# Database Scalability, Elasticity, and Autonomy in the Cloud*

## [Extended Abstract]

Divyakant Agrawal, Amr El Abbadi, Sudipto Das, and Aaron J. Elmore

Department of Computer Science
University of California at Santa Barbara
Santa Barbara, CA 93106, USA
{agrawal,amr,sudipto,aelmore}@cs.ucsb.edu
http://www.cs.ucsb.edu/~dsl

**Abstract.** Cloud computing has emerged as an extremely successful paradigm for deploying web applications. *Scalability*, *elasticity*, *pay-per-use* pricing, and *economies of scale* from large scale operations are the major reasons for the successful and widespread adoption of cloud infrastructures. Since a majority of cloud applications are data driven, database management systems (DBMSs) powering these applications form a critical component in the cloud software stack. In this article, we present an overview of our work on instilling these above mentioned "cloud features" in a database system designed to support a variety of applications deployed in the cloud: designing scalable database management architectures using the concepts of *data fission* and *data fusion*, enabling lightweight elasticity using low cost live database migration, and designing intelligent and autonomic controllers for system management without human intervention.

**Keywords:** Cloud computing, scalability, elasticity, autonomic systems.

## 1 Introduction

The proliferation of technology in the past two decades has created an interesting dichotomy for users. There is very little disagreement that an individual's life is significantly enriched as a result of easy access to information and services using a wide spectrum of computing platforms such as personal workstations, laptop computers, and handheld devices such as smart-phones, PDAs, and tablets (e.g., Apple's iPads). The technology enablers are indeed the advances in networking and the Web-based service paradigms that allow users to obtain information and data-rich services at any time blurring the geographic or physical distance between the end-user and the service. As network providers continue to improve the capability of their wireless and broadband infrastructures, this paradigm will continue to fuel the invention of new and imaginative services that simplify and enrich the professional and personal lives of end-users. However, some will argue that the same technologies that have enriched the lives of

* This work is partly funded by NSF grants III 1018637 and CNS 1053594 and an NEC Labs America University relations award.

the users, have also given rise to some challenges and complexities both from a user's perspective as well as from the service provider or system perspective. From the user's point-of-view, the users have to navigate through a web of multiple compute and storage platforms to get their work done. A significant end-user challenge is to keep track of all the applications and information services on his/her multiple devices and keep them synchronized. A natural solution to overcome this complexity and simplify the computation- and data-rich life of an end-user is to push the management and administration of most applications and services to the network core. The justification being that as networking technologies mature, from a user's perspective accessing an application on his/her personal device will be indistinguishable from accessing the application over the broadband wired or wireless network. In summary, the current technology trend is to host user applications, services, and data in the network core which is metaphorically referred to as the *cloud*.

The above transformation that has resulted in user applications and services being migrated from the user devices to the cloud has given rise to unprecedented technological and research challenges. Earlier, an application or service disruption was typically confined to a small number of users. Now, any disruption has global consequences making the service unavailable to an entire user community. In particular, the challenge now is to develop server-centric application platforms that are available to a virtually unlimited number of users $24 \times 7$ over the Internet using a plethora of modern Web-based technologies. Experiences gained in the last decade from some of the technology leaders that provide services over the Internet (e.g., Google, Amazon, Ebay, etc.) indicate that application infrastructures in the cloud context should be highly *reliable*, *available*, and *scalable*. Reliability is a key requirement to ensure continuous access to a service and is defined as the probability that a given application or system will be functioning when needed as measured over a given period of time. Similarly, availability is the percentage of times that a given system will be functioning as required. The scalability requirement arises due to the constant load fluctuations that are common in the context of Web-based services. In fact these load fluctuations occur at varying frequencies: daily, weekly, and over longer periods. The other source of load variation is due to unpredictable growth (or decline) in usage. The need for scalable design is to ensure that the system capacity can be augmented by adding additional hardware resources whenever warranted by load fluctuations. Thus, scalability has emerged both as a critical requirement as well as a fundamental challenge in the context of cloud computing.

