Cluster Comput (2014) 17:487–502
DOI 10.1007/s10586-013-0290-7

Cloud-hosted databases: technologies, challenges and opportunities

Sherif Sakr

Received: 10 December 2012 / Accepted: 18 June 2013 / Published online: 13 July 2013

Abstract One of the main advantages of the cloud computing paradigm is that it simplifies the time-consuming processes of hardware provisioning, hardware purchasing and software deployment. Currently, we are witnessing a proliferation in the number of cloud-hosted applications with a tremendous increase in the scale of the data generated as well as being consumed by such applications. Cloud-hosted database systems powering these applications form a critical component in the software stack of these applications. To better understand the challenges in developing effective cloud-hosted database systems, this article discusses the existing technologies for hosting the database tier of software applications in cloud environments, illustrates their strengths and weaknesses, and presents some opportunities for future work.

Keywords Cloud · Databases · Consistency · Transactions · Replication

1 Introduction

Cloud computing technology represents a new paradigm for hosting software applications. This paradigm simplifies the time-consuming processes of hardware provisioning, hardware purchasing and software deployment. Thus, it revolutionized the way computational resources and services are commercialized and delivered to customers. In particular, it shifts the location of this infrastructure to the network to reduce the costs associated with the management of hardware and software resources. Therefore, it represents the long-held dream of envisioning computing as a utility [4] where the economy of scale principles help to effectively drive down the cost of computing infrastructure. Hence, cloud computing promises a number of advantages for the deployment of software applications such as pay-per-use cost model, low time to market, and the perception of (virtually) unlimited resources and infinite scalability. In practice, the advantages of the cloud computing paradigm opens up new avenues for deploying novel applications which were not economically feasible in a traditional enterprise infrastructure setting. Therefore, the cloud has become an increasingly popular platform for hosting software applications in a variety of domains such as e-retail, finance, news and social networking. Thus, we are witnessing a proliferation in the number of applications with a tremendous increase in the scale of the data generated as well as being consumed by such applications. *Cloud-hosted database systems powering these applications form a critical component in the software stack of these applications*.

In general, data-intensive applications are classified into two main classes: (1) On Line Transaction Processing (OLTP) systems that deal with operational databases of sizes of up to a few Terabytes with write-intensive workloads that require ACID transactional support and response time guarantees. (2) On Line Analytical Processing (OLAP) systems that deals with historical databases of very large sizes of up to Petabytes with read-intensive workloads that are more tolerant to relaxed ACID properties. *In this article, we focus on cloud-hosted database solutions for OLTP systems*.

In principle, a successful cloud-hosted database tier of an OLTP system should sustain a number of goals [21]:

S. Sakr (✉)
National ICT Australia (NICTA) and School of Computer Science and Engineering, University of New South Wales, Sydney, Australia
e-mail: ssakr@cse.unsw.edu.au

- *Availability*: They must be always accessible even on the occasions of a network failure or when a whole datacenter has gone offline.
- *Scalability*: They must be able to support very large databases with very high request rates at very low latency. In particular, the system must be able to automatically replicate and redistribute data to take advantage of the new hardware. They must be also able to automatically move load between servers (replicas).
- *Elasticity*: They must cope with changing application needs in both directions (scaling up/out or scaling down/in). Moreover, the system must be able to gracefully respond to these changing requirements and quickly recover to its steady state.
- *Performance*: On public cloud computing platforms, pricing is structured in a way such that one pays only for what one uses, so the vendor price increases linearly with the requisite storage, network bandwidth and compute power. Hence, the system performance has a direct effect on its costs. Thus, efficient system performance is a crucial requirement to save money.

Arguably, one of the main goals of cloud-hosted database system is to facilitate the job of implementing every application as a *distributed*, *scalable* and *widely-accessible* service on the Web. The *Amazon* online retailer, *eBay*, *Facebook*, *Twitter*, *Flickr*, *YouTube*, and *Linkedin* are just examples of online services which are currently able to successfully achieve this goal. Such services have two main characteristics of which they are: *data-intensive* and very *interactive*. For example, the Facebook social network contains 950 million users.[1] Each user has an average of 130 friendship relations. Moreover, there are about 900 million objects with which registered users interact such as: pages, groups, events and community pages. Other smaller scale social networks such as Linkedin which is mainly used for professionals has more than 175 million registered users. Therefore, it becomes an ultimate goal to make it easy for every application to achieve such high scalability, availability and performance goals with minimum effort.

The quest for conquering the challenges posed by hosting databases on cloud computing environments has led to a plethora of systems and approaches. In practice, there are three main technologies which are commonly used for deploying the database tier of software applications in cloud platforms, namely, the services of *NoSQL* storage systems, *Database-as-a-service* (*DaaS*) platforms and *virtualized database servers*. This article aims to discuss the basic characteristics and the recent advancements of each of these technologies, illustrate the strengths and weaknesses of each technology and presents some opportunities for future work which are required to tackle existing research challenges and bring forward the vision of deploying data-intensive applications on cloud platforms.

[1] http://www.facebook.com/press/info.php?statistics.

2 NoSQL database systems

For decades, relational database management systems (e.g. MySQL, PostgreSQL, SQL Server, Oracle) have been considered as the *one-size-fits-all* solution for providing data persistence and its retrieval for decades. In principle, these systems have matured after extensive research and development efforts and very successfully created a large market of solutions in different business domains. However, the ever increasing need for scalability and new application requirements have created new challenges for traditional RDBMS. Therefore, recently, there has been some dissatisfaction with this one-size-fits-all approach in deploying the data storage tier for large scale online web services [48] which resulted in the emergence of a new generation of low-cost, high-performance database software that challenges the dominance of relational database management systems. A big reason for this movement, named as *NoSQL* (Not Only SQL), is that different implementations of Web, enterprise, and cloud computing applications that have different set of desideratum in the requirements from their data management tiers (e.g. not every application requires rigid data consistency) have opened up various possibilities in the design space. For example, for high-volume Web sites (e.g. eBay, Amazon, Twitter, Facebook), scalability and high availability are essential requirements that can not be compromised. For these applications, even the slightest outage can have significant financial consequences and impacts customer trust. The *CAP* theorem [12] have shown that a distributed database system can only choose at most two out of three properties: *Consistency*, *Availability* and *tolerance to Partitions*. Therefore, most of these systems decide to compromise the strict consistency requirement. In particular, they apply a relaxed consistency policy called *eventual consistency* [52] which guarantees that if no new updates are made to a replicated object, eventually all accesses will return the last updated value [52]. If no failures occur, the maximum size of the *inconsistency window* can be determined based on factors such as communication delays, the load on the system, and the number of replicas involved in the replication scheme.

BigTable [18] (presented by Google) and *Dynamo* [27] (presented by Amazon) have provided a *proof of concept* that inspired and triggered the development of a new wave of the NoSQL systems. In particular, BigTable has demonstrated that persistent record storage could be scaled to thousands of nodes while Dynamo has pioneered the idea of eventual consistency as a way to achieve higher availability

Table 1 Design decisions of sample NoSQL systems

System	Data model	Consistency guarantee	CAP options	License
BigTable	Column Families	Eventually Consistent	CP	Internal at Google
PNUTS	Key-Value Store	Timeline Consistent	AP	Internal at Yahoo!
Dynamo	Key-Value Store	Eventually Consistent	AP	Internal at Amazon
S3	Document Store	Eventually Consistent	AP	Commercialized by Amazon
SimpleDB	Key-Value Store	Eventually Consistent	AP	Commercialized by Amazon
HBase	Column Families	Strictly Consistent	CP	Open source–Apache
Cassandra	Column Families	Eventually Consistent	AP	Open source–Apache
MongoDB	Document Store	Eventually Consistent	AP	Open source–GPL

and scalability. In principle, the implementations of NoSQL systems have a number of common design features such as:

- Supporting flexible data models with the ability to dynamically define new attributes or data schema.
- A simple call level interface or protocol (in contrast to a SQL binding) which does not support join operations.
- Supporting weaker consistency models than the ACID transactions in most traditional RDBMS. These models are usually referred to as *BASE* models (*B*asically *A*vailable, *S*oft state, *E*ventually consistent) [41].
- The ability to horizontally scale out throughput over many servers.
- Efficient use of distributed indexes and RAM for data storage.

