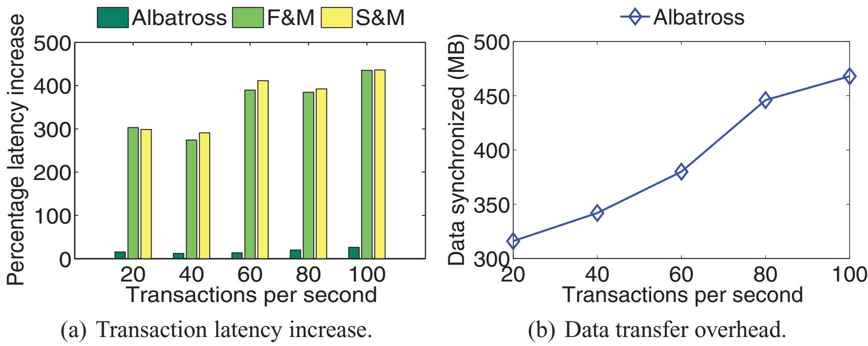
For all the techniques, an impact on transaction latency (and hence throughput) is observed only in a time window immediately after migration completes. Hence, for brevity in reporting the impact on latency, we report the percentage increase in transaction latency for $\mathbb{P}_M$ in the time window immediately after migration, with the base value being the average transaction latency observed before migration. We select 30 seconds as a representative time window based on the behavior of latency in Figure 17(a) where $\mathbb{N}_D$ is warmed up within about 30–40 seconds after the completion of migration. We also measured the percentage increase in latency in the period from start of migration to 30 seconds beyond completion of the respective migration techniques. Since Albatross takes much longer to complete compared to the other techniques and has minimal impact on latency during migration, this measure favors Albatross and unfairly reports a lower increase for Albatross. Therefore, we consider the 30 second window after migration such that all techniques can be evenly evaluated.

Effect of load. Figures 18 and 19 plot migration cost as a function of the load, expressed as transactions per second (TPS), on $\mathbb{P}_M$. As the load on a partition increases (from 20 TPS to 100 TPS), the amount of unflushed changes in the write cache also increases. Hence the unavailability window of S&M increases with load (see Figure 18(a)). But since both Albatross and F&M flush the cache (at least once) before the final phase, they are not heavily impacted by load. The unavailability window of



(a) Partition unavailability. (b) Failed requests.

Fig. 18. Evaluating the impact of transaction load on partition unavailability and number of failed requests. For failed requests 18(b), the wider bars represent aborted transactions and narrower bars represent failed operations.



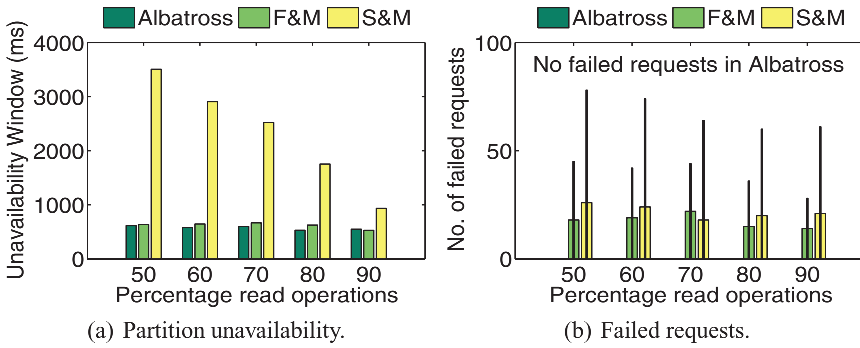
(a) Transaction latency increase. (b) Data transfer overhead.

Fig. 19. Evaluating the impact of transaction load on transaction latency and the amount of data transferred during migration.

Albatross increases slightly since at higher load more transaction state must be copied during the final handover phase. Similarly, a higher load implies more transactions are active at the start of migration; all such active transactions that are aborted in F&M and S&M, thus resulting in a large number of failed requests. Figure 18(b) plots the number of failed requests, where the wider bars represent transactions aborted and the narrower bars represent failed operations. *Albatross does not result in any failed requests since it copies transaction state and allows transactions to resume at* $\mathbb{N}_D$.

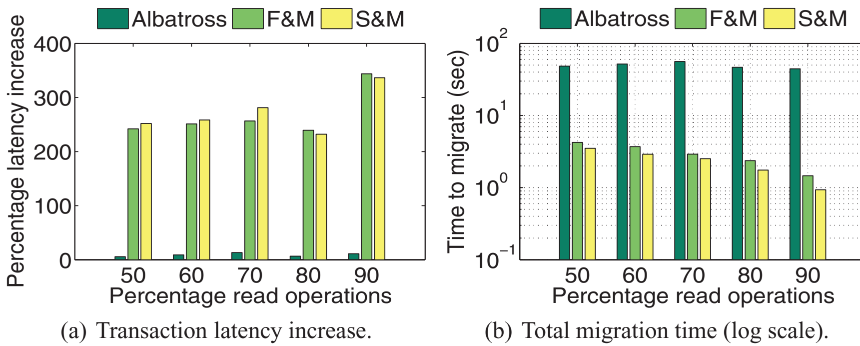
Both F&M and S&M incur a high penalty on transaction latency. The impact on latency increases with load since more read operations incur a cache miss, resulting in higher contention for accessing the NAS (see Figure 19(b)). Albatross results in only 5–15% transaction latency increase (over 80–100 ms average latency) in the 30 second window after migration, while both F&M and S&M result in 300–400% latency increase. Finally, Figure 19(b) plots the amount of data synchronized as a function of load. In spite of the increase in data transmission, this does not adversely affect performance when using Albatross. Both S&M and F&M incur the cost of warming the cache at $\mathbb{N}_D$, which starts with an empty cache. Thus, S&M and F&M incur data transfer overhead of approximately the cache size, that is, 250MB in this experiment.

