Who's Driving this Cloud? Towards Efficient Migration for Elastic and Autonomic Multitenant Databases

Aaron Elmore Sudipto Das Divyakant Agrawal Amr El Abbadi
Department of Computer Science
University of California, 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 but unpredictable access patterns. An autonomic and scalable *multitenant* database management system (DBMS) is therefore an important component of the software stack for platforms supporting these applications. Elastic load balancing is a key requirement for effective resource utilization and operational cost minimization. Efficient techniques for *database migration* are thus essential for elasticity in a multitenant DBMS. Our vision is a DBMS where multitenancy is viewed as virtualization in the database layer, and migration is a first class notion with the same stature as scalability, availability etc. This paper serves as the first step in this direction. We analyze the various models of database multitenancy, formalize the forms of migration, evaluate the off-the-shelf migration techniques, 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*

General Terms

Design

Keywords

Cloud computing, multitenancy, elastic data management, database migration

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 [12]. In addition to the sheer scale of the number of applications developed, 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 [15], 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 is thus essential for efficient use of resources and low cost of operations. Large multitenant databases are therefore an integral part of the infrastructure to serve such large number of small applications [16, 19, 20].

The concept of a multitenant database has been predominantly used in the context of Software as a Service (SaaS). The Salesforce.com model [19] is often cited as a canonical example of this service paradigm. However, various other models of multitenancy in the database tier [14, 16] and their interplay with resource sharing in the various cloud paradigms (IaaS, PaaS, and SaaS) are often overlooked. A thorough understanding of these models of multitenancy is crucial for designing effective database management system (DBMS)[2] targeting different application domains. Furthermore, irrespective of the multitenancy model or the cloud paradigm, autonomic management of large installations supporting *thousands of tenants*, *tolerating failures*, with *elastic load balancing* for effective resource utilization and cost optimization are some of the major challenges for multitenant databases for the cloud. Large distributed *Key-Value* stores – such as Bigtable [6], Dynamo [10], PNUTS [8], and their open source counterparts HBase [13] and Cassandra [5]– are designed to scale to large numbers of concurrent requests using commodity infrastructure of thousands of servers while being elastic and fault-tolerant. Although extremely successful, the *Key-Value* stores' simplified data model, lack of transactional support, and lack of attribute based accesses can result in considerable overhead in re-architecting legacy applications which are predominantly based on RDBMS technology. Additionally, an application with smaller storage requirements (tens of MBs to a few GB) would not utilize these scaling advantages of *Key-Value* stores. Hence, from an individual application's perspective, developers have to trade the lack of features and less portable code for unnecessary scale capabilities. There is therefore a need for a scalable multitenant RDBMS [20].

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 a work in progress in designing such a system targeted to serve a large number of small applications typically encountered in a DBMS for the PaaS paradigm. In this paper, we concentrate on the system level issues related to enabling a multitenant DBMS for a broader class of systems. We specifically focus on elastic load balancing which ensures high resource utilization and lowers operational costs. We

[1]The Facebook platform closely resembles the PaaS paradigm [11].

[2]DBMS refers to the general class of data stores, including non-relational systems while RDBMS refers to a subclass of systems supporting the traditional relational model, like MySQL etc.

# 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 and corresponding cloud computing paradigms.

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. Even though some known commercial solutions, such as Microsoft SQL Azure [18], possess some of the aforementioned goals of a multitenant database, no solution exists in published literature or is supported by open-source systems. This paper serves as the first step in this direction where we analyze the various models of multitenancy in the database tier [14, 16] and extend this classification to map to the IaaS, PaaS, and SaaS paradigms. We also categorize the forms of migration, evaluate the state of the art migration techniques available off-the-shelf, and identify the design space and research goals for an autonomic, elastic, and scalable multitenant database. Additionally, preliminary experimental results are provided for the different forms of migration. While much research has examined the various models of multitenancy [14, 16], to the best of our knowledge this is the first work which analyzes the relationship of different database multitenancy models, formalizes migration for a multitenant database, and evaluate the trade-offs based on a number of measures.

