Towards an Elastic and Autonomic Multitenant Database

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

ABSTRACT

The success of cloud computing as a platform for deploying web-applications has led to a deluge of applications characterized by small data footprints with unpredictable access patterns. A scalable *multitenant* database management system (DBMS) is therefore an important component of the software stack for platforms supporting these applications. Elastic load balancing and efficient *database migration* techniques are key requirements for effective resource utilization and operational cost minimization. Our vision is a DBMS where multitenancy is viewed as virtualization in the database layer, and elasticity is a first class notion with the same stature as scalability, availability etc. We analyze the various models of database multitenancy, formalize the forms of migration, and identify the design space and research goals for an autonomic and elastic multitenant database.

Categories and Subject Descriptors

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

General Terms

Design

Keywords

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

1. INTRODUCTION

Elasticity, *pay-per-use*, *low upfront investment*, *low time to market*, and *transfer of risks* are some of the enabling features that make cloud computing a ubiquitous paradigm for deploying novel applications which were not economically feasible in a traditional enterprise infrastructure settings. This transformation has resulted in an unforeseen surge in the number of applications being deployed in the cloud. For instance, the Facebook platform[1] has more than a million developers and more than 500K active applications [14].

In addition to the sheer scale of the number of applications deployed, these applications are characterized by high variance in popularity, small data footprints, unpredictable load characteristics, flash crowds, and varying resource requirements. As a result, PaaS providers, such as Joyent [17] or Google App Engine [1], hosting these applications face unprecedented challenges in serving this emerging class of applications and managing their data. Sharing the underlying data management infrastructure amongst a pool of tenants, or databases, is thus essential for efficient use of resources and low cost of operations.

The concept of a multitenant database has been predominantly used in the context of Software as a Service (SaaS). The Salesforce.com model [22] is often cited as a canonical example of this service paradigm. However, it is also interesting to study the various other models of multitenancy in the database tier [16, 19] and their interplay with resource sharing in the various cloud paradigms (IaaS, PaaS, and SaaS). A thorough understanding of these models of multitenancy is crucial for designing effective database management system (DBMS)[2] targeting different application domains. Moreover, irrespective of the multitenancy model or the cloud paradigm, autonomic management of large installations supporting *thousands of tenants*, *tolerating failures*, *dynamic sharding of databases*, with *elastic load balancing* for effective resource utilization and cost optimization are some of the major challenges for multitenant databases for the cloud.

Many large enterprises, in addition to public cloud providers, often host a vast number of databases to serve a variety of disjoint projects or teams. These enterprises can leverage a multitenant cloud platform to consolidate the number of servers dedicated to database hosting. Curino et al. demonstrated, with the consolidation engine Kairos, that the number of database nodes can be consolidated by a factor between 5.5:1 and 17:1 [9]. Large multitenant databases are therefore an integral part of the infrastructure to serve such large number of small applications [19, 22, 23].

Our vision is to develop an architecture of a multitenant DBMS that is *scalable, fault-tolerant, elastic, autonomic*, *consistent*, and supports a *relational data model*. We report on work in progress in designing such a system targeted to serve a large number of small applications typically encountered in a DBMS for a PaaS paradigm or enterprise environment. In this paper, we concentrate on the system level issues related to enabling a multitenant DBMS for

*This work is partly funded by NSF grants III 1018637 and CNS 1053594, and NEC Labs America.

NetDB'11, June 12, 2011, Athens, Greece.

a broader class of systems. We specifically focus on elastic load balancing which ensures high resource utilization and lowers operational costs, live migration of a database as a primitive for elasticity, and generating synthetic tenant loads to provide testing and simulation.

We view multitenancy as analogous to virtualization in the database tier for sharing the DBMS resources. Similar to virtual machine (VM) migration [7], efficient database migration in multitenant databases is an integral component to provide elastic load balancing. Furthermore, considering the scale of the system and the need to minimize the operational cost, the system should be autonomous in dealing with failures and varying load conditions. Migration should therefore be a first class notion in the system having the same stature as scalability, consistency, fault-tolerance, and functionality. This paper serves as step in this direction of building a rich autonomic, elastic, multitenant database. Following are our contributions:

- We categorize the forms of migration, identify metrics for comparing migration forms, discuss off-the-shelf migration techniques and recent work in database live migration.
- We survey and analyze the various multitenancy models in the database tier [16,19] and extend this classification to map to the IaaS, PaaS, and SaaS paradigms.
- We explore our preliminary design in building an autonomic controller for a multitenant database that controls tenant placement and load balancing.