In the context of most cloud-based application and service deployments, *data* and therefore the *database management system* (DBMS) is an integral technology component in the overall service architecture. The reason for the proliferation of DBMS, in the cloud computing space is due to the success DBMSs and in particular Relational DBMSs have had in modeling a wide variety of applications. The key ingredients to this success are due to many features DBMSs offer: overall functionality (modeling diverse types of application using the relational model which is intuitive and relatively simple), consistency (dealing with concurrent workloads without worrying about data becoming out-of-sync), performance (both high-throughput, low-latency and more than 25 years of engineering), and reliability (ensuring safety and persistence of data in the presence of different types of failures). In spite of this success, during the past decade

there has been a growing concern that DBMSs and RDBMSs are not *cloud-friendly*. This is because, unlike other technology components for cloud service such as the web-servers and application servers, which can easily scale from a few machines to hundreds or even thousands of machines), DBMSs cannot be scaled very easily. In fact, current DBMS technology fails to provide adequate tools and guidance if an existing database deployment needs to scale-out from a few machines to a large number of machines.

At the hardware infrastructure level, the need to host scalable systems has necessitated the emergence of large-scale data centers comprising thousands to hundreds of thousands of compute nodes. Technology leaders such as Google, Amazon, and Microsoft have demonstrated that data centers provide unprecedented economies-of-scale since multiple applications can share a common infrastructure. All three companies have taken this notion of *sharing* beyond their internal applications and provide frameworks such as Amazon's AWS, Google's AppEngine, and Microsoft Azure for hosting third-party applications in their respective data-center infrastructures (viz. the clouds). Furthermore, most of these technology leaders have abandoned the traditional DBMSs and instead have developed proprietary data management technologies referred to as *key-value stores*. The main distinction is that in traditional DBMSs, all data within a database is treated as a "whole" and it is the responsibility of the DBMS to guarantee the consistency of the entire data. In the context of key-value stores this relationship is completely severed into key-values where each entity is considered an independent unit of data or information and hence can be freely moved from one machine to the other. Furthermore, the atomicity of application and user accesses are guaranteed only at a single-key level. Key-value stores in conjunction with the cloud computing frameworks have worked extremely well and a large number of web applications have deployed the combination of this cloud computing technology. More recent technology leaders such as Facebook have also benefited from this paradigm in building complex applications that are highly scalable.

The requirement of making web-based applications *scalable* in cloud-computing platforms arises primarily to support virtually unlimited number of end-users. Another challenge in the cloud that is closely tied to the issue of scalability is to develop mechanism to respond to sudden load fluctuations on an application or a service due to demand surges or troughs from the end-users. Scalability of a system only provides us a guarantee that a system can be scaled up from a few machines to a larger number of machines. In cloud computing environments, we need to support additional property that such scalability can be provisioned dynamically without causing any interruption in the service. This type of dynamic provisioning where a system can be scaled-up dynamically by adding more nodes or can be scaled-down by removing nodes is referred to as *elasticity*. Key-value stores such as BigTable and PNUTS have been designed so that they can be elastic or can be dynamically provisioned in the presence of load fluctuations. Traditional database management systems, on the other hand, are in general intended for an enterprise infrastructure that is statically provisioned. Therefore, the primary goal for DBMSs is to realize the highest level of performance for a given hardware and server infrastructure. Another requirement that is closely related to scalability and elasticity of data management software is that of *autonomic management*. Traditionally, data administration is a highly manual task in an enterprise setting where a highly-trained

engineering staff continually monitor the health of the overall system and take actions to ensure that the operational platform continues to perform efficiently and effectively. As we move to the cloud-computing arena which typically comprises data-centers with thousands of servers, the manual approach of database administration is no longer feasible. Instead, there is a growing need to make the underlying data management layer *autonomic* or *self-managing* especially when it comes to load redistribution, scalability, and elasticity. This issue becomes especially acute in the context of *pay-per-use* cloud-computing platforms hosting multi-tenant applications. In this model, the service provider is interested in minimizing its operational cost by consolidating multiple tenants on as few machines as possible during periods of low activity and distributing these tenants on a larger number of servers during peak usage.

Due to the above desirable properties of key-value stores in the context of cloud computing and large-scale data-centers, they are being widely used as the data management tier for cloud-enabled Web applications. Although it is claimed that atomicity at a single key is adequate in the context of many Web-oriented applications, evidence is emerging that indicates that in many application scenarios this is not enough. In such cases, the responsibility to ensure atomicity and consistency of multiple data entities falls on the application developers. This results in the duplication of *multi-entity synchronization mechanisms* many times in the application software. In addition, as it is widely recognized that concurrent programs are highly vulnerable to subtle bugs and errors, this approach impacts the application reliability adversely. The realization of providing atomicity beyond single entities is widely discussed in developer blogs [28]. Recently, this problem has also been recognized by the senior architects from Amazon [23] and Google [16], leading to systems like MegaStore that provide transactional guarantees on key-value stores [3].

