SQL Azure as a Self-Managing Database Service: Lessons Learned and Challenges Ahead

Kunal Mukerjee, Tomas Talius, Ajay Kalhan, Nigel Ellis and Conor Cunningham
Microsoft Corporation

Abstract

When SQL Azure was released in August 2009, it was the first database service of its kind along multiple axes, compared to other Cloud services: shared nothing architecture and log-based replication; support for full ACID properties; providing consistency and high availability; and by offering near 100% compatibility with on-premise SQL Server delivered a familiar programming model at cloud scale. Today, just over two years later, the service has grown to span six hosting regions across three continents; hosting large numbers of databases (in the order 100s of thousands), increasing more than 5x each year with 10s of thousands of subscribers. It is a very busy service, clocking more than 30 million successful logins over a 24 hour period. In this paper we reflect on the lessons learned, and the challenges we will need to face in future, in order to take the SQL Azure service to the next level of scale, performance, satisfaction for the end user, and profitability for the service provider.

1 Introduction

These are exciting times to be working on cloud services. The buzz has seeped beyond technical forums, and is now a common and recurring theme in the popular media [1]. Economic conditions seem to also favor widespread interest in leveraging cloud services to start new businesses and grow existing ones with frugal spends on infrastructure [2]. Since it went live in August 2009, SQL Azure, which was the first large scale commercial RDBMS server (Amazon's Relational Database Service, or RDS, which is a distributed relational database service, was first released on 22nd October, 2009), has been growing steadily to accommodate both external and (Microsoft) internal [e.g., see [13]) customers and workloads. Given that the SQL Azure service was the first of its kind along several different axes [3], [4], and few others have ventured into providing this specific mix of architectural orientation and service guarantees [5], we feel that the time is right to reveal what have been our internal pain points, lessons learned, and perception of what the future holds, as we continue to grow the service to support a larger customer base, scale to store and query bigger data, and push ahead on performance and SLAs. The rest of the paper is organized as follows: The rest of Section 1 first introduces the canonical application pedigree and customer expectations currently employing the service; and then sets the SQL Azure architectural context, specifically for detecting and handling failures throughout the service and underlying infrastructure. From the very outset, SQL Azure has been serving industry strength workloads. Its most widely publicized incarnation has been the SQL Azure service [7], in which a large number of independent

Copyright 2011 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

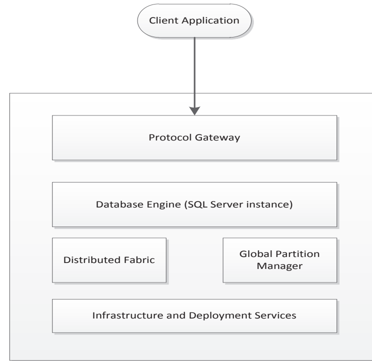

Figure 1: Overall system architecture

databases are served. Additionally, we hosted the Exchange Hosted Archive [9] on it, in which the infrastructure serves a massively scaled, multi-tenant, email archival service. As a result, we hit the ground running in terms of handling scale on all imaginable areas of the architecture, and that taught us a number of lessons in these past two years. We will describe some of these in Section 2. We describe challenges and future directions for the service in Section 3. Section 4 has our conclusions.

Canonical SQL Azure database: OLTP consolidation. In order to facilitate understanding of the material presented in this paper, it would be useful to introduce the reader to the kinds of databases, application contexts, and primary customer expectations our service aims to serve. OLTP consolidation is a predominant category of database applications being currently deployed on SQL Azure. The interesting characteristics of these are: 1) Each database's resource requirements are smaller than a single node in the cluster; 2) Relatively small nodes are used; and 3) Customers want to pay-as-they-go. Additionally: What pieces of the service experience do the bulk of SQL Azure customers place the most premium on? Our customers want a great deal. This is corroborated by the study conducted in [14], where Kossmann et al. concluded that Azure is currently the most affordable service for medium to large services. Our customers expect, and get this "good deal for the money" on commodity hardware, automatic management of core pieces of the system, and built-in High Availability (HA). We should note here, that SQL Azure is being used for non-OLTP scenarios as well. There are at least two Microsoft teams that are using it as their "OpsStore". TicketDirect is a good representative case study [15] of OLTP consolidation applications that are running on SQL Azure. Exchange Hosted Archive [13] is a representative of a non-OLTP workload (email archiving).