Organization: Section 2 surveys multitenancy models and the challenges associated with each model in enabling elasticity. Section 3 discusses various forms of migration and their associated costs. Section 4 discusses initial work in building a self-managing, elastic, and multitenant database system. Section 5 concludes the paper.

2. DATABASE MULTITENANCY

We now analyze the various database multitenancy models and relate them to the different cloud paradigms to determine the tradeoffs in supporting multitenancy.

2.1 Multitenancy Models

Multitenancy in databases has been prevalent for hosting multiple tenants within a single DBMS while enabling effective resource sharing [2, 16, 19]. Sharing of resources at different levels of abstraction and distinct isolation levels results in various multitenancy models. The three models explored in the past [16] consist of: *shared machine* (also referred to as shared hardware), *shared process*, and *shared table*. SaaS providers like Salesforce.com [22] are a common use cases for database multitenancy, and traditionally rely on the shared table model. The shared process model has been recently proposed in a number of database systems for the cloud, such as RelationalCloud [8], SQLAzure [4], ElasTraS [10]. Nevertheless, some features of cloud computing increases the relevance of the other models. Soror et al. [21] propose using the *shared machine* model to improve resource utilization. To improve understanding of multitenancy, we use the classification recently proposed by Reinwald [19] which uses a finer sub-division (see Table 1). Though some of these models can collapse to the more traditional models of multitenancy. However, the different isolation levels between tenants provided by these models make this classification interesting and helpful for selecting a target classification when building a multitenant database.

# Sharing Mode	Isolation	IaaS	PaaS	SaaS
1. *Shared hardware*	VM	✓	✓	
2. *Shared VM*	OS User		✓	
3. *Shared OS*	DB Instance		✓	
4. *Shared instance*	Database		✓	
5. *Shared database*	Schema		✓	
6. *Shared table*	Row		✓	✓

Table 1: Multitenant database models, how tenants are isolated, and the corresponding cloud computing paradigms.

Shared Hardware
The models corresponding to rows 1–3 share resources at the level of the same machine with different levels of abstractions, i.e., sharing resources at the machine level using multiple VMs (VM Isolation) or sharing the VM by using different user accounts or different database installations (OS and DB Instance isolation). There is no database resource sharing. Rows 1–3 only share the machine resources and thus correspond to the *shared machine* model in the traditional classification. While these models offer strong isolation between tenants, these models come with a cost of increased overhead due to redundant components and a lack of coordination using limited machine resources in an unoptimized way. The lack of coordination is prominent in the case of using a virtual machine for each tenant, row 1, where each tenant behaves as if it has exclusive disk access [8].

Shared Process
Rows 4–5 involve sharing the database process at various isolation levels—from sharing only the installation binary (database isolation), to sharing the database resources such as the logging infrastructure, the buffer pool, etc. (schema isolation), to sharing the same schema and tables (table row level isolation). How a database instance can be isolated between tenants varies between implementation. For example, with MySQL each tenant can be given their own schema with limited user permissions. Rows 4–5 thus span the traditional classes of *shared process* (for rows 4 and 5).[3] Advantages of the shared process model are discussed in section 4.

Shared Table
The shared table model uses a design which allows for extensible data models to be defined by a tenant with the actual data stored in single shared table. The design often utilizes 'pivot tables' to provide rich database functionality such as indexing and joins [2]. While this model offers advantages of maintaining a single database instance, isolating tenants for migration becomes difficult due to shared locking mechanisms. The reliance on consolidated pivot and heap tables could lead to poor performance due to all tenants sharing index structures. Additionally, the shared table model requires that all tenants reside on the same database engine and release (version). This limits specialized database functionality, such spatial or object based, and requires that all tenants use limited subset of functionality. This model is ideal when tenant data requirements follow similar structures or patterns, such as in the case of Force.com offering customizations on a customer relationship database [22].