Commercial cloud offerings of this approach include *Amazon S3*,[2] *Amazon SimpleDB*[3] and *Microsoft Azure Table Storage*.[4] In addition, there is a large number of open source projects that have been introduced which follow the same principles of NoSQL systems [14] such as *HBase*,[5] *Cassandra*,[6] *Voldemort*,[7] *Dynomite*,[8] *Riak*[9] and *MongoDB*.[10] In general, these NoSQL systems can be classified with respect to different characteristics. For example, based on their supported data model, they can be classified into the following categories:

- *Key-value stores*: These systems use the simplest data model which is a collection of objects where each object has a unique key and a set of attribute/value pairs.
- *Extensible record stores*: They provide variable-width tables (Column Families) that can be partitioned vertically and horizontally across multiple servers.
- *Document stores*: The data model of these systems consists of objects with a variable number of attributes with a possibility of having nested objects.

In addition, the systems can be classified based on their support of the properties of the CAP theorem into three categories:

- *CA systems*: Consistent and highly available, but not partition-tolerant.
- *CP systems*: Consistent and partition-tolerant, but not highly available.
- *AP systems*: Highly available and partition-tolerant, but not consistent.

In practice, choosing the adequate NoSQL system (from the very wide available spectrum of choices) with design decisions that best fit with the requirements of a software application is not a trivial task and requires a careful consideration. Table 1 provides an overview of different design decision for sample NoSQL systems. For comprehensive survey of the NoSQL system and their design decisions, we refer the reader to [14, 44].

In general, the capabilities of the NoSQL systems have attracted a lot of attractions. However, there are many obstacles still need to overcome before theses systems can appeal to mainstream enterprises such as:

- *Programming Model*: NoSQL databases offer few facilities for ad-hoc query and analysis. Even a simple query requires significant programming expertise. The inability of such systems to declaratively express the important join operation has been always considered one of the main limitations of these systems.
- *Transaction Support*: Transaction management is one of the powerful features of RDBMS. The current limited support (if any) of the transaction notion from NoSQL database systems is considered as a big obstacle towards their acceptance in implementing mission critical systems. In principle, developing applications on top of an

[2] http://aws.amazon.com/s3/.

[3] http://aws.amazon.com/simpledb/.

[4] http://msdn.microsoft.com/en-us/library/windowsazure/dd179423.aspx.

[5] http://hbase.apache.org/.

[6] http://cassandra.apache.org/.

[7] http://project-voldemort.com/.

[8] http://wiki.github.com/cliffmoon/dynomite/dynomite-framework.

[9] http://wiki.basho.com/display/RIAK/Riak.

[10] http://www.mongodb.org/.

eventually consistent NoSQL datastore requires a higher effort compared to traditional databases because they hinder the ability to support key features such as data independence, reliable transactions, and other crucial characteristics often required by applications that are fundamental to the database industry [53].

- *Migration*: Migrating existing software application that uses relational database to NoSQL offerings would require substantial changes in the software code due to the differences in the data model, query interface and transaction management support. In practice, it might require a complete re-write of the source code which requires any interaction with the data management tier of the software application.
- *Maturity*: RDBMS systems are well-know of their high stability and rich functionalities. In comparison, most NoSQL alternatives are still pre-production versions with many key features being either not stable enough or yet to be implemented. Therefore, enterprises are still approaching this new wave of data management with extreme caution.

Therefore, there is still a big debate between the proponents of the NoSQL and RDBMS camps which is centered around the right choice for implementing online transaction processing systems. RDBMS proponents think that the NoSQL camp has not spent sufficient time to understand the theoretical foundation of the transaction processing model. For example, the eventual consistency model is still not well-defined and different implementations may differ significantly with each other. This means figuring out all these inconsistent behavior lands on the application developer's responsibilities and make their life very much harder. In addition, they believe that NoSQL systems could be more suitable for OLAP applications rather than for OLTP applications [1]. On the other hand, the NoSQL camp argues that the domain-specific optimization opportunities of NoSQL systems give back more flexibility to the application developers who now no longer constrained by a one-size-fits-all model. However, they admit that making such optimization decision requires a lot of experience and can be very error-prone and dangerous if not done by experts.

3 Database-as-a-Service (DaaS)

Data centers are often under-utilized due to over-provisioning as well as time-varying resource demands of typical enterprise applications. *Multi-tenancy*, a technique which is pioneered by *salesforce.com*,[11] is an optimization mechanism for hosted services in which multiple customers are consolidated onto the same operational system and thus the economy of scale principles help to effectively drive down the cost of computing infrastructure. In particular, multi-tenancy allows pooling of resources which improves utilization by eliminating the need to provision each tenant for their maximum load. Therefore, multi-tenancy is an attractive mechanism for both of the cloud providers who are able to serve more customers with a smaller set of machines, and also to customers of cloud services who do not need to pay the price of renting the full capacity of a server.

In practice, there are three main approaches for the implementation of multi-tenant database systems [32]:

1. *Shared Server*: where each tenant is offered a separate database in the same database server.
2. *Shared Process*: where each tenant is offered its own tables while multiple tenants can share the same database.
3. *Shared Table*: where the data of all tenants is stored in the same tables and each tuple has an additional column with the tenant identifier.

Database-as-a-Service is a technology where a third party service provider hosts a relational database as a service [2]. Such services alleviate the need for their users to purchase expensive hardware and software, deal with software upgrades and hire professionals for administrative and maintenance tasks. Cloud offerings of this approach include *Amazon RDS*,[12] *Microsoft SQL Azure*,[13] *Google Cloud SQL*[14] and *Heroku Postgres*.[15] Research efforts include the *Relational Cloud* project.[16] While the shared table multi-tenancy model can be used by SaaS providers (e.g. Salesforce.com) because all tenants share the same database structure for their application, the shared server multi-tenancy model is the most commonly used by most commercial DaaS providers as it is considered to be the most effective approach to secure the isolation of each tenant's data and allocated computing resources.

Amazon RDS is an example of a relational database service which gives its users the access to the full capabilities of a familiar MySQL database or Oracle. Hence, the code, applications, and tools which are already designed on existing MySQL or Oracle databases can work seamlessly with Amazon RDS. Once the database instance is running, Amazon RDS can automate common administrative tasks such as performing backups or patching the database software. Amazon RDS can also manages automatic failover management. *Google Cloud SQL* is another service that provide the capabilities and functionality of MySQL database

[11] http://www.salesforce.com/.

[12] http://aws.amazon.com/rds/.

[13] http://www.microsoft.com/windowsazure/sqlazure/.

[14] https://developers.google.com/cloud-sql/.

[15] https://postgres.heroku.com/.

[16] http://relationalcloud.com/.

servers which are hosted in Google's cloud. Although there is tight integration of the services with *Google App Engine* (Google's Platform-as-a-Service software development environment), in contrast to the original built-in data store of Google App Engine, Google Cloud SQL allows the software applications to easily move their data in and out of Google's cloud without any obstacles. Microsoft has released the Microsoft *SQL Azure* Database system as a cloud-based relational database service which has been built on Microsoft SQL Server technologies. It provides a highly available, multi-tenant database service hosted by Microsoft in the cloud. Therefore, applications can create, access and manipulate tables, views, indexes, roles, stored procedures, triggers, and functions. It can execute complex queries and joins across multiple tables. It also supports Transact-SQL (T-SQL), native ODBC and ADO.NET data access. In particular, SQL Azure service can be seen as running an instance of SQL server in a cloud hosted server which is automatically managed by Microsoft instead of running on-premise managed server. Similarly, *Heroku Postgres* provides a web service which provides the functionalities of the SQL-compliant database, *PostgreSQL*.

Relational Cloud [24] represents a research effort for developing a system that hosts multiple databases on a pool of commodity servers inside one data center. In order to allow workloads to scale across multiple servers, the system relies on a graph-based data partitioning algorithm that groups data items according to their frequency of co-access within transactions/queries. The main goal of this partitioning process is to minimize the probability that a given transaction has to access multiple nodes to complete its execution. In addition, in order to effectively manage and allocate the available computing resources to the different tenants, the system monitors the access patterns induced by the tenants' workloads and the load of each database server, and uses this information to periodically determine the best way to place the database partitions on the back-end machines. The goal of this monitoring process is to minimize the number of used machines and balance the load on the different servers.

In practice, the migration of the database tier of any software application to a relational database service is expected to require minimal effort if the underlying RDBMS of the existing software application is compatible with the offered service. This helps the software applications to achieve faster time-to-market because they can quickly host the database tier of their application in cloud platforms, and utilize their features and advantages. However, many relational database systems are, as yet, not supported by the DaaS paradigm (e.g. *IBM DB2*, *Informix*, *Sybase*). In addition, some limitations or restrictions might be introduced by the service provider for different reasons[17] (e.g. the maximum size of the hosted database, the maximum number of possible concurrent connections). Moreover, software applications do not have sufficient flexibility in being able to control the allocated resources of their applications (e.g. dynamically allocating more resources for dealing with increasing workload or dynamically reducing the allocated resources in order to reduce the operational cost). The whole resource management and allocation process is controlled at the provider side which require an accurate planning for the allocated computing resources for the database tier and limits the ability of the consumer applications to maximize their benefits by leveraging the elasticity and scalability features of the cloud environment.