2. CLOUD MULTITENANT DATABASES

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

2.1 Multitenancy for Databases

Multitenancy in databases has been prevalent for hosting multiple tenants within a single DBMS while enabling effective resource sharing [3, 14, 16]. SaaS providers like Salesforce.com [19] are the most common use cases for database multitenancy. Sharing of resources at different levels of abstraction and distinct isolation levels results in various multitenancy models. The three models explored in the past [14] consist of: *shared machine*, *shared process*, and *shared table*. The Salesforce.com model uses shared table, while the other models of multitenancy have not been widely used; Das et al. [9] propose a design that uses the *shared process* model, and Soror et al. [17] propose using the *shared machine* model to improve resource utilization. Nevertheless, some features of cloud computing increases the relevance of the other models. To improve understanding of multitenancy, we use the classification recently proposed by Reinwald [16] which uses a finer sub-division (see Table 1). Though some of these models collapse to the more traditional models of multitenancy. However, the different isolation levels between tenants provided by these models make this classification interesting.

The models corresponding to rows 1–3 share resources at the level of the same machine with different levels of abstractions, i.e., whether 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, and the database instances remain independent. Rows 1–3 only share the machine resources and thus correspond to the *shared machine* model in the traditional classification. On the other hand, rows 4–6 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). Rows 4–6 thus span the traditional classes of *shared process* (for rows 4 and 5)[3] and *shared table* (row 6). Shared tables typically involve 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. 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 few 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. To span this spectrum, we define a common logical concept of *cell*:

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

We henceforth use the term *cell* to represent all information that is sufficient to serve a tenant. A multitenant database instance consists of thousands of *cells*, and the actual physical interpretation of a *cell* depends on the form of multitenancy. 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 however 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 few database tenants.

[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 based on the implementation.

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 a MegaStore [4]. However, PaaS systems with the flexibility to support to a variety of data stores, such as AppScale [2], 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 [3, 14, 19].

3. FORMS OF 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 migrations and identify state of the art migration techniques. But before we delve into the details of migration, we define the granule for migration. Recall that a *cell* corresponds to a tenant in the DBMS, and multiple *cells* will share some common system resources. When migrating, the system will therefore migrate one or more *cells*. For convenience we refer to a set of one or more *cells* for migration as a ***colony***.

An autonomic DBMS needs multiple forms of migration to support the variety of SLAs and applications found in the cloud. 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 an 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 *colonies* 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 to any multitenancy model.

Live migration involves a *seamless*, *instantaneous* migration of a *colony* from a source host directly to a destination host. 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.

Synchronous migration is a *real time*, *non-blocking* migration[4] where a source and destination operate as a synchronized cluster. While the destination host gradually acquires a synchronized state, reads and writes are performed on the host DBMS. Once a stable state is reached, the coordinating process notifies the source host to stop serving the *colony*, and all future connections are sent to the destination host. A minimal amount of downtime and interruption of service may occur while switching hosts. 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 of all *cells* of the migrated *colony* to the destination host.

Asynchronous migration is an *eventual*, *blocking* migration[4] which relies on a coordinating process to copy the *colony* from a source host to a destination host. The coordinator will track changes to the source during migration and potentially block transactions to ensure consistency. This might cause periods of downtime and service interruption. 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.

Live migration is the utopian world for database migration and is the most desirable as well as hardest to implement form of migration. *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. An elastic DBMS must at least support synchronous migration to minimize the impact of migration on the tenants and clients, while the goal will be to approach *live migration*. Table 2 summarizes these forms of migrations and compares their relative costs.

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 [7]. A lightweight virtual machine running a database process can use *live migration* for database migration. 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 via the method proposed by Yang et al. [20] which uses two-phase commit and a read one/write all *master-slave* mode. However, while some commercial DBMSs support a clustered mode off the shelf, running this mode for open source alternatives would require some scripting and short periods of downtime to change server states. Without a coordinating process, flushing table locks and a database copy tool could be used for consistent *asynchronous migration*. Additionally, a *master-slave* replication without two-phase commit could be used to *asynchronously migrate cells* between hosts. In both synchronous and asynchronous migration, a coordinating process will need to route application connections to the updated hosts and potentially needs to make configuration changes on both hosts to reflect the updated state. In contrast, live VM migration transfers client connections during migration.