[3]The *shared instance* model is primarily supported by commercial databases that allows multiple databases (processes) to share a common installation (or binary). Example usage includes running isolated production and test databases. This model can map to both shared machine as well as shared process.

With different forms of multitenancy, components that constitute a tenant vary. We henceforth use the term *cell* to represent all information necessary to serve a tenant. A multitenant database instance consists of thousands of *cells*, and the actual physical interpretation of a *cell* depends on the multitenancy model.

DEFINITION 1. *A* cell *is the self-contained granule representing a tenant in the database.*

At one extreme is the *shared hardware* model which uses virtualization to multiplex multiple VMs on the same machine with strong isolation. Each VM has only a single database process with the database of a single tenant. At the other extreme is the *shared table* model which stores multiple tenants' data on shared tables with the finest level of isolation. In the different models, tenants' data is stored in various forms. For shared machine, an entire VM corresponds to a tenant, while for shared table, a set of rows in a table correspond to a tenant. Thus, the association of a tenant to a database can be more than just the data for the client, and can include metadata or even the execution state. As the level of isolation moves away from shared hardware (row 1), the difficulty of *cell* migration increases; this is due to an increase in shared components, such as transaction managers, buffer pools, etc, which need to have a *cell* partitioned, or isolated, in order to migrate tenant without interrupting co-located tenants. With this understanding of the models and the abstraction corresponding to tenants, we now delve into analyzing the interplay of the different forms of multitenancy and the cloud paradigms.

2.2 Multitenancy for the Cloud

While broad in concept, three main paradigms have emerged for cloud computing: IaaS, PaaS, and SaaS. We now establish the connection between the database multitenancy models with the cloud computing paradigms (Table 1 summarizes this relationship), while analyzing the suitability of the models for various multitenancy scenarios.

IaaS provides the lowest level of abstraction such as raw computation, storage, and networking. Supporting multitenancy in the IaaS layer thus allows much flexibility and different schema for sharing. The *shared hardware* model is therefor best suited in IaaS. A simple multi-tenant system could be built of a cluster of high end commodity machines, each with a small set of virtual machines. Each virtual machine would host a single database tenants. This model provides isolation, security, and efficient migration for the client databases with an acceptable overhead, and is suitable for applications with lower throughput but larger storage requirements.

PaaS providers, on the other hand, provide a higher level of abstraction to the tenants. There exist a wide class of PaaS providers, and a single multitenant database model cannot be a blanket choice. For PaaS systems that provide a single data store API, a *shared table* or *shared instance* could meet data needs for the platform. For instance, Google App Engine uses the shared table model for its data store referred to as MegaStore [3]. However, PaaS systems with the flexibility to support to a variety of data stores, such as AppScale [6], can leverage any multitenant database model.

SaaS has the highest level of abstraction in which a client uses the service to perform a limited and focused task. Customization is typically superficial and workflows or data models are primarily dictated by the service provider. With rigid definitions of data and processes, and restricted access to a data layer through a web service or browser, the service provider has control over how the tenants will interact with a data store. The *shared table* model has thus been successfully used by various SaaS providers [2, 16, 22].

2.3 Elastic and Autonomic Databases

Techniques, or tools, not traditionally found in databases are required to achieve elasticity in a multitenant system. Since tenants are sensitive to the behavior of co-located tenants and usage patterns evolve over time, load balancing tenants is a critical operation. Tenant migration is required to evenly balance load across nodes. With the ability to rebalance tenants, a database must also be able to decide which tenants to migrate and when to migrate. A coordinating process within the system must monitor tenant usage, analyze behavior, and trigger rebalancing when needed. Therefore, a tenant migration mechanism and an autonomic controller are essential features for an elastic multitenant database system; the following two sections discuss the research challenges that arise in building the two components for multitenant databases.

3. FORMS OF DATABASE MIGRATION

The unpredictable usage patterns for the tenants in a multitenant DBMS mandate the need for elasticity. Migration is a key component for elasticity and load balancing, and hence, migration should be supported as a first class notion in any multitenant DBMS. We now classify forms of migration and identify state-of-the-art migration techniques. With this understanding we propose a classification of migration techniques along with a set of metrics to compare the proposed forms.