[17] http://msdn.microsoft.com/en-us/library/windowsazure/ee336245.aspx.

4 Virtualized database servers

Virtualization is a key technology of the cloud computing paradigm that abstracts away the details of physical hardware and provides virtualized resources for high-level applications. A virtualized server is commonly called a *virtual machine* (*VM*). VMs allow both the isolation of applications from the underlying hardware and other VMs. Ideally, each VM is both unaware and unaffected by other VMs which could be operating on the same physical machine. In principle, resource virtualization technologies add a flexible and programmable layer of software between applications and the resources used by these applications. The approach of *virtualized database server* makes use of these advantages where an existing database tier of a software application that has been designed to be used in a conventional data center can be directly ported to virtual machines in the public cloud. Such migration process usually requires minimal changes in the architecture or the code of the deployed application. In this approach, database servers, like any other software components, are migrated to run in virtual machines. While the provisioning of a virtual machine for each database replica imposes a performance overhead, this overhead is estimated to be of less than 10 % [39]. In principle, this approach represents a different model of multi-tenancy, *shared physical machine*, where a VM of a virtualized database server can be running on the same physical machine with other VMs which are not necessarily to be running database operations.

Dynamic provisioning is a well-known process of increasing or decreasing the allocated computing resources (e.g. number of virtualized database servers) to an application in response to workload changes. In practice, one of the major advantages of the *virtualized database server* approach is that the application can have full control in dynamically allocating and configuring the physical resources of the database tier (database servers) as needed [16, 42, 46]. Hence, software applications can fully utilize the elasticity

feature of the cloud environment to achieve their defined and customized scalability or cost reduction goals. However, achieving these goals requires the existence of an *admission control* component which is responsible for monitoring the system state and taking the corresponding actions (e.g. allocating more/less computing resources) according to the defined application requirements and strategies. Therefore, one of the main responsibilities of this admission control component is on deciding *when* to trigger an increase or decrease in the number of the virtualized database servers which are allocated to the software application.

In general, the decision of *when* to an increase or decrease in the allocated computed resources is made in a *lazy* fashion for the web and the application tiers of the software application, in response to an actual or anticipated significant workload change. Such lazy triggers are appropriate for these tiers since that new capacity can be added relatively in a quick manner and whenever required as the only incurred latency is for virtual machine startup. However, provisioning of a new database replica involves copying and restoring a new replica which can take minutes or hours depending on the database size. Therefore, the *Dolly* system [16] has presented an approach that takes the latency of provisioning a new database replicas into account when triggering *eager* provisioning decisions. In particular, Dolly incorporates a model to estimate the latency to create a replica, based on the virtual machine snapshot size and the database re-synchronization latency, and uses this model to trigger the replica spawning process well in advance of the anticipated workload increase. The *CloudDB AutoAdmin* framework [42] has presented another approach for SLA-based dynamic provisioning of the database tier of the software applications based on application-defined policies for satisfying their own SLA requirements. In this framework, the SLA of the consumer applications are declaratively defined in terms of rules that define goals which are subjected to a number of constraints that are specific to the application requirements. The framework continuously monitors the application-defined SLA and automatically triggers the execution of necessary provisioning actions when the conditions of the rules are met. Hence, the software applications has more flexibility in defining their own lazy or eager provisioning rules. Soror et al. [46] presented a *virtualization design advisor* which uses information about the anticipated workloads to automatically determine an appropriate configuration for the virtual machine in which it runs so that it can avoid allocating resources to DBMS instances of which little benefit will be obtained. The advisor relies on cost models that can predict the workload performance under different resource allocations. For example, the advisor can distinguish CPU intensive workloads from I/O intensive workloads and allocate more CPU to the former case.

5 Challenges and opportunities

Cloud-hosted database systems represent critical components of the cloud computing services and infrastructure. They play an important role in ensuring the smooth deployment or migration of software applications from the traditional enterprise infrastructures and on-premise data centers to the new cloud platforms and infrastructures. In this section, we shed the lights on a set of novel research challenges, that have been introduced by the cloud computing paradigm that need to be addressed in order to ensure that the vision of designing and implementing successful management solutions in the cloud environment can be achieved.

5.1 True elasticity

Cloud computing is by its nature a fast changing environment which is designed to provide services to unpredictably diverse sets of clients and heterogenous workloads. For example, a common characteristic of internet scale applications and services is that they can be used by large numbers of end-users and highly variable load spikes in the demand for services can occur depending on the day and the time of year, and the popularity of the application. In addition, the workload characteristic could vary significantly from one application type to another where possible fluctuations on the workload characteristics which could be of several orders of magnitude on the same business day may also occur [11].

In principle, elasticity is one of the most important features which is provided by cloud computing platforms. In general, cloud platforms support on-demand allocation of servers and employ a *pay-as-you-go* service model. These features are attractive from the perspective of the customers of cloud services, since servers can be requested only when a workload spike arrives or is anticipated, and charging is based only on the duration of the workload surge [49]. Therefore, to unleash the power of the cloud computing paradigm, cloud database systems should be able to transparently manage and utilize the elastic computing resources to deals with fluctuating workloads. In particular, they should allow users to add and remove computing resources as necessary. For example, to deal with increasing workloads, software applications can simply add more resources (e.g. database replicas or database servers) and when the workload is decreasing, software applications can release some resources back to the cloud provider in order to lower the monetary cost [43].

In practice, both of the commercial NoSQL cloud offerings (e.g. Amazon SimpleDB) and commercial DaaS offerings (e.g. Amazon RDS, Microsoft SQL Azure) do not provide their users any flexibility to dynamically increase or decrease the allocated computing resources of their applications. While NoSQL offerings claim to provide elastic

services of their tenants, they do not provide any guarantee that their provider-side elasticity management will provide scalable performance with increasing workloads [7]. Moreover, commercial DaaS pricing models require their users to pre-determine the computing capacity that will be allocated to their database instance as they provide standard packages of computing resources (e.g. *Micro*, *Small*, *Large* and *Extra Large* DB Instances). In practice, predicting the workload behavior (e.g. arrival pattern, I/O behavior, service time distribution) and consequently accurate planning of the computing resource requirements with consideration of their monetary costs are very challenging tasks. Therefore, the user might still tend to over-provision the allocated computing resources for the database tier of their application in order to ensure satisfactory performance for their workloads. As a result of this, the software application is unable to fully utilize the elastic feature of the cloud environment. The approach of virtualized database server provides software applications with more flexibility and control on being able to dynamically allocate and configure the physical resources of the database tier (database servers). However, this requires implementing an admission control component which is responsible for executing the application logic of its elasticity mechanism (see Sect. 4).

Xiong et al. [54] have presented an provider-centric approach for intelligently managing the computing resources in a shared multi-tenant database system at the virtual machine level. The proposed approach consists of two main components: (1) The system modeling module that uses machine learning techniques to learn a model that describes the potential profit margins for each tenant under different resource allocations. The learned model considers many factors of the environment such as SLA cost, client workload, infrastructure cost and action cost. (2) The resource allocation decision module dynamically adjusts the resource allocations, based on the information of the learned model, of the different tenants in order to achieve the optimum profits. Tatemura et al. [51] proposed a declarative approach for achieving elastic OLTP workloads. The approach is based on defining of two main components: (1) The transaction classes required for the application. (2) The actual workload with references to the transaction classes. Using this information, a formal model can be defined to analyze elasticity of the workload with transaction classes specified. The approach of [42] is more consumer-centric as it enables the software applications to declaratively define their scaling in and out rules according to specific application requirements and policies.

In general, we believe that there is a lack of flexible and powerful consumer-centric elasticity mechanisms that enable software application to have more control on allocating the computing resources for the database tier of their applications over the application running time and make the best use of the elasticity feature of the cloud computing environments. More attention should be given to these issues in the future work from the research community.