Effect of read/write ratio. We now present results from experiments varying other parameters of YCSB. Figures 20 and 21 plot the impact of varying the percentage read operations in a transaction; we vary the read percentage from 50 to 90. For an update heavy workload, the write cache has a large amount of unflushed updates that must



(a) Partition unavailability. (b) Failed requests.

Fig. 20. Evaluating the impact of varying the percentage of read operations in a transaction on partition unavailability and number of failed requests.



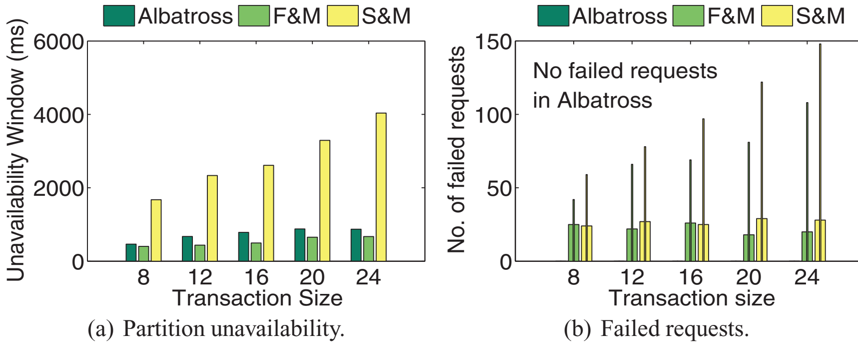
(a) Transaction latency increase. (b) Total migration time (log scale).

Fig. 21. Evaluating the impact of varying the percentage of read operations in a transaction on transaction latency and the time to migrate a partition.

be flushed during migration. As a result, S&M incurs a long unavailability window of about 2–4 seconds; the length of which decreases with a decrease in the percentage of writes (see Figure 20(a)). On the other hand, both F&M and Albatross flush the majority of updates before the final stop phase. Therefore, their unavailability window is unaffected by the distribution of reads and writes. However, since both S&M and F&M do not migrate transaction state, all transactions active at the start of migration are aborted, resulting in a large number of failed requests (see Figure 20(b)). Albatross, on the other hand, does not have any failed requests.

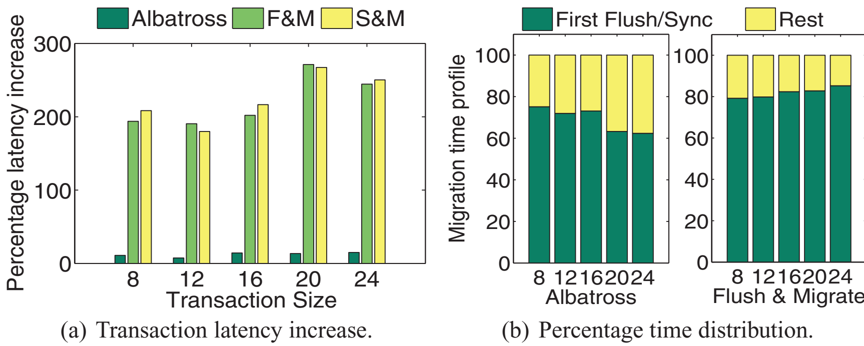
As observed in Figure 21(a), Albatross results in only 5–15% transaction latency increase, while both F&M and S&M incur a 300–400% increase in transaction latency due to the cost of warming up the cache at the destination. Since Albatross warms up the cache at the destination during the iterative phase, the total time taken by Albatross from the start to finish is much longer compared to that of F&M and S&M; S&M is the fastest followed by F&M (see Figure 21(b)). However, since $\mathbb{P}_M$ is still active and serving requests with no impact on transaction latency, this background loading process does not contribute to migration cost from the tenant's perspective. The iterative copying phase transfers about 340MB data between $\mathbb{N}_S$ and $\mathbb{N}_D$, which is about 35% greater that the cache size (250MB). F&M and S&M will also incur network overhead of 250MB resulting from cache misses at $\mathbb{N}_D$ and a fetch from NAS.

Effect of transaction size. Figures 22 and 23 show the effect of transaction size on migration cost; we vary the number of operations in a transaction from 8 to 24. As



(a) Partition unavailability. (b) Failed requests.

Fig. 22. Evaluating the impact of varying the number of operations in a transaction on partition unavailability and number of failed requests.