Downtime is the time a *cell* may be unavailable during migration. *Interruption of service* is the number of *in-flight* transactions of a tenant that fail during migration due to loss of transaction state, or not meeting the transactional requirements. *Required coordination* refers to the extent of coordination needed to initiate as well as complete the migration. Note that in a distributed autonomic system, a component within the DBMS should coordinate migration, i.e. determine when to migrate as well as the source and destination machines, and *cells* to migrate. The overhead in the system can be separated into: *operation overhead* which is the overhead on the DBMS during normal operation that might be incurred to make the system amenable to migration; and *migration overhead* which is the system overhead during migration. The abstract form definitions below identify the goals of migration and are independent of any multitenancy model. Table 2 summarizes these forms of migrations and compares their relative costs.

3.1 Asynchronous migration

Asynchronous migration is an *immediate*, *blocking*[4] migration which relies on a coordinating process to copy the *cell* from a source host to a destination host. The blocking stems from disabling the source during the copy to ensure consistency, resulting in a period of *downtime*. This migration is immediate due to a prompt migration upon initiation. A naive implementation is to stop the database process and copy the database between nodes. Copying can be performed by either a file copy or via a backup and restore process. To minimize impact a database could be flushed and set to read only to allow for some operations during migration. A stale replica (maintained by lazy replication) can be leveraged for migration; here a coordinator process disables the source to replay final updates at the destination. Once the migration has completed, the coordinator will redirect traffic to the destination. As the coordinator has more control over the migration initialization, this form works well for large *cells* with regular periods of inactivity.

[4]Blocking and non-blocking migration refers to potential blocking of client database calls, and not the internal implementation used to achieve the migration.

Form of Migration	Downtime	Interruption of Service	External Coordination	Operation Overhead	Migration Overhead
Asynchronous	Moderate	Moderate	High	None	High
Synchronous	Minimal	Minimal	Moderate	Moderate	Moderate
Live	None	Minimal	Minimal	None	Minimal

Table 2: Summary of the forms of migration and the associated costs.

3.2 Synchronous migration

Synchronous migration is an *eventual*, *non-blocking*[4] migration where a source and destination operate as a tightly coupled cluster. This requires the destination to act as an eager replica of the source, where updates must synchronously occur at the source and destination. If the destination host does not have an up to date replica of the *cell*, the source and destination hosts are configured to run as a synchronized cluster, and the destination gradually acquires a synchronized state by replaying writes that were performed on the source DBMS. Once a stable state is reached, the coordinating process notifies the source host to stop serving the *cell*, and all future connections are sent to the destination host. Many popular RBDMSs have the ability to run in a *master-slave* mode in order to efficiently replicate data across hosts in a cluster. *Synchronous migration* can be achieved using a method proposed by Yang et al. [23] which uses two-phase commit and a read one/write all *master-slave* mode. Even though many DBMSs support a clustered mode off the shelf, changing a lazy replication to a synchronized, or eager replication, often requires short periods of downtime to change server states. Synchronous migration is eventual due to the synchronization period required to complete migration. A minimal amount of downtime and interruption of service may occur while switching the primary master to the destination. The minimal operational overhead originates from the hosts needing to run in a mode which is ready for clustering. The coordinator is responsible for redirecting client connections to the destination host for *cells*.

3.3 Live migration

Live migration is an *immediate*, *non-blocking*[4] migration of a *cell* from a source host directly to a destination host with no *downtime* and minimal *interruption of service*. All client connections are migrated without the need to reconnect. To initiate migration, a coordinating process simply notifies the source host of the destination and relies on the live migration process to independently manage itself.

Several existing techniques can be utilized for database migration. *VM migration* has been thoroughly researched and provides an effective means for live migration of a VM without interrupting processes [5,7,18]. If virtual machines are used for tenant isolation, live virtual machine migration can be leveraged for quick database migration with minimal interruption of service. We were able to migrate a running 1 GB TPC-C in less than 20 seconds on average, with only a 5-10% increase of response time due to the VM overhead. However, this ease of migration and tenant isolation comes at a cost of increased overhead and limited consolidation due to duplicated OS and DB processes [9]. To allow more tenants to be consolidated at a single node, multiple *cells* must share the same database process and VM. In this case, VM migration does not allow fine-grained load balancing of *cells*, and all *cells* contained in a VM must then be migrated together.