5.2 Data replication and consistency management

Data replication is a well-known strategy to achieve the availability, scalability and performance improvement goals in the data management world. In general, stateless services are easy to scale in the cloud since any new *replicas* of these services can operate completely independently of other instances. In contrast, scaling stateful services, such as a *database system*, needs to guarantee a consistent view of the system for users of the service. However, the cost of maintaining several database replicas that are always strongly consistent is very high. As we have previously described, according to the *CAP* theorem, most of the cloud data management solutions overcome the difficulties of distributed replication by relaxing the consistency guarantees of the system and supporting various forms of weaker consistency models (e.g. eventual consistency [52]). In practice, a common feature of the *NoSQL* and *DaaS* cloud offerings is the creation and management of multiple replicas (usually 3) of the stored data while a replication architecture is running behind-the-scenes to enable automatic failover management and ensure high availability of the service. In general, replicating for performance differs significantly from replicating for availability or fault tolerance. The distinction between the two situations is mainly reflected by the higher degree of replication, and as a consequence the need for supporting weak consistency when scalability is the motivating factor for replication [15].

Zhao et al. [55] have conducted an experimental evaluation of the performance characteristics of database replication of virtualized database servers on cloud environments where different database replicas can be hosted on different data centers with different geographic locations. The results of the study show that the performance variation of the dynamically allocated virtual machines is an inevitable issue that needs to be considered when deploying database in the cloud. Different configurations of geographic locations can noticeably affect the end-to-end throughput as well. As the application workload increases, the replication delay increases. However, as the number of database replicas increases, the replication delay decreases. The replication delay showed to be more affected by the workload increase than the configurations of the geographic location of the database replicas.

Kraska et al. [34] have described a dynamic consistency strategy, called *Consistency Rationing*, to reduce the consistency requirements when possible and raise them when it matters. The adaptation is driven by a cost model and different strategies that dictate how the system should behave.

In particular, they divide the data items into three categories (A, B, C) and treat each category differently depending on the consistency level provided. The A category represents data items for which we need to ensure strong consistency guarantees as any consistency violation would result in large penalty costs, the C category represents data items that can be treated using session consistency as temporary inconsistency is acceptable while the B category comprises all the data items where the consistency requirements vary over time depending on the actual availability of an item. Therefore, the data of this category is handled with either strong or session consistency depending on a statistical-based policy for decision making. Keeton et al. [20] have proposed a similar approach in a system called *LazyBase* that allows users to trade off query performance and result freshness. LazyBase breaks up metadata processing into a pipeline of ingestion, transformation, and query stages which can be parallelized to improve performance and efficiency. By breaking up the processing, LazyBase can independently determine how to schedule each stage for a given set of metadata, thus providing more flexibility than existing monolithic solutions. LazyBase uses models of transformation and query performance to determine how to schedule transformation operations to meet users' freshness and performance goals and to utilize resources efficiently. Zhao et al. [56, 57] introduced an adaptive framework for asynchronous database replication that enables keeping several replicas of the database, on virtualized database servers, in different data centers (with potentially different geographic locations) and provides the software applications with flexible mechanisms for specifying different levels of service level agreements (SLA) of data freshness for the database replicas. In particular, the framework allows specifying an SLA of data freshness for each database replica and continuously monitor the replication delay of each replica so that once a replica violates its defined SLA, the framework automatically injects new replica at the closest geographic location in order to balance the workload and re-satisfy the defined SLA.

In general, data replication across different data centers is expensive. Synchronous wide-area replication mechanisms are considered to be unfeasible to achieve strong consistency requirements. Therefore, many solutions either rely on asynchronous replication mechanism and weaker consistency guarantees. *PNUTS* (Yahoo's NoSQL data store) [22] was one of the earliest systems to natively support geographic replication using asynchronous replication mechanism and publish/subscribe message exchange protocol. It uses a per-record selective replication mechanism by designating one copy of a record as the master and directing all updates of the record to its master copy. In this record-level mastering mechanism, mastership is assigned on a record-by-record basis, and different records in the same table can be mastered in different clusters. Each record maintains a metadata field that stores the identity of the current master. When a replica receives an update request, it first reads the record to determine if it is the master, and if not, to which replica to forward the request to. The mastership of a record can migrate between replicas according to its access pattern. All updates are propagated to non-master replicas by publishing them to the message broker and once the update is published, the system treats the transaction as committed. A master publishes its updates to a single broker, and thus updates are delivered to replicas in commit order. Lloyd et al. [38] presented the design and implementation of *COPS* (Clusters of Order-Preserving Servers), a key-value store that delivers a *causal+* consistency guarantee [8] across the wide-area. A key contribution of COPS is its scalability, which can enforce causal dependencies between keys stored across an entire cluster, rather than a single server. The central approach in COPS is tracking and explicitly checking whether causal dependencies between keys are satisfied in the local cluster before exposing writes. Walter [47] is another geo-replicated key-value store that supports transactions and ensures an isolation property called Parallel Snapshot Isolation (PSI) that provides a balance between consistency and latency. With PSI, hosts within a site observe transactions according to a consistent snapshot and a common ordering of transactions. Across sites, PSI enforces only causal ordering, not a global ordering of transactions, allowing the system to replicate transactions asynchronously across sites. Walter uses multi-version concurrency control within each site, and it can quickly commit transactions that write objects at their preferred sites. For other transactions, Walter resorts to two-phase commit to check for conflicts.

Google Megastore [5] has been presented as a scalable and highly available datastore which is designed to meet the storage requirements of large scale interactive Internet services. It relies on the *Paxos* protocol [17], a proven optimal fault-tolerant consensus algorithm with no requirement for a distinguished master, for achieving synchronous wide area replication. Megastore's replication mechanism provides a single, consistent view of the data stored in its underlying database replicas. Megastore replication semantics is done on *entity group* basis, a priori grouping of data for fast operations, basis by synchronously replicating the group's transaction log to a quorum of replicas. In particular, it uses a write-ahead log replication mechanism over a group of symmetric peers where any node can initiate reads and writes. Each log append blocks on acknowledgments from a majority of replicas, and replicas in the minority catch up as they are able. Kraska et al. [35] have proposed the *MDCC* (*M*ulti-*D*ata *C*enter *C*onsistency) commit protocol for providing strongly consistent guarantees at a cost comparable to eventually consistent protocols. In particular, in contrast to transactional consistency two-phase commit protocol (2PC), MDCC is designed to commit transactions in a single round-trip across data centers in the normal operational case. It also

does not require a master node so that apply reads or updates from any node in any data center by ensuring that every commit has been received by a quorum of replicas. It does not also impose any database partitioning requirements. The MDCC commit protocol can be combined with different read guarantees where the default configuration is to guarantee read committed consistency without lost updates. In principle, we believe that the problem of data replication and consistency management across different data centers in the cloud environment has, thus far, not attracted sufficient attention from the research community, and it represents a rich direction of future research and investigation.

5.3 Live migration

In general, live migration is an important component of the emerging cloud computing paradigm. It provides extreme versatility for management of cloud resources by allowing applications to be transparently moved across physical machines with a consistent state. In particular, the advantages of live migration techniques are manifold. For example, it can be used to improve compliance with tenant's SLA by migrating the tenant with excessive workload to a less loaded server. Thus, it is a main tool for achieving elasticity and dynamic provisioning. It is also used to ensure availability by migrating tenants to other servers when the host server is planned to go down for maintenance. Moreover, it can be used to consolidate multiple tenants onto a relatively idle server which alleviate the need of extra servers that can be shut down and thus reduce the operating costs. On the other side, live migration is a resource-intensive operation and can come at a price of degraded service performance during migration due the overhead caused by the extra CPU cycles which are consumed on both of the source and the destination servers in addition to the extra amount of network bandwidth which is consumed for the transmission process.

In general, the performance of a migration process is often measured by two main metrics:

1. The *down time* metric which represents the duration when the application is completely stopped and its application's service is entirely unavailable.
2. The *migration time* which represents the total time for all the involved migration process.

In practice, there is always a non-trivial trade-off between minimizing the total duration of the migration process and maintaining an acceptable quality of service during the migration process. In principle, there are two main techniques for database migration [29]:

1. The *Stop and Copy* technique represents the simplest form of migration where the system stops serving updates for the database, takes a snapshot of the database to be moved, moving and loading the data onto the new server, and finally restarting the service operations at the destination. This technique incurs a long service interruption and down time where the length of this period is proportional to the database size. However, it has a main advantage of simplicity and efficiency in terms of minimizing the amount of data transferred and the total migration time no data transfer overhead is involved in this approach.
2. The *Iterative State Replication* that uses an iterative approach where the checkpoint is created and iteratively copied. The source checkpoints the tenant's database and starts migrating the checkpoint to the destination, while it continues serving requests. While the destination loads the checkpoint, the source maintains the differential changes which are iteratively copied until the amount of change to be transferred is small enough or a maximum iteration count is reached. At this point, a final stop and copy is performed. Clearly, this technique incurs a small down time. However, it requires higher consumption of the computing resources due to the overhead incurs during the longer total migration time.