Cloud computing and the notion of large-scale data-centers will become a pervasive technology in the coming years. There are two major technology hurdles that we confront in deploying applications on cloud computing infrastructures: DBMS scalability and DBMS security. In this paper, we will focus on the problem of making DBMS technology cloud-friendly. In fact, we will argue that the success of cloud computing is critically contingent on making DBMSs scalable, elastic, and autonomic, which is in addition to the other well-known properties of database management technologies: high-level functionality, consistency, performance, and reliability. This paper summarizes the current state-of-the-art as well as identifies areas where research progress is sorely needed.

## 2 Database Scalability in the Cloud

In this section, we first formally establish the notion of scalability. In the context of cloud-computing paradigms, there are two options for scaling the data management layer. The first option is to start with key-value stores, which have almost limitless scalability, and explore ways in which such systems can be enriched to provide higher-level database functionality especially when it comes to providing transactional access to multiple data and informational entities. The other option is to start with a conventional

DBMS architecture and leverage from key-value store architectural design features to make the DBMS highly scalable. We now explore these two options in detail.

### 2.1 Scalability

*Scalability* is a desirable property of a system, which indicates its ability to either handle growing amounts of work in a graceful manner or its ability to improve throughput when additional resources (typically hardware) are added. A system whose performance improves after adding hardware, proportionally to the capacity added, is said to be a scalable system. Similarly, an algorithm is said to scale if it is suitably efficient and practical when applied to large situations (e.g. a large input data set or large number of participating nodes in the case of a distributed system). If the algorithm fails to perform when the resources increase then it does not scale.

There are typically two ways in which a system can scale by adding hardware resources. The first approach is when the system scales *vertically* and is referred to as *scale-up*. To scale vertically (or scale up) means to add resources to a single node in a system, typically involving the addition of processors or memory to a single computer. Such vertical scaling of existing systems also enables them to use virtualization technology more effectively, as it provides more resources for the hosted set of operating system and application modules to share. An example of taking advantage of such shared resources is by by increasing the number of Apache daemon processes running. The other approach of scaling a system is by adding hardware resources *horizontally* referred to as *scale-out*. To scale horizontally (or scale out) means to add more nodes to a system, such as adding a new computer to a distributed software application. An example might be scaling out from one web-server system to a system with three web-servers.

As computer prices drop and performance demand continue to increase, low cost "commodity" systems can be used for building shared computational infrastructures for deploying high-performance applications such as Web search and other web-based services. Hundreds of small computers may be configured in a cluster to obtain aggregate computing power which often exceeds that of single traditional RISC processor based supercomputers. This model has been further fueled by the availability of high performance interconnects. The scale-out model also creates an increased demand for shared data storage with very high I/O performance especially where processing of large amounts of data is required. In general, the scale-out paradigm has served as the fundamental design paradigm for the large-scale data-centers of today. The additional complexity introduced by the scale-out design is the overall complexity of maintaining and administering a large number of compute and storage nodes.

Note that the scalability of a system is closely related to the underlying algorithm or computation. In particular, given an algorithm if there is a fraction $\alpha$ that is inherently sequential then that means that the remainder $1 - \alpha$ is parallelizable and hence can benefit from multiple processors. The maximum *scaling* or *speedup* of such a system using $N$ CPUs is bounded as specified by Amdahl's law [1]:

$$Speedup = \frac{1}{\alpha + \frac{1-\alpha}{N}}.$$

For example if only 70% of the computation is parallelizable then the speedup with 4 CPUs is 2.105 whereas with 8 processors it is only 2.581. The above bound on scaling clearly establishes the need for designing algorithms and mechanisms that are inherently scalable. Blindly adding hardware resources may not necessarily yield the desired scalability in the system.

### 2.2 Data Fusion: Multi-key Atomicity in Key-value Stores

As outlined earlier in the prior section, although key-value stores provide almost infinite scalability in that each entity can (potentially) be handled by in independent node, new application requirements are emerging that require multiple entities (or equivalently keys) to be accessed atomically. Some of these applications are in the domain of cooperative work as well as in the context of multi-player games. This need has been recognized by companies such as Google who have expanded their application portfolio from Web-search to more elaborate applications such as Google documents and others. Given this need, the question arises as to how to support multi-key atomicity in key-value stores such as Google's Bigtable [7], Amazon's Dynamo [17], and Yahoo's PNUTS [9].