Recent research has explored implementing live database migration cognizant of the semantics of the database process. We have proposed Zephyr [12], a technique to migrate a *cell* in a *shared nothing* database architecture with no downtime. Zephyr uses a synchronized *dual* mode where both the source and the destination nodes concurrently execute transactions on the *cell*; the source completes execution of the transactions that were active at the start of migration, while the destination executes new transactions. As the first step of migration, Zephyr copies a wireframe of the database to the destination node. This wireframe consists of the minimal information needed for the destination to start executing transactions but does not include the actual application data stored in the *cell*. The wireframe includes database metadata to authenticate new connections and meta information about the tables and indices. For a database using B+-trees, the wireframe includes only the internal nodes of the tree; the leaf nodes containing the actual data is replaced by sentinels at the destination. Zephyr does not allow structural changes to the indices during migration. Once resources for the *cell* has been initiated, the destination starts executing new transactions while the source continues executing transactions that were active at the start of migration. Pages are pulled by the destination as transactions at the destination access them. Transactions may be aborted at the source when they access a page that has already been migrated and at both nodes when they result in change structural changes to the indices. Once transactions at the source complete, migration completes by pushing pages to the destination.

We have also proposed Albatross [11], a technique to migrate a *cell* in *shared storage* architectures with no aborted transactions and minimal performance impact. In a shared storage architecture, the persistent image of a *cell* is stored in a network addressable storage abstraction and hence does not need migration. Albatross focusses on migrating the database cache and the state of active transactions. In Albatross, the source takes a quick snapshot of a *cell*'s cache and the destination warms up its cache with this snapshot. While the destination initializes its cache, the source continues executing transactions. The destination therefore lags the source. Albatross uses an iterative phase where changes made to the source node's cache are iteratively copied to the destination. When the same amount of data is being copied in consecutive iterations or a maximum number of iterations is reached, transactions are blocked at the source and an atomic handover completes migration. The state of active transactions is copied in the final handover phase to allow them to resume execution at the destination which already has a warmed cache.

Live migration is the ideal candidate for database migration and is the hardest to implement. *Asynchronous migration* is at the other end of the spectrum and the baseline form of migration in system implementations not designed for migration, while *synchronous migration* strikes a middle ground. Ideally, an autonomic DBMS is aware of a *cell*'s service level agreement, and can leverage a migration form that minimizes the impact on performance.

4. AUTONOMIC CONTROLLER DESIGN

Selecting a multitenancy model is a primary consideration when building a multitenant database. With distinct advantages for each

model, the tenant applications and usage requirements should drive this decision. We target the shared instance model for its balance between tenant isolation, component redundancy, performance, and flexibility. The tenant isolation is critical for migration, data privacy, and customizable database settings such as isolation levels, replication methods and recovery techniques. The reduced redundancy allows for better tenant consolidation at a node. Having a single database instance per node that manages all co-located tenants provides the ability for the DBMS to fairly share resource access between tenants and limits performance issues due to a lack of coordination. The shared instance model allows for nodes to run heterogeneous database engines. This is critical for platforms and enterprises that support a variety of engines due to legacy application requirements, acquisitions, licensing needs, or a lack of a strong governance policy. This described environment is our target platform. We first identify a set of goals, for building a flexible multitenant DBMS, that are not met with current 'out of the box' open source database systems. We then describe our ongoing work in building an elastic, autonomic, and multitenant database system to meet our identified goals.

System Goals:

- Each tenant service level agreement are met. This guarantees an uptime percentage and that ample computing resources are available to respond to operations with in a factor of performance on a dedicated node.
- Easily add tenants to the system with out interrupting existing tenants service.
- Identify resource consumption for a specific tenant, despite a single instance pooling resources for multiple tenants.
- Rebalance tenants-to-node placement, or packing, to ensure that all tenants have fair access to resources required to serve requests, despite dynamic changes in tenant usage patterns.
- Minimize the impact of rebalancing on running tenants by factoring migration costs and SLAs.
- Incorporate any DBMS type (relational, spatial, in memory, key-value, object, etc.) to operate as a shared instance engine. This requires the ability to provide usage statistics on tenants and provide hooks, or APIs, to initiate tenant migration.
- Expand, or notify the need to expand, the number of nodes in the system if unable to meet SLAs with the given set of nodes.
- Dynamically partition, or shard, a tenant across multiple nodes if needed.