SQL Azure: Fault Tolerance Architecture. Cost factors have required delivering highly reliable services over inexpensive hardware with an operational model that is fully automated, in SQL Azure. Furthermore, failures of individual components are literally an everyday occurrence at this scale. To be a viable business for us, as service providers, the system must handle most faults without operator intervention. Figure 1 shows the top level view of SQL Azure's system architecture. As we explain below, the fundamental pillars of self-management and fault tolerance permeate each level of the system architecture.

Typically all user data is copied on three replica partitions. These data partitions are spread through multiple fault domains (racks and switches in the data center) and upgrade domains (units of software rollout), so that multiple replicas of the same partition are unlikely to go down at the same time. Each partition has at most one primary replica at any given moment of time, the other replicas are either secondary or are inactive. All user reads and writes originate at the primary replica.

The *Protocol Gateway* enables client applications to connect to the SQL Azure service. It coordinates with the distributed fabric to locate the primary, and renegotiates that choice in case a failure or system-initiated reconfiguration causes the election of a new primary. The *Database Engine*, which is an instance of SQL

Server, typically manages multiple locally attached disks, each of which is private to that instance. If one of the instance's disks fails, the watchdog processes which monitor hardware health will trigger appropriate failure recovery actions. In the SQL Azure database server, partitions are known as the *failover unit*. To attain high availability, we designed a highly reliable system management layer that, among other things, maintains up/down status of servers, detects server failures and orchestrates recoveries. These capabilities are built into the *Distributed Fabric* layer, which runs as a process on every server that runs a SQL Server instance. There is a highly-available *Global Partition Manager (GPM)* architectural module, which maintains a directory of information about partitions. For each partition P, there is a cluster-wide unique id. And for each replica R of P, it knows R's location and its state. If a hard failure is detected at any secondary, a new node is appointed and a copy of the partition is created on that node. If a primary dies, the GPM will elect one of the remaining secondary nodes to become the primary. Finally, the *Infrastructure and Deployment Services (IDS)* layer is responsible for hardware and software lifecycle management and monitoring. Hardware lifecycle includes such activities as taking bare metal machines and preparing them to play a role on the cluster, monitoring machine health, and taking it through repair cycles: reboot, reimage and return to manufacturer (RMA). The software lifecycle runs workflows to clean a cluster, to specify machine roles [4], deploy application software, deploy OS patches, etc.

Self-healing vs. self-managing. Our experience with boxed products shows that humans are the largest factor in operational cost. In a conventional setting you need humans to set up High Availability, orchestrate patches, plan upgrades, deal with hardware failures, etc. Cloud database systems need self-healing capabilities in order to scale to large numbers of clusters and large number of machines in each cluster. Without this we don't get efficiencies of scale. Current SQL Azure self-managing capabilities are in the following areas: 1) There are a number of cases in which SQL Server detects hardware failures. We convert these high severity exceptions into node failures; 2) System managed backups; 3) Periodic snapshots of SQL Server state on each node. These are used for trending and debugging; and 4) Throttling to prevent user requests from overwhelming the machine.

Experience from the box product also shows that managing database performance requires human intervention. In this context we will need to invest in traditional self-tuning capabilities. This includes auto-defragmentation, auto-indexing, etc. The multi-tenant nature of SQL Azure requires self-tuning capability in the form of resource governance. We have to ensure that workloads don't adversely affect each other and a workload which requires resource guarantees can be satisfied. Database systems will require a fair bit of investment to make them behave well in a multi-tenant setting. This is therefore, a rich area for research.