The various key-value stores differ in terms of data model, availability, and consistency guarantees, but the property common to all systems is the Key-Value abstraction where data is viewed as key-value pairs and atomic access is supported only at the granularity of *single keys*. This *single key* atomic access semantics naturally allows efficient horizontal data partitioning, and provides the basis for scalability and availability in these systems. Even though a majority of current web applications have *single key* access patterns [17], many current applications, and a large number of Web 2.0 applications (such as those based on collaboration) go beyond the semantics of *single key* access, and foray into the space of *multi key* accesses [2]. Present scalable data management systems therefore cannot directly cater to the requirements of these modern applications, and these applications either have to fall back to traditional databases, or to rely on various ad-hoc solutions.

In order to deal with this challenge, Google has designed a system called MegaStore [3] that builds on Bigtable as an underlying system and creates the notion of entity groups on top of it. The basic idea of MegaStore is to allow users to group multiple entities as a single collection and then uses write-ahead logging [22, 32] and two-phase commit [21] as the building blocks to support ACID transactions on *statically* defined *entity groups*. The designers also postulate that accesses across multiple entity groups are also supported, however, at a weaker or loose consistency level. Although Megastore allows entities to be arbitrarily distributed over multiple nodes, Megastore provides higher level of performance when the entity-group is co-located on a single node. On the other hand if the entity group is distributed across multiple nodes, in that case, the overall performance may suffer since more complex synchronization mechanisms such as two-phase commit or persistent queues may be necessary. We refer to this approach as a **Data Fusion** architecture for multi-key atomicity while ensuring scalability.

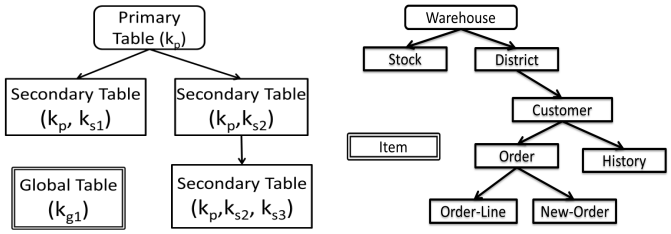
Google's MegaStore takes a step beyond *single key* access patterns by supporting transactional access for *entity groups*. However, since keys cannot be updated in place, once a key is created as a part of a group, it has to be in the group for the rest of its

lifetime. This static nature of *entity groups*, in addition to the requirement that keys be contiguous in sort order, are in many cases insufficient and restrictive. For instance, in case of an online casino application where different users correspond to different key-value pairs, *multi key* access guarantees are needed only during the course of a game. Once a game terminates, different users can move to different game instances thereby requiring guarantees on dynamic groups of keys–a feature not currently supported by MegaStore.

To circumvent this disadvantage, we have designed **G-Store** [14], a scalable data store providing transactional *multi key* access guarantees over *dynamic, non-overlapping groups of keys* using a key-value store as an underlying substrate, and therefore inheriting its scalability, fault-tolerance, and high availability. The basic innovation that allows scalable *multi key* access is the Key Group abstraction which defines a granule of on-demand transactional access. The Key Grouping protocol uses the Key Group abstraction to transfer *ownership*—i.e. the exclusive read/write access to keys—for all keys in a group to a single node which then efficiently executes the operations on the Key Group. This design is suitable for applications that require transactional access to groups of keys that are transient in nature, but live long enough to amortize the cost of group formation. Our assumption is that the number of keys in a group is small enough to be *owned* by a single node. Considering the size and capacity of present commodity hardware, groups with thousands to hundreds of thousands of keys can be efficiently supported. Furthermore, the system can scale-out from tens to hundreds of commodity nodes to support millions of Key Groups. G-Store inherits the data model as well as the set of operations from the underlying Key-Value store; the only addition being that the notions of atomicity and consistency are extended from a single key to a group of keys.

A Key Group consists of a **leader** key and a set of follower keys. The leader is part of the group's identity, but from an applications perspective, the semantics of operations on the leader is no different from that on the followers. Once the application specifies the Key Group, the *group creation* phase of Key Grouping protocol transfers ownership of follower keys to the node currently hosting the leader key, such that transactions executing on the group can be executed locally. Intuitively, the goal of the proposed Key Grouping protocol is to transfer key ownership safely from the followers to the leader during group formation, and from the leader to the followers during group deletion. Conceptually, the follower keys are locked during the lifetime of the group. Safety or correctness requires that there should never be an instance where more than one node claims ownership of an item. Liveness, on the other hand, requires that in the absence of repeated failures, no data item is without an owner indefinitely. The Key Grouping protocol can tolerate message and node failures as well as message re-ordering, concurrent group creation requests as well as detect overlapping group create requests [14].