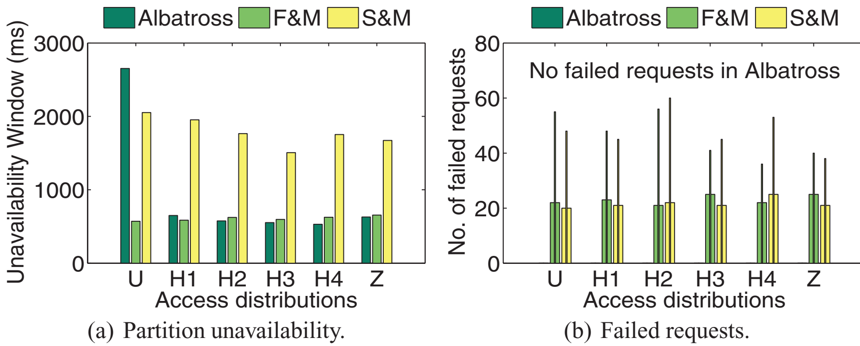


(a) Transaction latency increase. (b) Percentage time distribution.

Fig. 23. Evaluating the impact of varying the number of operations in a transaction on latency and the distribution of time spent in the different migration phases.

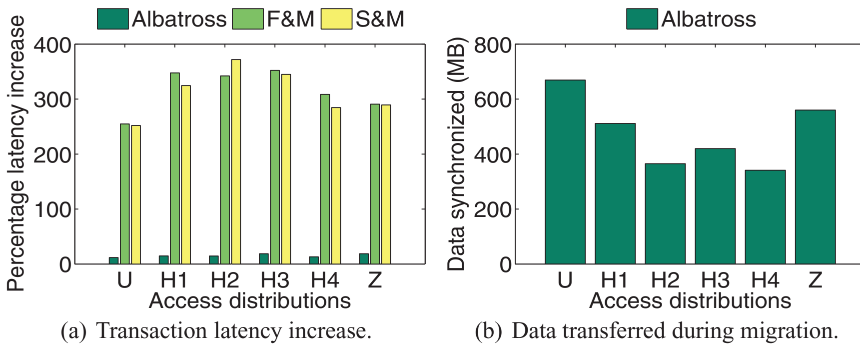
the transaction size increases, so does the number of updates, and hence the amount of un-flushed data in the write cache. Therefore, the unavailability window for S&M increases with increased transaction size (see Figure 22(a)). In this experiment, F&M has a smaller unavailability window compared to Albatross. This is because Albatross must copy the transaction state in the final handover phase, whose size increases with increased transaction size. F&M, on the other hand, aborts all active transactions and hence does not incur that cost. The number of failed requests is also higher for F&M and S&M, since an aborted transaction with more operations result in more work wasted (see Figure 22(b)).

The impact on transaction latency also increases with size since larger transactions have more reads (see Figure 23(a)). This is because the transaction load is kept constant in this experiment and more operations per transactions implies more operations issued on $\mathbb{P}_M$ per unit time. Transaction size also impacts the amount of time spent in the different migration phases; Figure 23(b) shows a profile of the total migration time. As expected, the majority of the time is spent in the first sync or flush, since it results in the greatest amount of data being transferred or flushed. As the number of operations in a transaction increases, the amount of state copied in the later iterations of Albatross also increases. Therefore, the percentage of time spent on the first iteration of Albatross decreases. On the other hand, since the amount of data to be flushed in F&M increases with transaction size, the time taken for the first flush increases.



(a) Partition unavailability. (b) Failed requests.

Fig. 24. Evaluating the impact of different data access distributions on the partition's unavailability and the number of failed requests. U denotes uniform and Z denotes Zipfian. H1–H4 denote hotspot distributions: 90-10, 90-20, 80-10, and 80-20, where x-y denotes $x\%$ operations accessing $y\%$ data items.

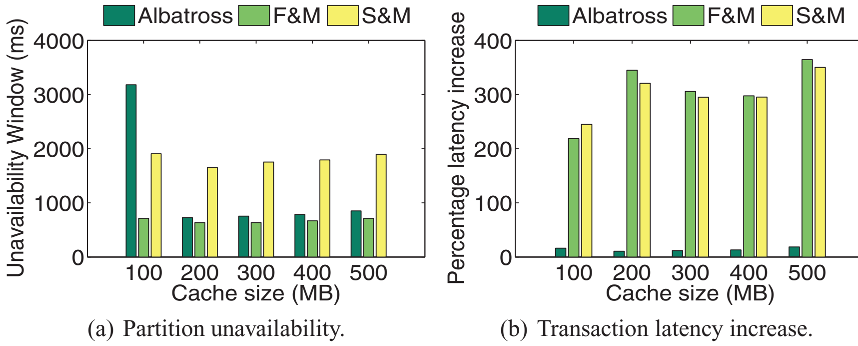


(a) Transaction latency increase. (b) Data transferred during migration.

Fig. 25. Evaluating the impact of different data access distributions on transaction response times and the amount of data transferred during migration. U denotes uniform and Z denotes Zipfian. H1–H4 denote hotspot distributions: 90-10, 90-20, 80-10, and 80-20, where x-y denotes $x\%$ operations accessing $y\%$ data items.

Effect of access distributions. Figures 24 and 25 plot the migration cost as a function of the distributions that determine the data items accessed by a transaction; we experimented with uniform, Zipfian, and four different variants of the hotspot distribution where we vary the size of the hot set and the number of operations accessing the hot set. Since the cache size is set to 25% of the database size, uniform distribution incurs a high percentage of cache misses. As a result, during the iterative copy phase, the database cache changes a lot because of a lot of blocks being evicted and loaded. Every iteration, therefore, results in a significant amount of data being transferred. Albatross tracks the amounts of data transferred in each iteration and this value converges quickly; in this experiment, Albatross converged after 3 iterations. However, the final handover phase has to synchronize a significant amount of data, resulting in a longer unavailability window. Therefore, a high percentage of cache misses results in a longer unavailability window for Albatross. F&M and S&M are, however, not affected since these techniques do not copy the database cache. This effect disappears for skewed workload where as expected, Albatross and F&M have similar unavailability window and S&M has a comparatively longer unavailability window.