2 Lessons in Self-Management and Fault Tolerance

In the previous section, we outlined how the overall system design is fundamentally geared to support fault tolerance, and to favor automatic, rather than manual mitigation and fix of any failure. In this section we discuss a number of specific dimensions that are identifiable towards this general theme, leveraging the architecture, as well as service operations (i.e., people and processes).

After running the SQL Azure service over two years, we have come to realize that everything in hardware and software fails sooner or later, and sometimes in bizarre and unpredictable ways! And so the emphasis of the monitoring and process layer surrounding the hardware and software architecture outlined in Section 1 is primarily concerned with generating early warnings, detailed metrics at different granularities, and stopping contagion from failures, rather than trying to come up with ways to avoid failures from ever happening. Here we outline the major thrusts of this policy.

Define the metrics to instrument and monitor the cluster effectively. The right mix of health metrics should be detailed enough, provide aggregate views across logical and/or physical groups of machines, and yet be able

to drill into failure cases and hotspots to effectively monitor from a compact dashboard.

An overall emphasis of our monitoring strategy is to accurately detect and react rapidly to black and white failures every time, as well as detect "gray area" failures, and coerce them to either flag a serious enough condition where the component should be shut down completely, or to realize that it is a benign condition. A philosophical explanation of the above strategy is that we accept that there will always be gray areas in the failure landscape, and so canned responses can never be truly comprehensive. Our approach to this problem is to install redundant checks to increase confidence in the type and severity of each failure.

We use a mix of logging, automatic watchdogs that trigger on failure and near-failure conditions, as well as alerts that trigger emails and/or phone calls for severe and/or sustained failures. The machines originating failure are also taken through a three step escalation protocol, in which increasingly aggressive steps are taken to try and fix the problem: 1) reboot; 2) reimage; and 3) return to manufacturer (RMA). This provides a hierarchy of escalation, where if the service, by virtue of its intrinsic and designed resilience to failure can recover from a failure state, would not require human intervention – this is what is required in order to scale the service up, keep costs down for our end users (see also Section 1.1), and for us to sleep at night!

Take a calibrated approach to dealing with serious failures. Coming up with graded responses to failures of different varieties is something of a black art, which practitioners of a service learn over time.

To use specific examples from running SQL Azure: serious software issues like a program hitting an Access Violation (AV) are typically highly localized to a single machine and in response to a very specific mix of workloads. Under these failure conditions that are highly localized, it is generally appropriate to bring the machine down quickly, because otherwise, persisting in the AV state where wild accesses to memory may be a possibility, even more serious consequences like index corruption are reachable. This may be summarized into the following rule of thumb: *fail fast when the failure conditions are localized*.

However, in other situations, failing fast isn't the best approach, because it could lead to cascading failures. For example, one ought not to bring machines down due to bad or invalid configuration, because configurations are common across many machines, and that could result in (automatically) leading to more widespread failures such as quorum loss [4], or even bringing the entire service down.

Due to the above intricacies of taking carefully calibrated responses to failures, we have a role for central command and control, which allows humans in the loop to assess the impact of the failure/repair action(s), before initiating them. Another best practice that has evolved, simply as a result of running a database service, is to disallow combinations of input data and SQL statements that are already known (because they generated failure either on the live service or on a test cluster) to be generate problems or failures, until a fix (QFE) has been rolled out. SQL Azure has a mechanism known as *statement firewall*.

To generalize, the lessons we learned with respect to calibrated responses to failures is that *cascading failures are to be avoided* at all cost, and that safeguards need to be put in place to stop them. Once the scope of the failure exceeds some standard deviation (automation can help to figure out that is happening), the recommended policy is to get humans involved, rather than continuing to try to auto-resolve. We actually have an explicit abstraction known as the *Big Red Switch* in the system, which is our mechanism for turning off automatic repair or compensation operations when the system is clearly experiencing an unplanned problem.