Zephyr [29] is a technique that have been proposed to efficiently migrate a live database in a shared nothing transactional database architecture. It minimizes the down time for the database being migrated by introducing a synchronized dual mode that allows both the source and destination to simultaneously execute transactions. The migration process starts with the transfer of the tenant's metadata to the destination which can then start serving new transactions while the source completes the transactions that were active when migration started. During this time, a methodology of on-demand pull and asynchronous push of data is followed where the source node, initially, owns the read/write access to all pages at the start and the destination nodes acquire the read/write page ownership on-demand as transactions at the destination access those pages. In addition, lightweight synchronization between the source and the destination to guarantees the serializability of transaction execution. Once the source node completes execution of all active transactions, migration completes with the ownership transfer of all database pages owned by the source to the destination node. Therefore, Zephyr can guarantee no service unavailability and few or no aborted transactions. A very similar approach to Zephyr, called *Albatross* [26], has been also proposed which is more focused on a shared process multitenant database environment where the persistent database image is stored in a network attached storage. Since the storage is shared, Albatross is more focused on optimizing the migration process by copying the cache during migration such that the destination starts with a warm cache and thus it can minimize any impact on transaction latency after migration.

Slacker [6] is another migration technique that leverages the off-the-shelf hot backup tools to achieve live migration with effectively nearly zero down-time in a shared process multi-tenancy environment. It tries to minimize the performance impact of the migration process on both the migrating tenant and collocated tenants, on the source and destination servers, by leveraging *migration slack*, resources that can be used for migration without excessively impacting performance latency. In particular, the migration process in Slacker is performed in three main steps: (1) The *initial snapshot transferring step* where Slacker streams the snapshot generated by off-the-shelf backup tool to the target server while the source continues to service queries. (2) The *delta updating step* which applies several rounds of deltas migration from the source node to the target node in order to bring the target node to the up-to-date at point of the source node. (3) The *handover step* which is executed once deltas between the source and destination are sufficiently small by performing a *freeze-and-handover* process in which the source is frozen, the final delta is copied, and the target becomes the new authoritative tenant. In general, the migration process adds an extra overhead on both of the source and target servers. Since the majority of the resource cost in migration is from reading, writing, or sending a large amount of data, Slacker control the effect of the migration process by controlling the upper bound rate of data transfer to the point at which performance latency steadily increases indicating that the database is overloaded and cannot maintain both its workloads and migrations. To achieve this goal, it applies a *PID* (*P*roportional-*I*ntegral-*D*erivative) controlling mechanism that allows the ability to automatically detect and exploit the available migration slack of computing resources in real time according to the dynamics of the executed workloads on both of the source and destination servers.

The *Dolly* [16] system has introduced a live migration technique for virtualized database servers where each database replica runs in a separate virtual machine. To create a new replica, Dolly clones the entire virtual machine of an existing replica, including the operating environment, the database engine with all its configuration, settings and the database itself. The cloned virtual machine is started on a new physical server, resulting in a new replica, which then synchronizes state with other replicas prior to processing application requests. In general, creating a new database replica is a time consuming process which increases proportionally with the size of the replicated database. In order to tackle this challenge, Dolly incorporates a model to estimate the latency to create a new database replica based on the snapshot size of the virtual machine and the database resynchronization latency and uses this model to trigger the replication process well in advance of its necessity to occur according to the anticipated workload increase.

In principle, live migration of databases in a timely fashion is a challenging task. In addition, there is a tradeoff between the migration time, the size of the database and the amount of update transactions in the workload which are executed during the migration process. Although various techniques have been proposed to tackle the challenge of *how* to migrate, very little attention from the proposed database migration mechanisms has been given to other important aspects such as *when* to migrate. In addition, in a multi-tenancy environment, the challenges of deciding *which* tenant to migrate and *where* (to which server) this tenant should be migrated to remain open issues for further investigation and careful consideration. Curino et al. [24] have outlined different strategies that can be used to improve the performance of live migration process such as: making use of database partitioning where the data to be moved into a number of small partitions and incrementally migrating these smaller partitions, exploiting existing replicas to serve read-only queries during migration and prefetching of data to prepare warm stand-by copies. These issues also represent a rich direction of future research and investigation. Moreover, the problem of database live migration across different data centers, which is naturally a very expensive process, represents another very challenging aspect that have not been well addressed yet by the research community and is widely open for novel solutions and optimization mechanisms.

5.4 SLA management

An SLA is a contract between a service provider and its customers. *Service Level Agreements* (SLAs) capture the agreed upon guarantees between a service provider and its customer. They define the characteristics of the provided service including service level objectives (SLOs) (e.g. maximum response times) and define penalties if these objectives are not met by the service provider. In practice, flexible and reliable management of SLA agreements is of paramount importance for both of cloud service providers and consumers. For example, Amazon found that every 100 ms of latency costs them 1 % in sales and Google found that an extra 500 ms in search page generation time dropped traffic by 20 %. In addition, large enterprise web applications (e.g., eBay and Facebook) need to provide high assurances in terms of SLA metrics such as response times and service availability to their users. Without such assurances, service providers of these applications stand to lose their user base, and hence their revenues.

In general, SLA management is a common general problem for the different types of software systems which are hosted in cloud environments for different reasons such as the unpredictable and bursty workloads from various users in addition to the performance variability in the underlying

cloud resources [23, 45]. In practice, resource management and SLA guarantee falls into two layers: the *cloud service providers* and the *cloud consumers* (users of cloud services). In particular, the cloud service provider is responsible for the efficient utilization of the physical resources and guarantee their availability for their customers (cloud consumers). The cloud consumers are responsible for the efficient utilization of their allocated resources in order to satisfy the SLA of their customers (application end users) and achieve their business goals. The state-of-the-art cloud databases do not allow the specification of SLA metrics at the application nor at the end-user level. In practice, cloud service providers guarantee only the availability (uptime guarantees), but not the performance, of their services [4, 7, 28]. In addition, sometimes the granularity of the uptime guarantees is also weak. For example, the uptime guarantees of Amazon EC2 is on a per data center basis where a data center is considered to be unavailable if a customer can not access any of its instances or can not launch replacement instances for a contiguous interval of five minutes. In practice, traditional cloud monitoring technologies (e.g. *Amazon CloudWatch*) focus on low-level computing resources (e.g. *CPU speed*, *CPU utilization*, *I/O disk speed*). In general, translating the SLO of software application to the thresholds of utilization for low-level computing resources is a very challenging task and is usually done in an ad-hoc manner due to the complexity and dynamism inherent in the interaction between the different tiers and components of the system. Furthermore, cloud service providers do not automatically detect SLA violation and leave the burden of providing the violation proof on the customer [7].

In the multi-tenancy environment of DaaS, it is an important goal for DaaS providers to promise high performance to their tenants. However, this goal normally conflicts with another goal of minimizing the overall running servers and thus operating costs by tenant consolidation. In general, increasing the *degree* of multi-tenancy (number of tenants per server) is normally expected to decrease per-tenant allocated resources and thus performance, but on the other side, it also reduces the overall operating cost for the DaaS provider and vice versa. Therefore, it is necessary, but challenging for the DaaS providers to balance between the performance that they can deliver to their tenants and the data center's operating costs. Several provider-centric approaches have been proposed to tackle this challenge. Chi et al. [19] have proposed cost-aware query scheduling algorithm, called *iCBS*, that takes the query costs derived from the service level agreements (SLA) between the service provider and its customers (in terms of response time) into account to make cost-aware scheduling decisions that aims to minimize the total expected cost. *SLA-tree* is another approach that have been proposed to efficiently support profit-oriented decision making of query scheduling. SLA-tree uses the information about the buffered queries which are waiting to be executed in addition to the service level agreement (SLA) for each query that indicates the different profits for the query for varying query response times and provides support for the answering of certain profit-oriented "*what if*" type of questions. Lang et al. [36] presented a framework that takes as input the tenant workloads, their performance SLA, and the server hardware that is available to the DaaS provider, and produces server characterizing models that can be used to provide constraints into an optimization module. By solving this optimization problem, the framework provides a cost-effective hardware provisioning policy and a tenant scheduling policy on each hardware resource. The main limitation of this approach is that the input information of the tenant workloads is not always easy to specify and model accurately. *PIQL* [3] (*P*erformance *I*nsightful *Q*uery *L*anguage) is a declarative language that has been proposed with a SLA compliance prediction model. The PIQL query compiler uses static analysis to select only query plans where it can calculate the number of operations to be performed at every step in their execution. In particular, PIQL extends SQL to allow developers to provide extra bounding information to the compiler. In contrast to traditional query optimizers, the objective of the query compiler is not to find the fastest plan but to avoid performance degradation. Thus, the compiler choose a potentially slower bounded plan over an unbounded plan that happens to be faster given the current database statistics. If the PIQL compiler cannot create a bounded plan for a query, it warns the developer and suggests possible ways to bound the computation. The *CloudDB AutoAdmin* framework [42] is more focused towards the consumer-centric view where traditional single tenant database with multiple replicas are hosted on virtualized database servers in cloud environments. The framework continuously monitors the database workload, tracks the satisfaction of the application-defined SLA, evaluates the condition of the application-defined action rules which are defined to maintain the application SLA when violations are detected, and executes the necessary actions when required. Therefore, it provides software application with more control on managing the SLA requirements of the database tier of their applications.