4. MULTITENANCY AND MIGRATION

Having defined the multitenant models and forms of migration, we now examine some strengths and weaknesses of the different multitenancy models for supporting migration. We also evaluate the applicability of "off the shelf" migration solutions and provide preliminary experimental results for the various forms.

4.1 Shared Table Migration

It is important to note that while the shared table model does offer many advantages and is the best fit for SaaS applications, mi-

[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
Live	None	Very Minimal	Minimal	Low/Moderate	Minimal
Synchronous	Minimal	Minimal	Moderate	Minimal	Moderate
Asynchronous	Moderate	Moderate	High	None	High

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

gration is extremely challenging and any potential method is coupled to the implementation. In systems without elastic migration, isolation controls must be present to prevent high load tenants from degrading system wide performance. Apex, a proprietary multitenant aware programming language implemented by SalesForce, is an example of delegating resource control to an external component [19]. Additionally, many shared table models use tenant identifiers or entity keys as a natural partition to control physical data placement [6, 19]. It is our belief that data placement and resource management decisions should be encapsulated within the DBMS in order to support robust migration for an elastic multitenant system. Lastly, the popular approach of using a 'single' heap storage for all tenants [3, 19] makes isolating a *cell* for migration extremely challenging. Without the ability to isolate a *cell*, none of the techniques in Section 3 allow for migration. This leaves efficient migration of shared tables an open problem.

4.2 Shared Hardware Migration

Using the shared hardware model gives flexibility for migration options. The ease, isolation, and performance of VM migration makes this an ideal form of migration. Additionally, using VM migration abstracts the complexity of managing memory state, file migration and networking configuration. Live migration only requires Xen be configured to accept migrations from a specified host. Using Xen and a 1 Gbps network switch, we were able to migrate an Ubuntu image running MySQL with a 1 GB TPC-C database between hosts on average in only 20 seconds. Running the TPC-C benchmark in a standard OS versus a virtual OS, we observed an average increase of response times by 5-10%. In summary, the major benefits of using shared hardware and live migration for *cell* migration are:

- Isolation, ease of use, and no network configuration updates.
- No downtime and minimal interruption of service.
- Determining when and what to migrate becomes a simple examination of resources available compared to resources consumed.

Disadvantages of using this multitenancy model:

- Potential need for shared filesystem or disk snapshots.
- Low to moderate operation overhead.
- If a VM is a *cell* then redundancy is high due to replicated OS and DB services across VMs.
- Number of tenants is coupled with the amount and quality of hardware available. Requires more horizontal growth.

4.3 Shared Instance Migration

For shared instance, we can have near independent DB instances spawned from a single binary installation. In this model each *cell* has a unique database process, resulting in replicated resources such as logging, caching, and query optimizers. While redundant, this replication simplifies *cell* migration by having isolation at the database level. This isolation provides the ability to leverage *synchronized migration*. As stated before, this migration form is not supported by all RDBMS implementations off the shelf. It is worth noting that the concepts outlined are applicable for the shared OS and VM models with a slight increase in configuration complexity and redundancy. The synchronized replication implementation by Yang et al. [20] only incurred a 5-25% overhead compared to stand alone database. As the amount of time required for the synchronized mode is limited, this overhead is acceptable. A non-synchronized master-slave configuration replicated a 1 GB TPC-C database in 500 seconds on average. If replication is desired for durability, this form can be leveraged for handling failures along with migration [20]. Primary benefits of using shared DB instance and synchronous migration are:

- Reduced redundancy compared to shared hardware.
- Good for DBMSs not optimized for multi-cores (each core can focus on a low number of databases).
- Minimal downtime, interruption of service, and operation overhead.

Disadvantages of using this multitenancy model and synchronous migration:

- Network configuration changes to reflect the new location of a *colony*.
- Isolation and impact on *cells* not in the migration *colony* will be implementation dependent.