We extend the architecture described by Yang et al. [23], with a database system composed of a fleet of nodes, each running a single DBMS hosting one or more tenants. A coordinating component, *iLandlord*, controls tenant placement, monitors node health, manages elasticity, and provides database connection information for applications. Each node has a lightweight agent to report usage statistics to the coordinator on demand. In order to achieve scalability *iLandlord* can easily be implemented on a highly available distributed system such as Zookeeper [15]. To prevent bottlenecks by the coordinator, it is important that only meta, or system, data moves through *iLandlord*, and that usage statistics are aggregated at the node level and pulled by the coordinator. While research exists on optimizing sampling techniques to learn accurate models, such as Shivam et al. [20], we leverage regular interval polling to piggyback heartbeats to determine node availability.

iLandlord ensures that each node is not over or under utilized, by regularly polling agents on each node to determine health and resource consumption for all nodes and tenants. Replicas are abstracted and treated as regular tenants for modeling, as their resource consumption, while potentially lower, still affects node performance and needs to be factored in rebalancing. Figure 1 demonstrates interactions in the system, including clients obtaining meta information from *iLandlord* for connection information, clients establishing connections directly to the nodes, and agents reporting usage data to *iLandlord*.

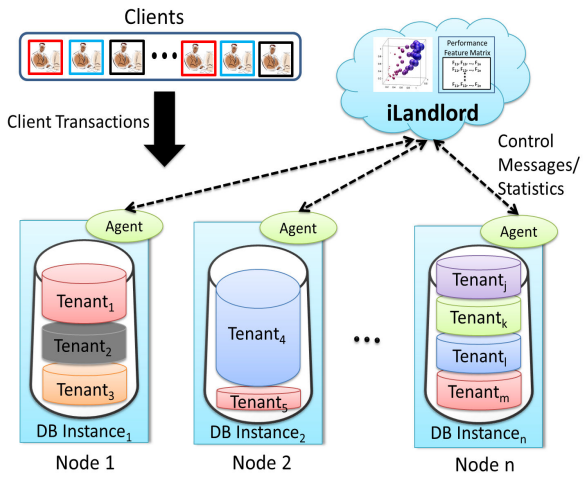

Figure 1: Overview of the system's architecture.

Since the number of tenants hosted may scale into the thousands, *iLandlord* should assist system administration by managing tenant placement to ensure that SLAs are being met while minimizing the number of nodes required. We leverage Machine learning machine learning to automate *iLandlord*'s functionality. Tenants are classified into tenant types based on database agnostic attributes, such as cache hit ratios, transaction length, cache coverage, throughput, update ratios, etc. The intuition is that a tenant type roughly describes resources required to effectively serve a tenant. For example, packing too many disk I/O intensive tenants on the same node can cause service disruption due to disk contention; therefore tenants are identified by operational requirements. Nodes are classified into types based on their resource consumption, such as CPU utilization, disk I/O, memory usage, and network I/O. Based on the training set of nodes, we can determine the degree a node is under or over utilized, and if any rebalancing is required. Based on tenant and node classifications, *iLandlord*, iteratively learns ideal tenant packing schemes by observing the bag of tenant types on healthy nodes[5], and observing the effects of migrating *cells* from over-utilized nodes or migrating *cells* to under utilized nodes. We classify tenants based on a recent history of database attributes to quickly determine when usage patterns change and to identify candidate *cells* to migrate for rebalancing. Additionally, the coordinator determines if the additional, or fewer, nodes are needed in order to meet all requirements; if deployed on an IaaS infrastructure, changes in the number of nodes can be managed programmatically.