Albatross does not result in any failed requests, while the number of failed requests in F&M and S&M is not heavily affected by the distribution (see Figure 24(b)). The uniform distribution results in a higher number of cache misses even at $\mathbb{N}_S$, which offsets



(a) Partition unavailability.

(b) Transaction latency increase.

Fig. 26. Evaluating the impact of varying the cache size allocated to each partition on unavailability and the increase in transaction latency.

the impact of cache misses at $\mathbb{N}_D$. Therefore, the percentage increase in transaction latency for S&M and F&M is lower for the uniform distribution when compared to other access patterns (see Figure 25(a)). Irrespective of the access distribution, Albatross has little impact on latency.

Figure 25(b) plots the amount of data synchronized by Albatross. Following directly from our discussion above, a uniform distribution results in larger amounts of data being synchronized when compared to other distributions. It is however interesting to note the impact of the different hotspot distributions on data synchronized. For H1 and H3, the size of the hot set is set to 10% of the database, while for H2 and H4, the size of the hot set is set to 20%. Since in H1 and H3, a fraction of the cold set is stored in the cache, this state changes more frequently compared to H2 and H4 where the cache is dominated by the hot set. As a result, H1 and H3 result in larger amounts of data synchronized. For the Zipfian distribution, the percentage of data items accessed frequently is even smaller than that in the experiments with 10% hot set, which also explains the higher data synchronization overhead.

Effect of cache size. Figure 26 plots migration cost as a function of the cache size while keeping the database size fixed; the cache size is varied from 100MB to 500MB and the database size is 1GB. Since Albatross copies the database cache during migration, a smaller database cache implies lesser data to synchronize. When the cache size is set to 100MB, the unavailability window of Albatross is longer than that of F&M and S&M (see Figure 26(a)). This behavior is caused by the fact that at 100MB, the cache does not entirely accommodate the hot set of the workload (which is set to 20% of the data items or 200MB), thus resulting in a high percentage of cache misses. This impact of a high percentage of cache misses on unavailability window is similar to that observed for the uniform distribution. However, since the iterations converge quickly, the amount of data synchronized is similar to that observed in other experiments. For cache sizes of 200MB or larger, the hot set fits in the cache, and hence expected behavior is observed. Even though Albatross has a longer unavailability window for a 100MB cache, the number of failed operations and the impact on transaction latency continues to be low. For F&M and S&M, the impact on transaction latency is lower for the 100MB cache because a large fraction of operations incurred a cache miss even at $\mathbb{N}_S$, which somewhat offsets the cost due to cache misses at $\mathbb{N}_D$ (see Figure 26(b)). Number of failed operations and data synchronized show expected behavior.

Figure 27 plots the impact of migration on latency as time progresses. In this experiment, we consider a scenario where the working set of the database does not fit in the cache. The cache size is set to 100MB when using a hotspot distribution where

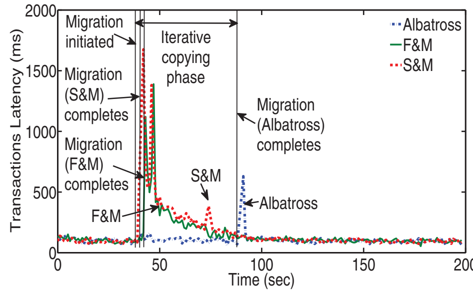
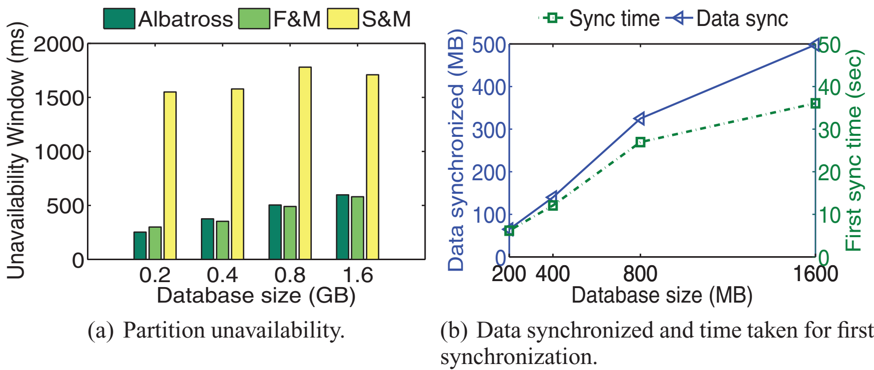


Fig. 27. Impact of migration on transaction latency when working set does not fit in the cache. Even though Albatross results in longer unavailability window, it continues to have low impact on transaction latency.



(a) Partition unavailability.