Software-based defense in depth is appropriate when building on commodity hardware. As explained in Section 1.1, [3] and [4], one of the fundamental design points of SQL Azure was to build the entire service on top of commodity hardware, in order to provide an attractive price point for our customers, as well as improve our margins, as the service provider. However, the flip side of running on commodity hardware is the need to create higher service reliability by investing more on the software side, in terms of hardening the stack towards common hardware and network errors. Only then can the overall system have the level of resilience and recoverability needed to run a highly reliable and available service like SQL Azure.

Here are some of the things we needed to support in software as our *defense in depth* strategy – they would not be needed when building a modern database system at a different price point on the hardware stack, e.g., RAID5 or RAID10, or SANs. 1. Using checksums to verify data on disk – we learned this lesson the hard way, after observing some bit-flips on disks; 2. Periodically checking (by sampling) data on disk for disk rot, and ensuring that log handshakes are honored; 3. Signing all network communication, to guard against bugs in the network interface card (NIC) firmware – this was actually done after discovering faulty behavior of some NICs; and 4. Storing information (such as identity metadata) redundantly, thus allowing us to reconstruct / rebuild in the case of catastrophic failure. Examples are GPM rebuild and DB identity (server + database name).

To summarize our design philosophy with respect to defense in depth: when building on commodity hardware and network stacks, *trust nothing, verify everything*. Sometimes if you don't trust the stack below you and check everything, then one needs to also measure the cost of each such check. If a check is very expensive at a higher level of the stack then it might be possible to push it down the stack. In these cases, guarantees of reliability need to be negotiated between providers of different levels of the stack, and suitable levels of acceptance testing should be instituted to maintain the trustworthiness of these contracts, or "stack-internal SLAs". A simplification that helps implement the defense-in-depth policy cheaply is to simply take the view that we will *enforce homogeneity* across the cluster. This can be done through a multitude of relatively inexpensive ("hammer") implementation strategies, such as polling, diff-ing, etc., to explicitly check for homogeneity, and interpret that as an indicator of system health.

Build a failure classifier and playbook. Let each failure teach you something for the future. To offer examples, here are some of our most commonly recognizable failure types, and what we do with them:

Planned failures. These are the most common. Examples are application software upgrades, OS patches, firmware upgrades, and planned load balancing events. Handling updates without compromising availability and without breaking any application is a very important design tenet of SQL Azure, and is mediated by the IDS layer (Figure 1). Our *two-phase upgrade* scheme is described in [4]. One must also be careful to throttle upgrade actions, to maintain availability of the overall service. Also, where possible, it helps greatly if one is able to deconstruct a monolithic service or major component into several smaller services, and if these smaller services can be deployed independently of one another. This reduces the strain on the overall system by minimizing the scope of updating pieces of the service in small increments, and provides good opportunities for load balancing.

Handling upgrades well carries paramount importance because if they can be done very successfully and efficiently then we are able to release more often; and if we are able to release more often, then we can improve the service faster. Therefore, quality of upgrades translates directly into continuously improving the quality of the service, as well as adding to it in terms of features and functionality – these translate into a higher quality user experience, as well as better business health for the service provider.

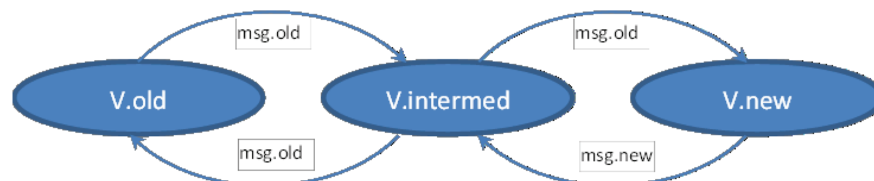

Figure 2: Two-phase rollout strategy

In our view, building towards efficient upgrades, as well as the ability to rollback an update are both equally important. Examples of when we have needed to rollback an upgrade are situations where a rollout resulted in changing network or data protocols in incompatible ways. The standard practice used by SQL Azure is a two phase upgrade, with each of the phases being a fully monitored roll-out. In the first phase the new binaries must be capable of understanding both old and new protocols, but should not initiate the new protocol. This allows the new code to communicate with the old version of code and also allows rolling back safely to the old

version. During the second phase, the software can initiate the new protocols because we know the first phase has completed by the time the second phase starts. If there is a rollback, then it only needs to take one step back to the first phase which is still capable of using the new protocols and data formats. The two phase roll-out pattern (Figure 2) is widely applied in SQL Azure in both database and gateway services, and is a key piece of our fault tolerant system design.