This data fusion approach provides the building block for designing scalable data systems with consistency guarantees on data granules of different sizes, supporting different application semantics. The two alternative designs have resulted in systems with different characteristics and behavior.




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

Fig. 1: Schema level database partitioning.

### 2.3 Data Fission: Database Partitioning Support in DBMS

Contrary to the approach of data fusion, where multiple small data granules are combined to provide stringent transactional guarantees on larger data granules at scale, another approach to scalability is to split a large database unit into relatively independent *shards* or partitions and provide transactional guarantees only on these shards. We refer to this approach as **Data Fission**. This approach of partitioning the database and scaling out with partitioning is popularly used for scaling web-applications. Since the inefficiencies resulting from distributed transactions are well known (see [11] for some performance numbers), the choice of a good partitioning technique is critical to support flexible functionality while limiting transactions to a single partition. Many modern systems therefore partition the schema in a way such that the need for distributed transactions is minimized–an approach referred to as *schema level partitioning*. Transactions accessing a single partition can be executed efficiently without any dependency and synchronization between the database servers serving the partitions, thus allowing high scalability and availability. Partitioning the database schema, instead of partitioning individual tables, allows supporting rich functionality even when limiting most transactions to a single partition. The rationale behind schema level partitioning is that in a large number of database schemas and applications, transactions only access a small number of related rows which can be potentially spread across a number of tables. This pattern can be used to group related data together in the same partition.

One popular example of partitioning arises when the schema is a "tree 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 [4]. Figure 1(a) provides an illustration of such a schema type. This schema supports three types of tables: **Primary Tables**, **Secondary Tables**, and **Global Tables**. The primary table forms the root of the tree; a schema has exactly 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 1(a), the key $k_p$ of the primary table appears as a foreign key in each of the secondary tables. This structure implies that corresponding to every row in the primary table, there are a group of related rows in the secondary tables, a structure called a **row group** [4]. All rows in the same row group are

guaranteed to be co-located and 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 these 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. In addition to accessing only one row group, an operation in a transaction can only read a global table. Figure 1(b) shows a representation of the TPC-C schema [29] as a tree schema. Such a schema forms the basis of the design of a number of systems such as MS SQL Azure [4], ElasTraS [12], and Relational Cloud [10]. The MS SQL Azure and Relational Cloud designs are based on the *shared nothing* storage model where each DBMS instance on a node is independent and an integrative layer is provided on the top for routing queries and transactions to an appropriate database server. The ElasTraS design on the other hand utilizes the *shared storage* model based on append-only distributed file-systems such as GFS [20] or HDFS [25]. The desirable feature of the ElasTraS design is that it supports elasticity of data in a much more integrated manner. In particular, both MS SQL Azure and Relational Cloud designs need to be augmented with database migration mechanisms to support elasticity where database partition migration involves moving both memory-resident database state and disk-resident data. ElasTraS, on the other hand, can support database elasticity for relocating database partitions by simply migrating the memory state of the database which is considerably simpler. In fact, well-known VM migration techniques [6, 8, 27] can be easily adopted in the case of ElasTraS [15].

This schema level partitioning splits large databases into smaller granules which can then be scaled out on a cluster of nodes. Our prototype system—named ElasTraS [12, 13]—uses this concept of data fission to scale-out database systems. ElasTraS is a culmination of two major design philosophies: traditional relational database systems (RDBMS) that allow efficient execution of OLTP workloads and provide ACID guarantees for small databases and the Key-Value stores that are elastic, scalable, and highly available allowing the system to scale-out. Effective resource sharing and the consolidation of multiple tenants on a single server allows the system to efficiently deal with tenants with small data and resource requirements, while advanced database partitioning and scale-out allows it to serve tenants that grow big, both in terms of data as well as load. ElasTraS operates at the granularity of these data granules called *partitions*. It extends techniques developed for Key-Value stores to scale to large numbers of partitions distributed over tens to hundreds of servers. On the other hand, each partition acts as a self contained database; ElasTraS uses technology developed for relational databases [22] to execute transactions efficiently on these partitions. The partitioning approach described here can be considered as static partitioning. There have been recent efforts to achieve database partitioning at run-time by analyzing the data access patterns of user queries and transactions on-the-fly [11].

## 3 Database Elasticity in the Cloud

One of the major factors for the success of the cloud as an IT infrastructure is its *pay per use* pricing model and *elasticity*. For a DBMS deployed on a pay-per-use cloud

infrastructure, an added goal is to optimize the system's operating cost. *Elasticity*, i.e. the ability to deal with load variations by adding more resources during high load or consolidating the tenants to fewer nodes when the load decreases, all in a live system without service disruption, is therefore critical for these systems.