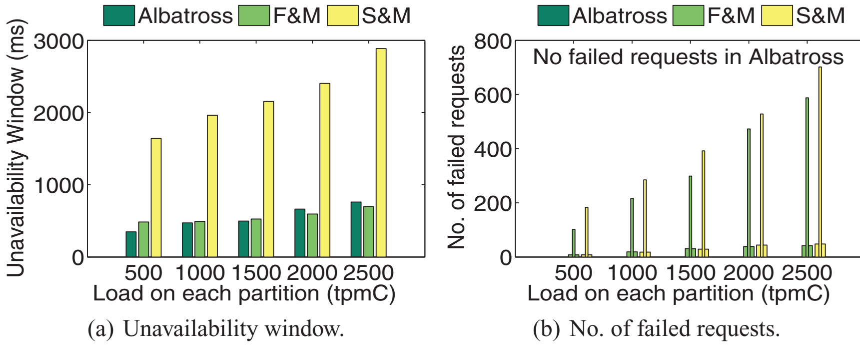
(b) Data synchronized and time taken for first synchronization.

Fig. 28. Evaluating the impact of varying the tenant database size (or size of a partition) on the length of unavailability window and the amount of data synchronized and the time taken to complete synchronizing the initial snapshot.

the hot set is 20% of the database. This experiment confirms our earlier observation that when the working set does not fit in the cache, even though Albatross results in a longer unavailability window, there is minimal impact on transaction latency.

Effect of database size. Figure 28 plots the migration cost as a function of the database size. Since $\mathbb{P}_M$'s persistent data is not migrated, the actual database size does not have a big impact on migration cost. We therefore vary the cache size along with the database size such that the cache is set to 25% of the database size. Since the cache is large enough to accommodate the hot set (we use the default hotspot distribution with the hot set as 20% of the database), the migration cost will be lower for a smaller database (with a smaller cache); the cost increases with an increase in the database size (see Figure 28(a)). Similarly, as the size of the database cache increases, the amount of state synchronized and the time taken for the synchronization also increases (see Figure 28(b)).

7.7.3. Evaluation Using the TPC-C Benchmark. We now evaluate the migration cost using the TPC-C benchmark [Transaction Processing Performance Council 2009] adapted for a multitenant setting. The goal is to evaluate Albatross using complex transaction workloads representing real-life business logic and complex schema. Figure 29 plots the migration cost as the load on each tenant partition is varied; in both subfigures, the y-axis plots the migration cost measures, while the x-axis plots the load on the system. As the load on each partition increases, the amount of data transferred to synchronize state also increases. As a result, the length of the unavailability window



(a) Unavailability window. (b) No. of failed requests.

Fig. 29. Evaluating migration cost using the TPC-C benchmark.

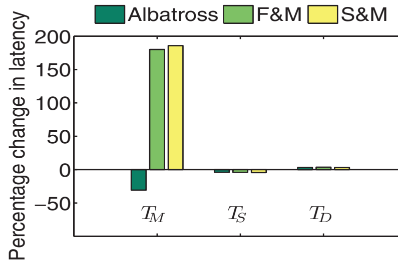


Fig. 30. Impact of migrating tenant T_M from a heavily loaded node to a lightly loaded node. T_S and T_D represent a representative tenant at $\mathbb{N}_S$ and $\mathbb{N}_D$ respectively.

increases with an increase in the load on the tenant (see Figure 29(a)). Furthermore, since the arrival rate of operations is higher at a higher load, more transactions are active at any instant of time, including the instant when migration is initiated. As a result, S&M and F&M result in more failed requests as the load increases. This increase in number of failed requests is evident in Figure 29(b).

From this experiment, it is evident that even with complex transactional workloads, the performance of Albatross is considerably better than S&M and F&M. The behavior of transaction latency increase and amount of data synchronized is similar to previous set of experiments. Albatross incurred less than 15% increase in transaction latency compared to a 300% increase for F&M and S&M, while Albatross synchronized about 700MB data during migration; the cache size was set to 500MB.

7.7.4. Migration Cost During Overload. In all the previous experiments, neither $\mathbb{N}_S$ nor $\mathbb{N}_D$ were overloaded. We now evaluate the migration cost in a system with high load; we use YCSB for this experiment. Figure 30 shows the impact of migrating a tenant from an overloaded node ($\mathbb{N}_S$) to a lightly loaded node ($\mathbb{N}_D$). In this experiment, the load on each tenant (or partition) is set to 50 TPS and the number of tenants served by $\mathbb{N}_S$ is gradually increased to 20 when $\mathbb{N}_S$ becomes overloaded. As the load on $\mathbb{N}_S$ increases, all tenants whose database is located at $\mathbb{N}_S$ experience an increase in transaction latency. At this point, one of the tenants at $\mathbb{N}_S$ is migrated to $\mathbb{N}_D$.