Another thing worth mentioning is that our overall approach to system design is to aim for simplicity, because robustness usually follows. For example, rather than building upgrade support on a feature by feature basis, we chose to build a single upgrade scheme and use this as a "hammer" to solve all issues.

To summarize: *perfect your upgrade scheme; it is the goose that lays golden eggs for the service*.

Unplanned failures. These fall into the following buckets: 1) Hardware and network errors, e.g., top-of-rack switch failures, disk failures, rare failures such as bit flips in NIC, memory, etc. – for these we use software-based defense in depth policy, as outlined in Section 2.4; 2) Software bugs – for these we take a *mitigate first, fix next* policy. For example, if a known combination of statements and inputs are generating a type of failure, we temporarily install a statement firewall, effectively banning the problem combination, until a fix can be rolled out; and 3) Human error – an example of this is an error in configuration management.

In general, we have found handling configuration failures to have been extremely painful, because they may impact multiple machines and workloads, and may quickly spread their contagion before we are able to roll back. Longer running test runs that check for any possible regression on large integration and test clusters is one preventative measure. In a similar vein, having to roll back OS patches has also been painful. Byzantine failures are hard to debug. NIC errors have fallen into this category. Software hangs are comparatively easy to debug, because the SQL org is intimately familiar with the database software and tool chain.

Design everything to scale out. The key theme here is to favor decentralized service architecture, and avoid any choke points. The GPM (Figure 1) is the only centralized piece of the SQL Azure architecture. It is a scaled out single point of failure, with seven redundant copies across seven instances. We have designed and modified this piece extremely carefully, running somewhat exhaustive stress tests before deploying modifications. We also enforce machine roles [4], because that ensures that the service follows a loose coupling paradigm, and that there are effectively firewalls across machines belonging to separate roles, which arrest cascading failures. But one does need a balanced approach towards creating roles, because they tend to eat into cluster size, and eventually, into our business margin.

The lessons we have learned during the two years of deployment are to constantly assess what are the access patterns and network pathways. By doing so, we become aware of hotspots in access patterns, and are able to adapt partitioning strategies to mitigate them and release stress across the system. In practice, this turns out to be a rather effective early warning and failure avoidance strategy, e.g., we observed that overloaded partitions lead to throttling that may ripple across significantly large portions of the cluster, and overloaded VIPs (virtual IP addresses) may lead to connectivity issues. It is best to avoid these types of failures by *constant monitoring and adaptation*.

Protect the service with throttling. SQL Azure is not a VM-based solution; therefore the onus is on the service, and not the user, to provide sensible throttling and resource governance, in a manner that tries to balance supply and demand of scale from the users' perspective, while maintaining high availability and reliability guarantees on behalf of the service.

SQL Server offers a varied set of functionality. When combined with customers' varying workloads, the pattern of stress that may be caused across various parts of the architecture, e.g., connectivity, manageability, memory, IO, disk usage, etc., quickly becomes very complex. However, we take a relatively simple view of how to handle this complexity: *no one request should be able to bring the service (or a significant portion of it) down*. This translates into always ensuring that there will be some spare capacity in the system, and that in turn

translates into enforcing thresholds, soft and hard limits, and quotas on almost every shared resource throughout the system.

Resource governance [8] is an intrinsic feature provided by the SQL Server instance. We leverage this to ensure that the system should be able to control the minimum and maximum resource allocations, and throttle activities that are trending towards exceeding quotas. At the same time, we are able to use the resource governance framework to guarantee that demanding users will get performance guarantees, within what the system can sustain.