As a tenant's resource requirements expand beyond the capacity of a single node, a dynamic partitioning scheme and routing is needed to shard the database based on workloads. While a framework for *iLandlord* is in place, we are working with various orga-

[5]Node health is determined if resource consumption falls within given levels and if all tenants are meeting service level agreements. We allow these thresholds to be parameterized to allow flexibility in how aggressive the system is in node utilization.

nizations to build accurate load generators that reflect realistic multitenant workloads to determine the effectiveness of an autonomic tenant placement controller. Additionally, we are experimenting with modeling of tenants and migration impact to achieve efficient and correct learning of tenant rebalancing.

5. CONCLUSION

Elasticity, and database migration to enable elasticity, is critical for the efficient operation of scalable multitenant databases which drive large cloud platforms. We expanded existing multitenancy models and provided a coupling of these models to the various cloud paradigms. We also formalized the forms of migration classifications, introduced the concept of a *cell* to abstract the tenants and the granule of migration, analyzed the trade-offs associated with the different multitenancy models, and discussed migration techniques.

In summary, we observed that even though a *shared table* is the most common form of multitenancy in a database, some other lesser known models are more suitable for designing an *elastic* multitenant DBMS. Furthermore, even though *shared hardware* provides the best isolation amongst tenants and allows near ideal migration, practical hardware limitations restrict the scale of such a design in terms of number of tenants that can be hosted. Thus, even though virtualization and virtual machine migration [7] have been heavily studied from the systems perspective, the state-of-the-art in virtualization for databases and migration of databases have significant shortcomings which need to be addressed for designing a *scalable, fault-tolerant, elastic*, and *autonomic* multitenant database for scalable cloud platforms.

Projecting into the future, our observation is that migration techniques should be embedded into the fabric of multitenant DBMSs to allow efficient migration as supported by the *shared hardware* model. Much of these shortcomings can be attributed to replicated OS and DB processes which restrict the number of tenants due to hardware limits. A system designed to scale to a large number of clients should minimize redundancy.

The shared instance models minimize this redundancy, and we aim to focus our efforts on these models. Evaluating the trade-offs between the amount of redundancy and the degree of isolation, and their impact on migration is an interesting research problem. Furthermore, the scale of the cloud mandates autonomic migration and management with minimal or no manual intervention and supervision. Major research challenges for autonomic management include modeling load patterns for determining the time for migration, and identifying the *cells* that need migration, thus leading to systematic approaches to database migration to support elasticity.

6. REFERENCES

[1] Google App Engine. `http://code.google.com/appengine/`, 2010.

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

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

[4] P. A. Bernstein, I. Cseri, N. Dani, N. Ellis, A. Kalhan, G. Kakivaya, D. B. Lomet, R. Manner, L. Novik, and T. Talius. Adapting Microsoft SQL Server for Cloud Computing. In *ICDE*, pages 1255–1263, 2011.

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

[6] N. Chohan, C. Bunch, S. Pang, C. Krintz, N. Mostafa, S. Soman, and R. Wolski. AppScale: Scalable and Open AppEngine Application Development and Deployment. In *CloudComp*, 2009.

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

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

[9] C. Curino, E. P. C. Jones, S. Madden, and H. Balakrishnan. Workload-aware database monitoring and consolidation. In *SIGMOD*, 2011.

[10] S. Das, S. Agarwal, D. Agrawal, and A. El Abbadi. ElasTraS: An Elastic, Scalable, and Self Managing Transactional Database for the Cloud. Technical Report 2010-04, CS, UCSB, 2010.

[11] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi. Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud using Live Data Migration. *PVLDB*, 4(8):494–505, May 2011.

[12] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi. Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms. In *SIGMOD*, 2011.

[13] Facebook Developer Platform. `http://developers.facebook.com/`, 2010.

[14] Facebook Statistics. `http://www.facebook.com/press/info.php?statistics`, Retreived Nov 30, 2010.

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

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

[17] Joyent: Enterprise Class Cloud Computing. `http://www.joyent.com/`, 2010.

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

[19] B. Reinwald. Database support for multi-tenant applications. In *IEEE Workshop on Information and Software as Services*, 2010.

[20] P. Shivam, S. Babu, and J. S. Chase. Active sampling for accelerated learning of performance models. In *SysML*, 2006.

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

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

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