In Figure 30, the y-axis plots the percentage change in transaction latency in the 30 second window after migration; a negative value implies reduction in latency. T_M is the tenant that was migrated, T_S is a tenant at $\mathbb{N}_S$ and T_D is a tenant at $\mathbb{N}_D$. The latency of T_M's transactions is higher when it is served by an overloaded node. Therefore, when T_M is migrated from an overloaded $\mathbb{N}_S$ to a lightly loaded $\mathbb{N}_D$, the transaction latency

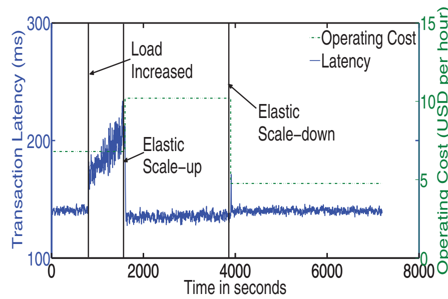


Fig. 31. Evaluating the benefits of elastic scaling.

of T_M should decrease. This expected behavior is observed for Albatross, since it has a low migration cost. However, due to the high performance impact of F&M and S&M, even after T_M is migrated to a lightly loaded $\mathbb{N}_D$, T_M's average transaction latency is higher compared to that at $\mathbb{N}_S$. This further asserts the effectiveness of Albatross for elastic scaling/load balancing when compared to other heavyweight techniques like S&M and F&M. All migration techniques, however, have low overhead on other tenants co-located at $\mathbb{N}_S$ and $\mathbb{N}_D$. This low overhead is evident from Figure 30, where a small decrease in latency of T_S results from lower aggregate load on $\mathbb{N}_S$ and a small increase in transaction latency of T_D results from the increased load at $\mathbb{N}_D$.

7.8. Elasticity and Operating Cost

ElasTraS uses Albatross for low cost live database migration. We now evaluate the effectiveness of Albatross in elastic scaling of the overall system; Figure 31 illustrates this effect as OTMs are added to (or removed from) the system. We vary the overall load on the system during a period of two hours using YCSB. Along the x-axis, we plot the time progress in seconds, the primary y-axis (left) plots the transaction latency (in ms) and the secondary y-axis (right) plots the operating cost of the system (as per the cost of `c1.xlarge` EC2 instances). We plot the moving average of latency averaged over a 10 second window. The number of tenants is kept constant at 200 and the load on the system is gradually increased. The experiment starts with a cluster of 10 OTMs. As the load on the system increases, the transaction latency increases (shown in Figure 31). When the load increases beyond a threshold, elastic scaling is triggered and five new OTMs are added to the cluster. The TM Master rebalances the tenants to the new partitions using Albatross. The ElasTraS client library ensures that the database connections are reestablished after migration. After load-balancing completes, load is distributed on more OTMs resulting in reduced average transaction latency. When the load decreases below a threshold, elastic scaling consolidates tenants to 7 OTMs (also shown in Figure 31). Since the number of OTMs is reduced, the operating cost of the system also reduces proportionately. This experiment demonstrates that elastic scaling using low overhead migration techniques can effectively reduce the system's operating cost with minimal impact on tenant performance.

7.9. Analysis and Lessons Learned

Our evaluation of the ElasTraS prototype using a wide variety of workloads and scenarios show that when the working set fits in the cache, the high cost of accessing the decoupled storage layer in ElasTraS is amortized by caching at the OTM. Since for most high performance OLTP systems, the working set must fit the cache such that the peak throughput is not limited by disk I/O, this architecture will suit most practical OLTP workloads. On the other hand, using a decoupled storage layer allows

migrating partitions with minimal performance impact. Using separate TMs for each tenant partition allows effective isolation between the tenants. Limiting small tenants to a single database process obviates distributed synchronization and lends good scalability to the design. On one hand, the shared process multitenancy model in ElasTraS provides effective resource sharing between tenants allowing consolidation of hundreds of small tenants at an OTM. On the other hand, schema level partitioning allows scaling-out large tenants.

Our experiments revealed an important aspect in the design—whether to share the cache among all the partitions (or tenants) at the OTM or provide an exclusive cache to each tenant. It is a trade-off between better resource sharing and good tenant isolation. In our current implementation, all tenants share the cache at an OTM. This allows better sharing of the RAM while allowing the tenants to benefit from temporal distributions in access—when a tenant requires a larger cache, it can steal some cache space from other tenants that are not currently using it. However, when multiple tenants face a sudden increase in cache requirements, this lower isolation between tenants results in contention for the cache. In such a scenario, some tenants suffer due to a sudden change in behavior of a co-located tenant, which is not desirable. In the future, we plan to explore the choice of having an exclusive cache for each partition. We also plan to explore modeling tenant behavior to determine which cache sharing scheme to use for a given set of tenants co-located at an OTM.

Another important aspect is whether to store an OTM's transaction log on a disk locally attached to the OTM or in the DFS; our current implementation stores the log in the DFS. Storing the log in the DFS allows quick recovery after an OTM failure, since the partitions of the failed OTM can be recovered independent of the failed OTM. However, in spite of using group commit, DFS appends are slower compared to that of a local disk that limits the peak OTM throughput and places a higher load on the network. It is, therefore, a trade-off between performance during normal operation and recovery time after a failure.

Finally, it is also important to consider whether to co-locate an OTM and a storage layer daemon, if possible. Since the OTMs do not result in disk I/O for any locally attached disks, colocating the storage layer with the transaction management layer allows effective resource usage. As CPUs become even more powerful, they will have abundant capacity for both the OTMs as well as storage layer daemons.

During the experiments, we also learnt the importance of monitoring the performance of the nodes in a system, in addition to detecting node failures, especially in a shared (or virtualized) cluster such as EC2. There were multiple instances in an EC2 cluster when we observed a node's performance deteriorating considerably, often making the node unusable. Detecting such unexpected behavior in the cluster and replacing the node with another new node or migrating partitions to other OTMs is important to ensure good performance. We augmented our system models with node-specific performance measures to detect such events and react to them.