3 Challenges Ahead

Much of the work that has gone into designing a self-managing and self-healing system architecture (Section 1), are geared to ensure that the service runs smoothly, and mostly without any human intervention, except under conditions of severe failure. Nevertheless, these measures are mostly to maintain the service infrastructure in a highly available and reliable state 24x7. In order to take the service to the next level in terms of providing ever-improving customer value, and also continuing to improve our own business margins, we need to invest along a number of different dimensions, some of which are outlined in this section.

Execute crisp and granular SLAs that offer more cost/consistency/performance options. Florescu and Kossmann [11] argued that in cloud environments, the main metric that needs to be optimized is the cost as measured in dollars. Therefore, the big challenge of data management applications is no longer on how fast a database workload can be executed or whether a particular throughput can be achieved; instead, the challenge is on how the performance requirements of a particular workload may be met transparently to the user, by the service performing internal load balancing of available capacity. Indeed, one of the fundamental points of Cloud PaaS is to have customers no longer have to worry about physical units such as machines, disk drives, etc. At the same time, it is also important for the service provider to play inside of reasonably attractive price:performance sweet spots.

In addition to optimizing on hardware cost, we are also very motivated to offer more options to our customers, and at a finer level of granularity, in order to appeal to ever more numbers of customers to bring their varied workloads to our platform.

One area of investment would be to formulate crisper contracts (SLAs) with customers and partners, which provide a finer granularity of setting expectations. For example, these advanced and granular SLAs would agree on monitoring mechanisms, expected load characteristics, and types of issues that the customer might be willing to debug, with help from the system. We would also negotiate more detailed SLAs with our internal and external providers, and use those to optimize parts of the architecture, e.g., error checking, as well as streamlining Ops, with a view to optimizing price points of running the service.

In a multi-tenant environment, users need to get the resources they pay for, while the system needs to load balance users across servers. Part of the solution may be defining the parameters of a service level agreement in a way that can be supported with high probability.

It would be interesting to investigate algorithms as in [9], which aim to jointly optimize choice of query execution plans and storage layout, to meet a given SLA at minimum cost. Kraska et al. [10] describe a dynamic consistency strategy, called *Consistency Rationing*, to reduce the consistency requirements when possible (i.e., when the penalty cost is low) 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. Algorithms like these easily map on to granular SLAs that offer more fine-tunable options to our customers, with respect to their workload *du-jour*.

Leverage economies of scale; break bottlenecks through strategic hardware investments; optimize database layout vs. cost. An obvious win in the area of leveraging economies of scale, would be to converge on the same hardware SKU as Windows Azure. This should automatically result in more favorable unit prices, and therefore, improve our margins.

An example of moving opportunistically with timely hardware investments to break observed performance and scale bottlenecks is by upgrading to 10 Gb Ethernet connections from rack to rack in a SQL Azure cluster. This helps us break through an IOPS bottleneck we were observing, with intense data traffic across the cluster, e.g., with distributed query workloads. Another possible hardware investment geared towards offering an attractive price:performance tradeoff point for the customer would be to adopt Solid State Disk (SSD) technology. If the price is not right to move every single disk on the cluster to SSD right away, it might still be at the right price point to selectively start the move, e.g., on disks with high traffic, like log disks (multiple virtual databases share a common transaction log in SQL Azure, therefore making log drives a bottleneck).

Generalizing, the first step is to instrument the cluster really well, and monitor it closely with good metrics, to understand where the common bottlenecks lie. Then, as hardware price points evolve, we can latch on to the sweet spots in prioritized fashion, striving to improve the price:performance ratio for the customer, while maintaining comfortable price points of running the service. It would be interesting to investigate query optimizers that adapt storage usage patterns to the relative performance of different volumes, if we were to pursue a hybrid disk strategy in future, e.g., [9]. Adding such adaptivity into the overall architecture would, by itself, generate fresh avenues for cost savings. As in [9], we could formulate the database layout problem as an optimization problem, e.g., by placing smaller on-disk data structures at high QoS levels we can approach a better price:performance point.