4.4 Shared Database Migration

Using a single database instance and having *cells* isolated on schemas (or tablespaces supported in Oracle), potentially removes the ability to use synchronized database replication as these migrations occur at the database level within the process. Since the *colony* being migrated might be a proper subset of all *cells* located on a database, the migration process will need to be selective in building the *colony* to be migrated. Therefore, *asynchronous migration* is an ideal candidate for shared database migration. With state information, such as transactions logs, shared amongst *cells* connected clients may be affected by the locking required to achieve a consistent copy. To test the impact of asynchronous migration we created a shared MySQL database with five small TPC-C databases (200 MB) in different schemas. TPC-C load testing applications were run against four of the databases to measure change in response time during the asynchronous replication of the fifth database. With a moderately high load (five threads per *cell* testing maximal throughput), we observed average response times degrade by 10% during a transaction wrapped online copy to a destination host. Attention must be paid to the locking schemes used; For example MySQL allows for all tables to be locked during a copy, which results in blocking non-migrating *cells* for the duration of the copy. Without any locking or transactions the copy process had minimal impact on performance, but without a coordinating process this can lead to consistency problems. This model's reduced replication's impact on migration isolation and consistency, demonstrates the need for an autonomic migration process. A naive asynchronous migration using an export and import tool (MySQLDump), took an average 920 seconds for a 1 GB TPC-C database. It is worth noting that for the naive migration 95% of the time was spent on importing the database, with only 5% of the time for exporting and copying. In summary, benefits of using shared database and asynchronous migration include:

- Reduced redundancy allows for a higher number of *cells* to be hosted on a single host.
- No operational overhead.

Disadvantages of using shared database and asynchronous migration:

- Requires greater coordination and a longer migration window.
- Moderate downtime and interruption of service.
- Locking and heavy reads during migration may degrade performance of non-migrating *cells* on the same physical disk.

5. DISCUSSION AND FUTURE WORK

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 some preliminary techniques for migration of a *colony* using "off-the-shelf" technology. 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, while minimizing the redundancy shortcomings observed here. 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 DB and 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. At this time it is uncertain if the ideal multitenancy model for an elastic DBMS is shared instance, shared DB, or a marriage of the two. 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 for supporting elasticity.

Acknowledgements

The authors would like to thank Ceren Budak and Shoji Nishimura for their insightful comments on the earlier versions of the paper which has helped in improving this paper. This work is partially supported by NSF Grants IIS-0744539 and IIS-0847925.

6. REFERENCES

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

[2] AppScale: Open Source Google AppEngine. `http://appscale.cs.ucsb.edu/`, 2010. 3

[3] S. Aulbach, T. Grust, D. Jacobs, A. Kemper, and J. Rittinger. Multi-tenant databases for software as a service: schema-mapping techniques. In *SIGMOD*, pages 1195–1206, 2008. 2, 3, 4

[4] R. Barrett. Transactions Across Datacenters. Google IO, May 2009. 3

[5] Cassandra: A highly scalable, eventually consistent, distributed, structured key-value store, 2010. `http://incubator.apache.org/cassandra/`. 1

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*, pages 205–218, 2006. 1, 4

[7] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *NSDI*, pages 273–286, 2005. 2, 3, 5

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

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

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

[11] Facebook Developer Platform. `http://developers.facebook.com/`, 2010. 1

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

[13] HBase: Bigtable-like structured storage for Hadoop HDFS, 2010. `http://hadoop.apache.org/hbase/`. 1

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

[15] Joyent: Enterprise Class Cloud Computing. `http://www.joyent.com/`, 2010. 1

[16] B. Reinwald. Database support for multi-tenant applications. In *IEEE Workshop on Information and Software as Services*, 2010. 1, 2

[17] 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. 2

[18] Microsoft SQL Azure Database. `http://www.microsoft.com/windowsazure/sqlazure/`, 2010. 2

[19] C. D. Weissman and S. Bobrowski. The design of the force.com multitenant internet application development platform. In *SIGMOD*, pages 889–896, 2009. 1, 2, 3, 4

[20] F. Yang, J. Shanmugasundaram, and R. Yerneni. A scalable data platform for a large number of small applications. In *CIDR*, 2009. 1, 3, 4