In general, adequate SLA monitoring strategies and timely detection of SLA violations represent challenging research issues in the cloud computing environments. Salman [7] has suggested that it may be necessary, in the future, for cloud providers to offer performance based SLAs for their services with a tiered pricing model, and charge a premium for guaranteed performance. While this could be one of the directions to solve this problem, we believe that it is a very challenging goal to delegate the management of the fine-granular SLA requirements of the consumer applications to the side of the cloud service provider due to the wide heterogeneity in the workload characteristics, details

and granularity of SLA requirements, and cost management objectives of the very large number of consumer applications (tenants) that can be simultaneously running in a cloud environment. Therefore, it becomes a significant issue for the cloud consumers to be able to monitor and adjust the deployment of their systems if they intend to offer viable service level agreements (SLAs) to their customers (end users). It is an important requirement for cloud service providers to enable the cloud consumers with a set of facilities, tools and framework that ease their job of achieving this goal effectively.

5.5 Transaction support

A transaction is a core concept in the data management world that represents a set of operations which are required to be executed *atomically* on a single consistent view of a database [31]. In general, the expertise gained from building distributed database systems by researchers and practitioners have shown that supporting distributed transactions does not allow building scalable and available system [50]. Therefore, to satisfy the scalability requirements of large scale internet services, many systems have sacrificed the ability to support distributed transactions. For example, most of the NoSQL systems (e.g. Bigtable, Dynamo, SimpleDB) supports atomic access only at the granularity of single keys. This design choice allows these systems to horizontally partition the tables, without worrying about the need for distributed synchronization and transaction support. *Microsoft SQL Azure Database* [10] supports the relational data model and ACID transactions. However, it requires manual data partitioning and does not support distributed transactions or queries across multiple data partitions located in different servers. In particular, database size is constrained to fit on a single node. For larger data sets, an application needs to partition the data among different database instances. While many web applications can live with single key access patterns [18, 27], many other applications (e.g. payment, auction services, online gaming, social networks, collaborative editing) would require atomicity guarantee on multi key accesses patterns. In practice, leaving the burden of ensuring transaction support to the application programmer normally leads to increased code complexity, slower application development, and low-performance client-side transaction management. Therefore, one of the main challenges of cloud-hosted database systems that has been considered is to support transactional guarantees for their applications without compromising the scalability property as one of the main advantages of the cloud environments.

The *G-Store* system [25] has been presented as a scalable data store which provides transactional multi key access guarantees over non-overlapping groups of keys using a key-value store. The main idea of GStore is the *Key Group* abstraction that defines a relationship between a group of keys and represents the granule for on-demand transactional access. This abstraction allows the Key Grouping protocol to collocate control for the keys in the group to allow efficient access to the group of keys. In particular, the Key Grouping protocol enables the transfer of ownership for all keys in a group to a single node which then efficiently executes the operations on the Key Group. At any instance of time, each key can only belong to a single group and the Key Group abstraction does not define a relationship between two groups. Thus, groups are guaranteed to be independent of each other and the transactions on a group guarantee consistency only within the confines of a group. The Key Grouping protocol ensures that the ownership of the members of a group reside with a single node. Thus, the implementation of the transaction manager component does not require any distributed synchronization and is similar to the transaction manager of any single node relational database management systems. The key difference is that in G-Store, transactions are limited to smaller logical entities (key groups). A similar approach has been followed by the *Google Megastore* system [5]. It implements a transactional record manager on top of the BigTable data store [18] and provides transaction support across multiple data items where programmers have to manually link data items into hierarchical groups and each transaction can only access a single group. Megastore partitions the data into a collection of *entity groups*, a priori user-defined grouping of data for fast operations , where each group is independently and synchronously replicated over a wide area. In particular, Megastore tables are either entity group root tables or child tables. Each child table must declare a single distinguished foreign key referencing a root table. Thus, each child entity references a particular entity in its root table (called the root entity). An entity group consists of a root entity along with all entities in child tables that reference it. Entities within an entity group are mutated with single- phase ACID transactions (for which the commit record is replicated via Paxos). Operations across entity groups could rely on expensive two-phase commit operations but they could leverage the built-in Megastore's efficient asynchronous messaging to achieve these operations. Similar to Megastore, *Microsoft SQL Azure Database* [10] requires that a *table group* (i.e., a user database) is either keyless, meaning that its tables are co-located, or it is keyed, meaning that its tables have a common partitioning key, and that every update transaction reads and writes records with a single value of that partitioning key. This ensures that every transaction can be executed on one server.

Deuteronomy [37] have presented a radically different approach towards scaling databases and supporting transactions in the cloud by *unbundling* the database into two components: (1) The *transactional component* (TC) that manages transactions and their concurrency control and

undo/redo recovery but knows nothing about physical data location. (2) The *data component* (DC) that maintains a data cache and uses access methods to support a record-oriented interface with atomic operations but knows nothing about transactions. Applications submit requests to the TC which uses a lock manager and a log manager to logically enforce transactional concurrency control and recovery. The TC passes requests to the appropriate Data Component (DC). The DC, guaranteed by the TC to never receive conflicting concurrent operations, needs to only support atomic record operations, without concern for transaction properties that are already guaranteed by the TC. In this architecture, data can be stored anywhere (e.g., local disk, in the cloud, etc) as the TC functionality in no way depends on where the data is located. The TC and DC can be deployed in a number of ways. Both can be located within the client, and that is helpful in providing fast transactional access to closely held data. The TC could be located with the client while the DC could be in the cloud, which is helpful in case a user would like to use its own subscription at a TC service or wants to perform transactions that involve manipulating data in multiple locations. Both TC and DC can be in the cloud, which is helpful if a cloud data storage provider would like to localize transaction services for some of its data to a TC component. There can be multiple DCs serviced by one TC, where transactions spanning multiple DCs are naturally supported because a TC does not depend on where data items are stored. Also, there can be multiple TCs, yet, a transaction is serviced by one specific TC.

5.6 Benchmarking

With the emergence of cloud database services, several studies have attempted to assess the performance and scalability of cloud computing solutions for data management applications. Kossmann et al. [33] have used the TPC-W benchmark to evaluate the performance of different database architectures for processing OLTP workload in commercial database services (e.g. Amazon RDS, Amazon SimpleDB, Microsoft SQL Azure). The results of the experiments have shown that the cost, performance and the scalability of the cloud services vary significantly depending on the characteristics of the workload (e.g. read/write ratio). *AppScale* [13] is an open source implementation of the Google App Engine (GAE) Datastore which unifies access to a wide range of open source distributed database technologies. AppScale has been used for conducting an evaluation of the performance characteristics of several NoSQL systems including: HBase, Hypertable, Cassandra, Voldemort and MongoDB. The Yahoo! Cloud Serving Benchmark (YCSB) [23] is another effort for benchmarking cloud serving systems. The benchmark consists of a package of workloads with different characteristics (e.g. read-heavy workloads, write-heavy workloads, scan workloads, etc). The initial implementation of the YCSB benchmark has been used for evaluating four systems: Cassandra, HBase, PNUTS, and a simple sharded MySQL in terms of their performance and elasticity characteristics. The scope of the benchmark has been recently extended, YCSB++ [40], to support complex features such as including multi-tester coordination for increased load and eventual consistency measurement, multi-phase workloads to quantify the consequences of work deferment and the benefits of anticipatory configuration optimization such as B-tree pre-splitting or bulk loading. The YCSB++ features are used for evaluating two BigTable-like table stores: Apache HBase and Accumulo.[18] Wada et al. [53] presented an approach for measuring time-based staleness by writing timestamps to a key from one client, reading the same key and computing the difference between the reader's local time and the timestamp read. However, this approach is very primitive and is unsuitable in a production environment. In particular, if the workload is such trivial and uses only a single writer then all operations will be just atomic and the workload clearly does not cover any complex or special execution paths that the underlying storage system need to deal with under heavy or concurrent workloads. These limitations hurt the accuracy of the reported measurements. Bermbach and Tai [9] have tried to address a side of these limitations by extending the experiments of [53] using a number of readers which are geographically distributed. They measure the consistency window by calculating the difference between the latest read timestamp of version n and the write timestamp of version $n+1$. Their experiments with that Amazon S3 showed that the system frequently violates monotonic read consistency.