React rapidly to maxing out capacity and availability + affordability of new hardware. A successful service must be able to balance the time to deploy a new cluster vs. the declining cost curve of hardware over time. If you buy hardware before there is demand, you waste it. If you buy hardware too late, then your service will not be able to deliver capacity on demand.

A similar set of challenges is brought about when new hardware SKUs become available. To not move fast enough hurts both our customers' value proposition compared to alternatives on the market, as well as hurts our business model. The trick here is to anticipate trends in hardware and price:performance a little bit ahead of the curve, get the certification process in top gear, and roll it out at a point where we are competitive, as well as leverage improving hardware efficiencies to benefit the business.

In order to sustain currently accelerating rates of growth, we need to balance utilization profiles across currently active data centers, as well as gear up to deploy into more data centers with growing demand. To do this efficiently, we need to streamline our processes along multiple dimensions, e.g., acquisitions, hardware and vendor partner chain, gearing internal processes such as testing to move more rapidly on certification, etc.

There is obviously a non-linear cost function involved in deploying a new cluster, just like building a new power-plant is a large CapEx expense for a power company. At times, there may be benefits in incentivizing customers to change their usage pattern so that the service can optimize its own trade-offs in deploying new hardware. This could manifest as a "sale" on an under-utilized data center vs. a full one or perhaps as a rebate for customers that upgrade to a new, resource-conserving feature.

Bake TCO into our engineering system. It has been one of our fundamental principles of system design, that packing more databases per machine reduces our TCO. Moving forward, we need to bake TCO into our software design process, so that it becomes one of the fundamental pillars of system architecture. Here are some examples of how this might be achieved, by asking the following questions: 1) Is the design of the new feature significantly reducing the percentage of total disk capacity that is monetizable, e.g., by reducing the effective cluster-wide capacity to carry non-redundant customer data? 2) Is the design of the new feature significantly reducing the total CPU horsepower of the cluster? 3) Is the design channeling away IOPS capacity in a way that

slows down monetizable customer queries, with respect to negotiated SLA? 4) Are the Resource Management policies and protocols in good alignment with the pricing model?

The cost vs. value of new feature/functionality needs to be considered for each resource dimension. Once considered, the feature may need to be designed differently to minimize large, non-monetizable elements. When there are large costs, giving end users control and visibility over that cost will help maximize the use of shared resources.

System help on performance tuning and Management. As we approach a point where the underlying infrastructure just works, then the main point of failure will shift closer to the customer, i.e., be mostly attributable to a combination of workload, functionality, scale, access patterns, etc. If we can improve the level of self-help that the system can provide to the customer, then in many cases it will become more economical for both the customer, as well as for us as service providers, if the customers are able to undertake recommended remedial actions of their own accord, with existing help and hints from the system.

Pre-cloud era research focused on auto-tuning mechanisms such as automatic index or statistics creation to improve the overall quality of a given plan. These core issues do not change in a simple hosted environment or a platform database service. Various techniques and tools have been created to tune workloads or assist in physical database design, and these same techniques can often be used on a database platform (perhaps with small modifications).

Costs influence performance tuning in cloud environments, automatic or otherwise. For the individual purchasing computing resources, there are incentives now to reduce the recurring cost. It may become very important not just to tune logical and physical database design but to do so against a cost function with multiple price points. For example, properly indexing an application and validating that each OLTP query does seeks instead of scans will be easily measurable in the service bill. Performance evaluation will need to understand the size of the database workspace in addition to traditional core database concepts. If a business decides to increase the computing capacity for a busy season, this will influence performance evaluation and optimization techniques in different ways than on fixed hardware.

From the perspective of the tenant hosting the service, there will be incentives to reduce the cost of delivering the service. This may force re-evaluation of computationally expensive items such as query compilation to both reduce time spent generating plans as well as to find plans that perform well in a multi-tenant hosted environment. This will bias some engineering choices towards reducing internal costs since buying new computing capacity is expensive. Features including physical plan persistence, automatic physical database design tuning, and automatic identification of suboptimal queries now become ways to reduce service delivery costs.