Even though elasticity is often associated with the scale of the system, a subtle difference exists between elasticity and scalability when used to express a system's behavior. Scalability is a static property of the system that specifies its behavior on a static configuration. For instance, a system design might scale to hundreds or even to thousands of nodes. On the other hand, elasticity is dynamic property that allows the system's scale to be increased *on-demand* while the system is operational. For instance, a system design is elastic if it can scale from 10 servers to 20 servers (or vice-versa) on-demand. A system can have any combination of these two properties.

Elasticity is a desirable and important property of large scale systems. For a system deployed on a pay-per-use cloud service, such as the Infrastructure as a Service (IaaS) abstraction, elasticity is critical to minimize operating cost while ensuring good performance during high loads. It allows consolidation of the system to consume less resources and thus minimize the operating cost during periods of low load while allowing it to dynamically scale up its size as the load decreases. On the other hand, enterprise infrastructures are often statically provisioned. Elasticity is also desirable in such scenarios where it allows for realizing energy efficiency. Even though the infrastructure is statically provisioned, significant savings can be achieved by consolidating the system in a way that some servers can be powered down reducing the power usage and cooling costs. This, however, is an open research topic in its own merit, since powering down random servers does not necessarily reduce energy usage. Careful planning is needed to select servers to power down such that entire racks and alleys in a data-center are powered down so that significant savings in cooling can be achieved. One must also consider the impact of powering down on availability. For instance, consolidating the system to a set of servers all within a single point of failure (for instance a switch or a power supply unit) can result in an entire service outage resulting from a single failure. Furthermore, bringing up powered down servers is more expensive, so the penalty for a miss-predicted power down operation is higher.

In our context of a database system, migrating parts of a system while the system is operational is important to achieve on-demand elasticity—an operation called **live database migration**. While being elastic, the system must also guarantee the tenants' service level agreements (SLA). Therefore, to be effectively used for elasticity, live migration must have low impact—i.e. negligible effect on performance and minimal service interruption—on the tenant being migrated as well as other tenants co-located at the source and destination of migration.

Since migration is a necessary primitive for achieving elasticity, we focus our efforts on developing live migration for the two most common common cloud database architectures: shared disk and shared nothing. Shared disk architectures are utilized for their ability to abstract replication, fault-tolerance, consistency, fault tolerance, and independent scaling of the storage layer from the DBMS logic. Bigtable [7], HBase [24] and ElasTraS [12, 13] are examples of databases that use a shared disk architecture. On the other hand, a shared nothing multi-tenant architecture uses locally attached storage

for storing the persistent database image. Live migration for a shared nothing architecture requires that all database components are migrated between nodes, including physical storage files. For ease of presentation, we use the term **partition** to represent a self-contained granule of the database that will be migrated for elasticity.

In a shared storage DBMS architecture the persistent image of the database is stored in a network attached storage (**NAS**). In the shared storage DBMS architecture, the persistent data of a partition is stored in the NAS and does not need migration. We have designed **Iterative Copy** for live database migration in a shared storage architecture. To minimize service interruption and to ensure low migration overhead, Iterative Copy focuses on transferring the main memory state of the partition so that the partition starts "warm" at the destination node resulting in minimal impact on transactions at the destination, allowing transactions active during migration to continue execution at the destination, and minimizing the tenant's unavailability window. The main-memory state of a partition consists of the cached database state (DB state), and the transaction execution state (Transaction state). For most common database engines [22], the DB state includes the cached database pages or some variant of this. For a two phase locking (**2PL**) based scheduler [22], the transaction state consists of the lock table; for an Optimistic Concurrency Control (**OCC**) [26] scheduler, this state consists of the read and write sets of active transactions and a subset of committed transactions. Iterative Copy guarantees serializability for transactions active during migration and ensures correctness during failures. A detailed analysis of this technique, optimizations, and a detailed evaluation can be found in [15].