In principle, benchmarks need to play an effective role in empowering cloud users to make better decisions regarding choosing the adequate systems and technologies that suit their application's requirements. In general, designing a good benchmark is a challenging task due to the many aspects that should be considered and can influence the adoption and the usage scenarios of the benchmark. In particular, a benchmark is considered to be good if it can provide true and meaningful results for all of its stakeholders [30]. We believe that it is important that cloud users become able to paint a comprehensive picture of the relationship between the capabilities of the different type of cloud database services, the application characteristics and workloads, and the geographical distribution of the application clients and the underlying database replicas. As yet, the literature does not contain any comprehensive assessments and measurements of the performance, scalability, elasticity or consistency guarantees of the different categories of cloud database services. This is a clear gap that we suggest to attract more attention from the research community.

[18] http://wiki.apache.org/incubator/AccumuloProposal.

Table 2 Open research challenges of cloud-hosted database systems

Research aspect	Related factors	Open research challenges
Elasticity management	– Application workload – SLA Satisfaction – Monetary Cost – Side of Control (Provider or Consumer)	– Designing accurate models for characterizing and predicting Internet scale application workloads. – Designing flexible dynamic provision mechanisms that carefully consider the target consumer application SLA and the target monetary costs. – Enabling the consumer applications with powerful and flexible tools (admission controllers) to declaratively define and control their elasticity policies.
Data replication and consistency management	– CAP Theorem – Levels of Consistency Guarantee – Replica Locations	– Designing adaptable consistency mechanisms that can be flexibly configured on the runtime according to the application context. – Designing efficient data replication and consistency management protocols across different data centers in the cloud environment. – Further understanding to the practical limits of the CAP theorem.
Live migration	– Down Time – Migration Time – SLA Effect – Triggering of Migration Need (When to Migerate?)	– Optimizing the down time and migration time metrics of the live migration techniques. – Minimizing the performance effect and SLA degradation of the co-located tenants during the migration process. – Designing partitioning-aware live database migration techniques. – Designing intelligent schedulers for the activities of the migration processes. – Designing intelligent techniques for deciding the optimal source and destination tenants and servers with aim of optimizing the overall system performance and the overall utilization of the computing resources.
SLA management	– Side of Control – SLA Granularity – Monetary Cost	– Designing efficient mechanisms for monitoring and timely detecting SLA violations in cloud environments. – Providing fine-granular SLA guarantees for cloud hosting database services. – Designing cost-aware SLA management techniques. – Enabling the consumer applications with flexible mechanisms to declaratively define, monitor and control their SLA requirements.
Transaction support	– Granularity of Atomicity – Distributed Transactions – Performance	– Providing efficient multi row atomicity guarantees on NoSQL systems. – Designing intelligent workload-aware and transaction-aware database partitioning mechanisms for cloud-hosted databases. – Providing scalable transactional guarantees over multiple partitions for distributed database (across different data centers) in cloud environments.

6 Conclusion

This article presented an overview of the state-of-the-art of existing technologies for hosting the database tier of software applications in cloud environments. We crystallized the design choices, strengths, weaknesses of each technology. In addition, we discussed a set of novel challenges, that have been brought on by the reliance on cloud computing platforms and faced by application developers and designers of cloud database systems, and pointed out alternative research directions for tackling them. Table 2 summarizes some of the open research challenges along with the key related factors which could influence the design of their solutions. We believe that this article could be valuable for both the designers and developers of new cloud-hosted database systems and interested users of cloud database services as well. For user of cloud database services, we can draw the following recommendations:

- NoSQL systems are viable solutions for applications that require scalable data stores which can easily scale out over multiple servers and support flexible data model and storage scheme. However, the access pattern of these applications should not require much join operations and can work with limited transaction support and weaker consistency guarantees. In general, NoSQL systems are recommended for newly developed applications but not for migrating existing applications which are written on top of traditional relational database systems. For example, Amazon Web Services describe the anti-patterns for using its cloud-hosted NoSQL solution, SimpleDB, to include: pre-developed software applications which are tied to traditional relational database or applications that may require many join operations and complex transactions.[19]

[19] http://aws.amazon.com/whitepapers/storage-options-aws-cloud/.

In addition, with the wide options and variety of currently available NoSQL systems, software developers need to well understand the requirements of their application to choose the NoSQL system with adequate design decisions of their applications.

- Database-as-a-Service solutions are recommended for software applications which are built on top of relational databases. They can be easily migrated to cloud servers and alleviate the need to purchase expensive hardware, deal with software upgrades and hire professionals for administrative and maintenance tasks. However, these application should have the ability to accurately predict their application workloads and provision the adequate computing resources that can achiever their performance requirements. Unfortunately, these applications should be ready to not automatically leverage the elasticity and scalability promises of cloud services.
- Virtualied database servers are recommended for software applications which require to leverage the full elasticity and scalability promises of cloud services and need to have full control on the performance of their applications. However, these application need to build and configure their admission control for managing the database tier of their applications.

For designers and developers, it is clear that there is no single perfect technology or solution for hosting databases in cloud platforms. Different application target different aspects in the design space, and multiple open problems still remain. Therefore, they can use the challenges which are discussed in this article to effectively decide on the points which can be improved in order to make an effective contribution towards the vision of designing and implementing successful data management solutions in the cloud environment. We believe that there is still many opportunities for new innovations and optimizations in this area. For users of cloud database services, they often have the challenge of choosing the appropriate technology and system that can satisfy their specific set of application requirements. Therefore, a thorough understanding of current cloud database technologies is essential for dealing with this situation. Hence, we believe that this article could be helpful on guiding those users for making an effective decision to select the most suitable technology for the requirements of their software applications.

References

1. Abadi, D.J.: Data management in the cloud: limitations and opportunities. IEEE Data Eng. Bull. **32**(1) (2009). http://sites.computer.org/debull/A09mar/abadi.pdf
2. Agrawal, D., El Abbadi, A., Emekçi, F., Metwally, A.: Database management as a service: challenges and opportunities. In: ICDE (2009)
3. Armbrust, M., Curtis, K., Kraska, T., Fox, A., Franklin, M.J., Patterson, D.A.: PIQL: success-tolerant query processing in the cloud. Proc. VLDB Endow. **5**(3), 181–192 (2011)
4. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R.H., Konwinski, A., Lee, G., Patterson, D.A., Rabkin, A., Stoica, I., Zaharia, M.: Above the clouds: a Berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, University of California, Berkeley (2009)
5. Baker, J., Bond, C., Corbett, J., Furman, J.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)
6. Barker, S.K., Chi, Y., Moon, H.J., Hacigümüs, H., Shenoy, P.J.: "Cut me some slack": latency-aware live migration for databases. In: EDBT, pp. 432–443 (2012)
7. Baset, S.A.: Cloud SLAs: present and future. Oper. Syst. Rev. **46**(2), 57–66 (2012)
8. Belaramani, N., Dahlin, M., Gao, L., Nayate, A., Venkataramani, A., Yalagandula, P., Zheng, J.: Practi replication. In: NSDI (2006)
9. Bermbach, D., Tai, S.: Eventual consistency: how soon is eventual? An evaluation of Amazon s3's consistency behavior. In: Proceedings of the 6th Workshop on Middleware for Service Oriented Computing (2011)
10. Bernstein, P.A., Cseri, I., Dani, N., Ellis, N., Kalhan, A., Kakivaya, G., Lomet, D.B., Manne, R., Novik, L., Talius, T.: Adapting microsoft SQL server for cloud computing. In: ICDE, pp. 1255–1263 (2011)
11. Bodík, P., Fox, A., Franklin, M.J., Jordan, M.I., Patterson, D.A.: Characterizing, modeling, and generating workload spikes for stateful services. In: SoCC, pp. 241–252 (2010)
12. Brewer, E.A.: Towards robust distributed systems (abstract). In: PODC, p. 7 (2000)
13. Bunch, C., Chohan, N., Krintz, C., Chohan, J., Kupferman, J., Lakhina, P., Li, Y., Nomura, Y.: An evaluation of distributed datastores using the AppScale cloud platform. In: IEEE CLOUD, pp. 305–312 (2010)
14. Cattell, R.: Scalable SQL and NoSQL data stores. SIGMOD Rec. (2010). doi:10.1145/1376616.1376691
15. Cecchet, E., Candea, G., Ailamaki, A.: Middleware-based database replication: the gaps between theory and practice. In: SIGMOD Conference, pp. 739–752 (2008)
16. Cecchet, E., Singh, R., Sharma, U., Shenoy, P.J.: Dolly: virtualization-driven database provisioning for the cloud. In: VEE (2011)
17. Chandra, T.D., Griesemer, R., Redstone, J.: Paxos made live: an engineering perspective. In: PODC, pp. 398–407 (2007)
18. 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. ACM Trans. Comput. Syst. (2008). doi:10.1145/1365815.1365816
19. Chi, Y., Moon, H.J., Hacigümüs, H.: ICBS: incremental costbased scheduling under piecewise linear SLAs. Proc. VLDB Endow. **4**(9), 563–574 (2011)
20. Cipar, J., Ganger, G.R., Keeton, K., Morrey, C.B., Soules, C.A.N., Veitch, A.C.: LazyBase: trading freshness for performance in a scalable database. In: EuroSys, pp. 169–182 (2012)
21. Cooper, B.F., Baldeschwieler, E., Fonseca, R., Kistler, J.J., Narayan, P.P.S., Neerdaels, C., Negrin, T., Ramakrishnan, R., Silberstein, A., Srivastava, U., Stata, R.: Building a cloud for Yahoo! IEEE Data Eng. Bull. **32**(1) (2009). http://sites.computer.org/debull/A09mar/cooper1.pdf
22. 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)

23. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: SoCC (2010)
24. Curino, C., Jones, E.P.C., Popa, R.A., Malviya, N., Wu, E., Madden, S., Balakrishnan, H., Zeldovich, N.: Relational cloud: a database service for the cloud. In: CIDR, pp. 235–240 (2011)
25. Das, S., Agrawal, D., El Abbadi, A.: G-Store: a scalable data store for transactional multi key access in the cloud. In: SoCC, pp. 163–174 (2010)
26. Das, S., Nishimura, S., Agrawal, D., El Abbadi, A.: Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration. Proc. VLDB Endow. **4**(8), 494–505 (2011)
27. 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)
28. Durkee, D.: Why cloud computing will never be free. Commun. ACM (2010). doi:10.1145/1735223.1735242
29. Elmore, A.J., Das, S., Agrawal, D., El Abbadi, A.: Zephyr: live migration in shared nothing databases for elastic cloud platforms. In: SIGMOD Conference, pp. 301–312 (2011)
30. Gray, J. (ed.): The Benchmark Handbook for Database and Transaction Systems, 1st edn. Morgan Kaufmann, San Mateo (1991)
31. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, San Mateo (1992)
32. Jacobs, D., Aulbach, S.: Ruminations on multi-tenant databases. In: BTW, pp. 514–521 (2007)
33. Kossmann, D., Kraska, T., Loesing, S.: An evaluation of alternative architectures for transaction processing in the cloud. In: SIGMOD Conference, pp. 579–590 (2010)
34. Kraska, T., Hentschel, M., Alonso, G., Kossmann, D.: Consistency Rationing in the Cloud: Pay only when it matters. Proc. VLDB Endow. **2**(1) (2009). http://www.vldb.org/pvldb/2/vldb09-759.pdf
35. Kraska, T., Pang, G., Franklin, M.J., Madden, S.: MDCC: multi-data center consistency. In: CoRR, arXiv:1203.6049 abs (2012)
36. Lang, W., Shankar, S., Patel, J.M., Kalhan, A.: Towards multi-tenant performance SLOs. In: ICDE, pp. 702–713 (2012)
37. Levandoski, J.J., Lomet, D.B., Mokbel, M.F., Zhao, K.: Deuteronomy: transaction support for cloud data. In: CIDR, pp. 123–133 (2011)
38. Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In: SOSP, pp. 401–416 (2011)
39. Minhas, U.F., Yadav, J., Aboulnaga, A., Salem, K.: Database systems on virtual machines: how much do you lose? In: ICDE Workshops, pp. 35–41 (2008)
40. Patil, S., Polte, M., Ren, K., Tantisiriroj, W., Xiao, L., López, J., Gibson, G., Fuchs, A., Rinaldi, B.: YCSB++: benchmarking and performance debugging advanced features in scalable table stores. In: SoCC (2011)
41. Pritchett, D.: BASE: an acid alternative. ACM Queue **6**(3), 48–55 (2008)
42. Sakr, S., Liu, A.: SLA-based and consumer-centric dynamic provisioning for cloud databases. In: IEEE CLOUD, pp. 360–367 (2012)
43. Sakr, S., Liu, A.: Is your cloud-hosted database truly elastic? In: IEEE 9th World Congress on Services (2013)
44. Sakr, S., Liu, A., Batista, D.M., Alomari, M.: A survey of large scale data management approaches in cloud environments. IEEE Commun. Surv. Tutor. **13**(3), 311–336 (2011)
45. Schad, J., Dittrich, J., Quiané-Ruiz, J.-A.: Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. Proc. VLDB Endow. **3**(1) (2010)
46. Soror, A.A., Minhas, U.F., Aboulnaga, A., Salem, K., Kokosielis, P., Kamath, S.: Automatic virtual machine configuration for database workloads. In: SIGMOD Conference (2008)
47. Sovran, Y., Power, R., Aguilera, M.K., Li, J.: Transactional storage for geo-replicated systems. In: SOSP, pp. 385–400 (2011)
48. Stonebraker, M.: One size fits all: an idea whose time has come and gone. Commun. ACM **51**(12), 76 (2008)
49. Suleiman, B., Sakr, S., Jeffrey, R., Liu, A.: On understanding the economics and elasticity challenges of deploying business applications on public cloud infrastructure. J. Internet Serv. Appl. **3**(2), 173–193 (2012)
50. Tamer Ozsu, M., Valduriez, P.: Principles of Distributed Database Systems, 3rd edn. Springer, Berlin (2011)
51. Tatemura, J., Po, O., Hacigümüs, H.: Microsharding: a declarative approach to support elastic OLTP workloads. Oper. Syst. Rev. **46**(1), 4–11 (2012)
52. Vogels, W.: Eventually consistent. ACM Queue **6**, 14–19 (2008)
53. Wada, H., Fekete, A., Zhao, L., Lee, K., Liu, A.: Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective. In: CIDR (2011)
54. Xiong, P., Chi, Y., Zhu, S., Moon, H.J., Pu, C., Hacigümüs, H.: Intelligent management of virtualized resources for database systems in cloud environment. In: ICDE, pp. 87–98 (2011)
55. Zhao, L., Sakr, S., Fekete, A., Wada, H., Liu, A.: Application-managed database replication on virtualized cloud environments. In: Data Management in the Cloud (DMC), ICDE Workshops (2012)
56. Zhao, L., Sakr, S., Liu, A.: Application-managed replication controller for cloud-hosted databases. In: IEEE CLOUD, pp. 922–929 (2012)
57. Zhao, L., Sakr, S., Liu, A.: A framework for consumer-centric SLA management of cloud-hosted databases. IEEE Trans. Serv. Comput. (2013)

Sherif Sakr is a Senior Researcher in the Software Systems Research Group at National ICT Australia (NICTA), ATP lab, Sydney, Australia. He is also a Senior Lecturer in The School of Computer Science and Engineering (CSE) at University of New South Wales (UNSW). He received his Ph.D. degree in Computer and Information Science from Konstanz University, Germany in 2007. He received his B.Sc. and M.Sc. degrees in Computer Science from the Information Systems department at the Faculty of Computers and Information in Cairo University, Egypt, in 2000 and 2003 respectively. In 2008 and 2009, Sherif held an Adjunct Lecturer position at the Department of Computing of Macquarie University. In 2011, he held a Visiting Researcher position at the eXtreme Computing Group, Microsoft Research Laboratories, Redmond, WA, USA. In 2012, Sherif held a Research MTS position in Alcatel-Lucent Bell Labs. Dr. Sakr's research interest is data and information management in general, particularly in areas of indexing techniques, query processing and optimization techniques, graph data management, social networks, data management in cloud computing.