Current platform database systems largely avoid traditional large database-tuning problems by focusing on small query workloads which are easy to tune manually and because database sizes are currently more limited than non-hosted counterparts. As the cloud hosting platforms mature, these differences should largely disappear and the same core challenges will remain.

Performance tuning will also become somewhat more complicated due to multi-tenancy. If hardware is shared by many customers, isolating performance issues may become difficult if they are not isolated from each other. Cloud database platforms will need to invest in mechanisms to isolate problem applications both to troubleshoot them and to manage their service to prevent one database application from negatively impacting others if there is a problem.

4 Conclusions

In this paper we have reflected on the lessons we have learned while ramping up the SQL Azure database service these past two years. The main themes emerging from past lessons we have learned have been to design everything for scale-out, carefully instrument and study the service in deployment and under load, catalog

failures, take a calibrated approach towards responding to them, and to design both the architecture, as well as failure response protocols and processes to contain the scope of failures and mitigate their effects effectively, rather than focusing too heavily on failure avoidance.

With regards to ongoing challenges and investments, the focus is on moving towards a higher level of sophistication in empowering the user of the service to get help from the system to figure their way out of localized failures. We would also like to move towards more granular and load-tunable SLAs. As service providers, it is important for us to move with greater agility to combine hardware trends and algorithmic innovations in areas like optimized database layout and query plans, to approach price:performance sweet spots. In response to accelerating adoption of SQL Azure, it will become important to deploy new data centers rapidly, and to continuously improve TCO, efficiency and utilization, in order to improve the business model of running the service. Finally, we have work to do in terms of improving our tools, both to attract and migrate large and varied customers with mission-critical data and computations to the SQL Azure service.

References

[1] The Cloud: Battle of the Tech Titans, Bloomberg Businessweek, March 3, 2011 (http://www.businessweek.com/magazine/content/11_11/b4219052599182.htm).

[2] Weinman, J., Cloud Economics and the Customer Experience, InformationWeek, March 24, 2011 (http://www.informationweek.com/news/cloud-computing/infrastructure/229400200?pgno=1).

[3] Campbell, D. G., Kakivaya, G., and Ellis, N., Extreme scale with full SQL language support in Microsoft SQL Azure, In Proc. 2010 International Conference on Management of Data, pp. 1021-1024, 2010.

[4] 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, ICDE 2011, pp. 1255-1263.

[5] Sakr, S., Liu, A., Batista, D.M., Alomari, M., A Survey of Large Scale Data Management Approaches in Cloud Environments, IEEE Communications, 2011.

[6] Microsoft Corp.: Microsoft Exchange Hosted Archive. http://www.microsoft.com/online/exchange-hostedservices.mspx

[7] Microsoft Corp.: Microsoft SQL Azure. http://www.microsoft.com/windowsazure/sqlazure/

[8] Microsoft Corp.: Introducing Resource Governor (http://technet.microsoft.com/en-us/library/bb895232.aspx)

[9] Reiss, F.R., Satisfying database service level agreements while minimizing cost through storage QoS, in SCC '05: Proceedings of the 2005 IEEE International Conference on Services Computing.

[10] Kraska, T., Hentschel, M., Alonso, G., Kossmann, D., Consistency Rationing in the cloud: Pay only when it matters, PVLDB, 2(1):253-264, 2009.

[11] Florescu, D., Kossmann, D., Rethinking cost and performance of database systems, SIGMOD Record, 38(1):43-48, 2009.

[12] Thakar, A., Szalay, S., Migrating a large science database to the cloud, HPDC 2010, 430-434.

[13] Microsoft Corp.: Microsoft Case Studies. EHA. http://www.microsoft.com/casestudies/Case_Study_Detail.aspx?casestudyid=4000003098.

[14] Kossmann, D., Kraska, T., Loesing, S., An evaluation of alternative architectures for transaction processing in the cloud, In SIGMOD, 2010.

[15] Microsoft Corp.: Microsoft Case Studies. TicketDirect. http://www.microsoft.com/casestudies/Case_Study_Detail.aspx?CaseStudyID=4000005890.