8. CONCLUSION

An elastic, scalable, fault-tolerant, self-managing, and transactional multitenant DBMS is critical for the success of cloud application platforms. This article presented two critical technologies towards this goal; we presented ElasTraS, a scale-out transaction processing DBMS, and Albatross, a lightweight live database migration technique for elastic load balancing.

ElasTraS leverages the design principles of scalable key-value stores and decades of research in transaction processing, resulting in a scalable system with transactional and relation semantics. Our design effectively deals with large numbers of small tenants while providing scalability for large tenants. A prototype implementation

deployed on a cluster of commodity servers can efficiently serve thousands of tenants while sustaining aggregate throughput of hundreds of thousands of transactions per second. We further demonstrate that while having performance similar to RDBMSs, ElasTraS can achieve lightweight elasticity and effective multitenancy, two critical facets of a multitenant database for cloud platforms. ElasTraS demonstrates that an appropriate choice of design factors can allow a database management system to be elastic and scalable, provide transactional guarantees, while allowing effective resource sharing in a multitenant system.

Albatross is the first end-to-end solution for live database migration in a decoupled storage architecture such as ElasTraS. Albatross results in minimal performance impact and minimal disruption in service for the tenant whose database is being migrated. Albatross decouples a partition from the OTM *owning* it, and allows the system to routinely use migration as a primitive for elastic load balancing. Since the persistent data is not migrated in a decoupled storage architecture, Albatross focuses on migrating the database cache and the state of the active transactions. This ensures that the destination node of migration starts with a warm cache, thus minimizing the impact of migration. We presented the detailed design and implementation of Albatross. We also demonstrated the effectiveness of Albatross using two OLTP benchmarks, YCSB and TPC-C. Our evaluation showed that Albatross can migrate a live database partition with no aborted transactions, negligible impact on transaction latency and throughput both during and after migration, and an unavailability window as low as 300ms.

In the future, we plan to extend the control layer to add more sophisticated models to improve prediction accuracy, better workload consolidation, and improved partition assignments. This will allow the system to deal with a wide variety of workloads and tenants with different sets of performance and behavioral characteristics. The design of Albatross can also be advanced through multiple optimizations. Since Albatross copies the state of the database cache iteratively, pages that are frequently being updated are copied more than once. Incorporating the access patterns into the decision of which pages to copy during an iteration might reduce the amount of data transferred during migration. It will also be useful to *predict* the migration cost so that the system controller can effectively use migration without violating the SLAs.

ELECTRONIC APPENDIX

The electronic appendix to this article can be the accessed in the ACM Digital Library.

ACKNOWLEDGMENTS

This work has benefited from contributions by Shashank Agarwal in the initial prototype implementation and Shoji Nishimura in implementation of Albatross. The authors would also like to thank Nagender Bandi, Philip A. Bernstein, Pamela Bhattacharya, and Aaron Elmore for their insightful comments on the earlier versions of the article and the anonymous reviewers for suggestions to improve the article.

REFERENCES

AGUILERA, M. K., MERCHANT, A., SHAH, M., VEITCH, A., AND KARAMANOLIS, C. 2007. Sinfonia: A new paradigm for building scalable distributed systems. In *Proceedings of the 21st ACM Symposium on Operating System Principles*. ACM, New York, NY, 159–174.

BAKER, J., BOND, C., ET AL. 2011. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*. 223–234.

BERENSON, B., BERNSTEIN, P., GRAY, J., MELTON, J., O'NEIL, E., AND O'NEIL, P. 1995. A critique of ANSI SQL isolation levels. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, 1–10.

BERNSTEIN, P., REID, C., AND DAS, S. 2011b. Hyder: A transactional record manager for shared Flash. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*. 9–20.

BERNSTEIN, P. A., CSERI, I., ET AL. 2011a. Adapting Microsoft SQL Server for cloud computing. In *Proceedings of the 27th International Conference on Data Engineering. IEEE*, 1255–1263.

BERNSTEIN, P. A. AND NEWCOMER, E. 2009. *Principles of Transaction Processing*, 2nd Ed. Morgan Kaufmann Publishers Inc., San Francisco, CA.

BRADFORD, R., KOTSOVINOS, E., FELDMANN, A., AND SCHIÖBERG, H. 2007. Live wide-area migration of virtual machines including local persistent state. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*. ACM, New York, NY, 169–179.

BRANTNER, M., FLORESCU, D., GRAF, D., KOSSMANN, D., AND KRASKA, T. 2008. Building a database on S3. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, 251–264.

CHANDRASEKARAN, C. AND BAMFORD, R. 2003. Shared cache - The future of parallel databases. In *Proceedings of ICDE*. IEEE, 840–850.

CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. 2006. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. USENIX Association, Berkeley, CA, 205–218.

CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. 2005. Live migration of virtual machines. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation*. USENIX Association, Berkeley, CA, 273–286.

COOPER, B. F., RAMAKRISHNAN, R., SRIVASTAVA, U., SILBERSTEIN, A., BOHANNON, P., JACOBSEN, H.-A., PUZ, N., WEAVER, D., AND YERNENI, R. 2008. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow. 1*, 2, 1277–1288.

COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*. ACM, New York, NY, 143–154.

CURINO, C., JONES, E., POPA, R., MALVIYA, N., WU, E., MADDEN, S., BALAKRISHNAN, H., AND ZELDOVICH, N. 2011. Relational Cloud: A database service for the cloud. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*. 235–240.