In the shared nothing architecture, the persistent image of the database must also be migrated, which is typically much larger than the database cache migrated in the shared disk architecture. As a result, an approach different from Iterative Copy is needed. We have designed **Zephyr**, a technique for live migration in a shared nothing transactional database architecture [19]. Zephyr minimizes service interruption for the tenant being migrated by introducing a synchronized phase that allows both the source and destination to simultaneously execute transactions for the tenant. Using a combination of on-demand pull and asynchronous push of data, Zephyr allows the source node to complete the execution of active transactions, while allowing the destination to execute new transactions. Lightweight synchronization between the source and the destination, only during the short mode of synchronized operation, guarantees serializability, while obviating the need for two phase commit [21]. Zephyr guarantees no service disruption for other tenants, no system downtime, minimizes data transferred between the nodes, guarantees *safe* migration in the presence of failures, and ensures the strongest level of transaction isolation. It uses standard tree based indices and lock based concurrency control, thus allowing it to be used in a variety of DBMS implementations. Zephyr does not rely on replication in the database layer, thus providing greater flexibility in selecting the destination for migration, which might or might not have the tenant's replica. However, considerable performance improvement is possible in the presence of replication when a tenant is migrated to one of the replicas.

## 4 Database Autonomy in the Cloud

Managing large systems poses significant challenges in monitoring, management, and system operation. Moreover, to reduce the operating cost, considerable autonomy is needed in the administration of such systems. In the context of database systems, the responsibilities of this autonomic controller include monitoring the behavior and performance of the system, elastic scaling and load balancing based on dynamic usage patterns, modeling behavior to forecast workload spikes and take pro-active measures to handle such spikes. An autonomous and intelligent system controller is essential to properly manage such large systems.

Modeling the behavior of a database system and performance tuning has been an active area of research over the last couple of decades. A large body of work focuses on tuning the appropriate parameters for optimizing database performance [18, 31], primarily in the context of a single database server. Another line of work has focused on resource prediction, provisioning, and placement in large distributed systems [5,30].

To enable autonomy in a cloud database, an intelligent system controller must also consider various additional aspects, specifically in the case when the database system is deployed on a pay-per-use cloud infrastructure while serving multiple application tenant instances, i.e., a multitenant cloud database system. In such a multitenant system, each tenant pays for the service provided and different tenants in the system can have competing goals. On the other hand, the service provider must share resources amongst the tenants, wherever possible, to minimize the operating cost to maximize profits. A controller for such a system must be able to model the dynamic characteristics and resource requirements of the different application tenants to allow elastic scaling while ensuring good tenant performance and ensuring that the tenants' service level agreements (**SLAs**) are met. An autonomic controller consists of two logical components: the *static* component and the *dynamic* component.

The static component is responsible for modeling the behavior of the tenants and their resource usage to determine tenant placement to co-locate tenants with complementary resource requirements. The goal of this tenant placement algorithm is to minimize the total resource utilization and hence minimize operating cost while ensuring that the tenant SLAs are met. Our current work uses a combination of machine learning techniques to classify tenant behavior followed by tenant placement algorithms to determine optimal tenant co-location and consolidation. This model assumes that once the behavior of a tenant is modeled and a tenant placement determined, the system will continue to behave the way in which the workload was modeled, and hence is called the static component. The dynamic component complements this static model by detecting dynamic change in load and resource usage behavior, modeling the overall system's behavior to determine the opportune moment for elastic load balancing, selecting the minimal changes in tenant placement needed to counter the dynamic behavior, and use the live database migration techniques to re-balance the tenants. In addition to modeling tenant behavior, it is also important to predict the migration cost such that a migration to minimize the operating cost does not violate a tenant's SLA. Again, we use machine learning models to predict the migration cost of tenants and the re-placement model accounts for this cost when determining *which* tenant to migrate, *when* to migrate, and *where* to migrate [15].

## 5 Concluding Remarks

Database systems deployed on a cloud computing infrastructure face many new challenges such as dealing with large scale operations, lightweight elasticity, and autonomic control to minimize the operating cost. These challenges are in addition to making the systems fault-tolerant and highly available. In this article, we presented an overview of some of our current research activities to address the above-mentioned challenges in designing a scalable data management layer in the cloud.