CURINO, C., JONES, E., ZHANG, Y., AND MADDEN, S. 2010. Schism: a workload-driven approach to database replication and partitioning. *Proc. VLDB Endow. 3*, 1 48–57.

DAS, S., AGRAWAL, D., AND EL ABBADI, A. 2009. ElasTraS: An elastic transactional data store in the cloud. In *Proceedings of the 1st USENIX Workshop on Hot topics on Cloud Computing*. USENIX Association, Berkeley, CA, 1–5.

DAS, S., AGRAWAL, D., AND EL ABBADI, A. 2010. G-Store: A scalable data store for transactional multi key access in the cloud. In *Proceedings of the 1st ACM Symposium on Cloud Computing*. ACM, New York, NY, 163–174.

DAS, S. NISHIMURA, S., AGRAWAL, D., AND EL ABBADI, A. 2011. Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration. *Proc. VLDB Endow. 4*, 8, 494–505.

DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. 2007. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating System Principles*. ACM, New York, NY, 205–220.

ELMORE, A. J., DAS, S., AGRAWAL, D., AND EL ABBADI, A. 2011. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, 301–312.

ESWARAN, K. P., GRAY, J. N., LORIE, R. A., AND TRAIGER, R. A. 1976. The notions of consistency and predicate locks in a database system. *Comm. ACM 19*, 11, 624–633.

GRAY, J. 1978. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, Springer, 393–481.

GRAY, J. AND REUTER, A. 1992. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA.

HDFS 2010. HDFS: A distributed file system that provides high throughput access to application data. http://hadoop.apache.org/hdfs/.

HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. 2010. ZooKeeper: wait-free coordination for internet-scale systems. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA.

JACOBS, D. AND AULBACH, S. 2007. Ruminations on multi-tenant databases. In *Proceedings of Datenbanksysteme in Business, Technologie und Web*. 514–521.

KUNG, H. T. AND ROBINSON, J. T. 1981. On optimistic methods for concurrency control. *ACM Trans. Data. Syst. 6*, 2, 213–226.

LAMPORT, L. 1998. The part-time parliament. *ACM Trans. Comput. Syst. 16*, 2, 133–169.

LIU, H., JIN, H., LIAO, X., HU, L., AND YU, C. 2009. Live migration of virtual machine based on full system trace and replay. In *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing*. ACM, New York, NY, 101–110.

LOMET, D. B., FEKETE, A., WEIKUM, G., AND ZWILLING, M J. 2009. Unbundling Transaction Services in the Cloud. In *Proceedings of the 4th Biennial Conference on Innovative Data Systems Research*.

NEUVONEN, S., WOLSKI, A., MANNER, M., AND RAATIKKA, V. 2009. Telecommunication application transaction processing (TATP) benchmark description 1.0. http://tatpbenchmark.sourceforge.net/TATP Description.pdf.

PATTERSON, P., ELMORE, A. J., NAWAB, F., AGRAWAL, D., AND EL ABBADI, A. 2012. Serializability, not serial: Concurrency control and availability in multi-datacenter datastores. *Proc. VLDB Endow. 5*, 11, 1459–1470.

PENG, D. AND DABEK, F. 2010. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, Berkeley, CA.

RAO, L. 2010. One database to rule the cloud: salesforce debuts Database.com for the enterprise. http://goo.gl/0BxVq.

SOCKUT, G. H. AND IYER, B. R. 2009. Online reorganization of databases. *ACM Comput. Surv. 41*, 3, 1–136.

TANKEL, D. 2010. Scalability of the Hadoop distributed file system. http://yhoo.it/HDFS.

TRANSACTION PROCESSING PERFORMANCE COUNCIl. 2009. TPC-C benchmark (Version 5.10.1).

THEIMER, M. M., LANTZ, K. A., AND CHERITON, D. R. 1985. Preemptable remote execution facilities for the V-system. In *Proceedings of the 10th ACM Symposium on Operating System Principles*. ACM, New York, NY, 2–12.

TRUSHKOWSKY, B., BODÍK, P., FOX, A., FRANKLIN, M. J., JORDAN, M. I., AND PATTERSON, D. A. 2011. The SCADS director: Scaling a distributed storage system under stringent performance requirements. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies*. USENIX Association, Berkeley, CA.

VO, H. T., CHEN, C., AND OOI, B. C. 2010. Towards elastic transactional cloud storage with range query support. *Proc. VLDB Endow. 3*, 1, 506–514.

VON EICKEN, T. 2008. Righscale blog: Animoto's Facebook scale-up. http://goo.gl/C7Bh.

WEIKUM, G. AND VOSSEN, G. 2001. *Transactional Information Systems: Theory, Algorithms, and the Practice Of Concurrency Control and Recovery*. Morgan Kaufmann Inc., San Francisco, CA.

WEISSMAN, C. D. AND BOBROWSKI, S. 2009. The design of the force.com multitenant Internet application development platform. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, 889–896.

YANG, F., SHANMUGASUNDARAM, J., AND YERNENI, R. 2009. A scalable data platform for a large number of small applications. In *Proceedings of the 4th Biennial Conference on Innovative Data Systems Research*.

Received November 2011; revised June 2012, October 2012; accepted December 2012