## References

1. Amdahl, G.: Validity of the single processor approach to achieving large-scale computing capabilities. In: AFIPS Conference. p. 483485 (1967)
2. Amer-Yahia, S., Markl, V., Halevy, A., Doan, A., Alonso, G., Kossmann, D., Weikum, G.: Databases and Web 2.0 panel at VLDB 2007. SIGMOD Rec. 37(1), 49–52 (2008)
3. Baker, J., Bond, C., Corbett, J., Furman, J., Khorlin, A., Larson, J., Leon, J.M., Li, Y., Lloyd, A., Yushprakh, V.: Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In: CIDR. pp. 223–234 (2011)
4. Bernstein, P.A., Cseri, I., Dani, N., Ellis, N., Kalhan, A., Kakivaya, G., Lomet, D.B., Manner, R., Novik, L., Talius, T.: Adapting Microsoft SQL Server for Cloud Computing. In: ICDE (2011)
5. Bodík, P., Goldszmidt, M., Fox, A.: Hilighter: Automatically building robust signatures of performance behavior for small- and large-scale systems. In: SysML (2008)
6. Bradford, R., Kotsovinos, E., Feldmann, A., Schiöberg, H.: Live wide-area migration of virtual machines including local persistent state. In: VEE. pp. 169–179 (2007)
7. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: A Distributed Storage System for Structured Data. In: OSDI. pp. 205–218 (2006)
8. Clark, C., Fraser, K., Hand, S., Hansen, J.G., Jul, E., Limpach, C., Pratt, I., Warfield, A.: Live migration of virtual machines. In: NSDI. pp. 273–286 (2005)
9. Cooper, B.F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.A., Puz, N., Weaver, D., Yerneni, R.: PNUTS: Yahoo!'s hosted data serving platform. Proc. VLDB Endow. 1(2), 1277–1288 (2008)
10. Curino, C., Jones, E., Popa, R., Malviya, N., Wu, E., Madden, S., Balakrishnan, H., Zeldovich, N.: Relational Cloud: A Database Service for the Cloud. In: CIDR. pp. 235–240 (2011)
11. Curino, C., Zhang, Y., Jones, E.P.C., Madden, S.: Schism: a workload-driven approach to database replication and partitioning. PVLDB 3(1), 48–57 (2010)
12. Das, S., Agarwal, S., Agrawal, D., El Abbadi, A.: ElasTraS: An Elastic, Scalable, and Self Managing Transactional Database for the Cloud. Tech. Rep. 2010-04, CS, UCSB (2010)
13. Das, S., Agrawal, D., El Abbadi, A.: ElasTraS: An Elastic Transactional Data Store in the Cloud. In: USENIX HotCloud (2009)
14. Das, S., Agrawal, D., El Abbadi, A.: G-Store: A Scalable Data Store for Transactional Multi key Access in the Cloud. In: ACM SoCC. pp. 163–174 (2010)
15. Das, S., Nishimura, S., Agrawal, D., El Abbadi, A.: Live Database Migration for Elasticity in a Multitenant Database for Cloud Platforms. Tech. Rep. 2010-09, CS, UCSB (2010)
16. Dean, J.: Talk at the Google Faculty Summit (2010)

17. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: SOSP. pp. 205–220 (2007)
18. Duan, S., Thummala, V., Babu, S.: Tuning database configuration parameters with ituned. Proc. VLDB Endow. 2, 1246–1257 (August 2009)
19. Elmore, A., Das, S., Agrawal, D., El Abbadi, A.: Zephyr: Live Database Migration for Lightweight Elasticity in Multitenant Cloud Platforms. under submission for review
20. Ghemawat, S., Gobioff, H., Leung, S.T.: The Google file system. In: SOSP. pp. 29–43 (2003)
21. Gray, J.: Notes on data base operating systems. In: Operating Systems, An Advanced Course. pp. 393–481. Springer-Verlag, London, UK (1978)
22. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers Inc. (1992)
23. Hamilton, J.: I love eventual consistency but... http://bit.ly/hamilton-eventual (April 2010)
24. HBase: Bigtable-like structured storage for Hadoop HDFS (2010), http://hadoop.apache.org/hbase/
25. HDFS: A distributed file system that provides high throughput access to application data (2010), http://hadoop.apache.org/hdfs/
26. Kung, H.T., Robinson, J.T.: On optimistic methods for concurrency control. ACM Trans. Database Syst. 6(2), 213–226 (1981)
27. Liu, H., et al.: Live migration of virtual machine based on full system trace and replay. In: HPDC. pp. 101–110 (2009)
28. Obasanjo, D.: When databases lie: Consistency vs. availability in distributed systems. http://bit.ly/4J0Zm (2009)
29. The Transaction Processing Performance Council: TPC-C benchmark (Version 5.10.1) (2009)
30. Urgaonkar, B., Rosenberg, A.L., Shenoy, P.J.: Application placement on a cluster of servers. Int. J. Found. Comput. Sci. 18(5), 1023–1041 (2007)
31. Weikum, G., Moenkeberg, A., Hasse, C., Zabback, P.: Self-tuning database technology and information services: from wishful thinking to viable engineering. In: VLDB. pp. 20–31 (2002)
32. Weikum, G., Vossen, G.: Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery. Morgan Kaufmann Publishers Inc. (2001)