AzureBench: Benchmarking the Storage Services of the Azure Cloud Platform

Dinesh Agarwal and Sushil K. Prasad
Department of Computer Science
Georgia State University
Atlanta - 30303, USA
Email: {dagarwal2, sprasad}@gsu.edu

Abstract—Cloud computing is becoming mainstream for High Performance Computing (HPC) application development over the last few years. However, even though many vendors have rolled out their commercial cloud infrastructures, the service offerings are usually only best-effort based, without any performance guarantees. Cloud computing effectively saves the eScience developer the hassles of resource provisioning but utilization of these resources will be questionable if it can not meet the performance expectations of deployed applications. Furthermore, in order to make application design choices for a particular cloud offering, an eScience developer needs to understand the performance capabilities of the underlying cloud platform. Among all clouds, the emerging Azure cloud from Microsoft remains a challenge for HPC program development both due to lack of its support for traditional parallel programming support such as MPI and map-reduce and due to its evolving APIs. To aid the HPC developers, we present an open-source benchmark suite, AzureBench, for Windows Azure cloud platform. We report comprehensive performance analysis of Azure cloud platform’s storage services which are its primary artifacts for inter-processor coordination and communication. We also report on how much scalability Azure platform affords using up to 100 processors and point out various bottlenecks in parallel access of storage services. The paper also has pointers to overcome the steep learning curve for HPC application development over Azure. We also provide an open-source generic application framework that can be a starting point for application development for bag-of-task applications over Azure.

I. INTRODUCTION

Cloud computing promises scientists with a new infrastructure and paradigm for large scale distributed computing [1]. The emerging cloud platforms, such as Microsoft’s Azure - with their potential for large scale computing and storage capabilities, easy accessibility by common users and scientists, on demand availability, easy maintenance, sustainability, and portability - have the promise to be the platform of choice for HPC applications. Cloud computing provides users with quick access to a large-scale HPC cluster without having to worry about the setup, maintenance, or initial investment. The utility based framework facilitates experimenting with large amount of compute power obviating the need to own a parallel or distributed system [2]. Scientific applications with varying load cycles are the perfect fit for pay-as-you-go cost model of cloud computing as the resources in cloud platform can be allocated and deallocated on-demand based on the application’s requirements.

There are several vendors offering cloud services in the market today. The service offerings differ in terms of software support, platform support, and also developmental tools support. While some vendors empower the developer to deploy his/her own virtual machine images, some others provide their own APIs to interact with their cloud services. Moreover, some vendors support traditional programming support such as MPI and Map-reduce while others have different software infrastructures. Therefore, it is critical for an eScience developer to choose the cloud platform that is most suitable for his/her application based on the availability of resources required for development and maintenance of that application.

There have been some initial work to understand the pros and cons of scientific cloud computing in general [1]–[4] and over Windows Azure cloud platform [5]–[8] specifically. Windows Azure cloud platform lacks the traditional HPC software environment including MPI and Map-reduce. On the other hand, while it might not be the best fit for some scientific applications that essentially require MPI-like functionality, it can provide a very simple model for a wide variety of scientific applications.

Azure platform’s middleware provides robust fault tolerance through its Queue storage mechanism, query-like interface to store data through Table storage, and persistent yet fast random access to hierarchically stored data through Blob storage (similar to files). The importance of Azure platform has been recognized by industry as well as academia as is evident from the rare partnership of National Science Foundation (NSF) and Microsoft in funding scientific research on Azure cloud.

In this paper we present AzureBench - a suite of benchmarks for Azure platform’s storage services. We provide a comprehensive scalability assessment of Azure platform’s storage services using up to 100 processors. Our work extends a preliminary study by Hill et al. [5] in 2010 and provides additional key insights. Our assessments of Azure platform provide updated, realistic performance measurements as we utilize the APIs released after significant changes were made to the Azure cloud platform since 2010. Some of the earlier restrictions of Azure platform’s storage services, such as expiration of a message in Queue storage after 2 hours,
rendered Azure platform problematic for long-running real-world scientific applications.

AzureBench is an open-source benchmark suite hosted at Codeplex repository available under GPLv2. The open-source nature will motivate further research in this direction. We have chosen not to include the assessment of operating cost and SQL-Azure functionalities in this study. Moreover, comparison with other cloud platforms is also not studied in this paper - primarily due to the differences in architectures. We plan to address both these issues by including it in AzureBench and provide a detailed report in near future.

The Windows Azure cloud platform provides three types of storage services: Blob storage, Queue storage, and Table storage. We have done an extensive study of all three storage mechanisms for varying load and compute instances.

In order to evaluate Windows Azure storage mechanisms, we deploy varying number of virtual machines (VM) and these virtual machines read/write from/to Azure storage concurrently. For the sake of fair comparison, we maintain the same amount of input and output load throughout the benchmarking process.

Azure platform performs impressively better than what has been reported earlier [5], as many of the previous drawbacks have been addressed by Microsoft. Nevertheless, there are still few bottlenecks left in Azure storage mechanisms that we have discovered. We also lay out a simplistic generic application framework for Azure cloud to give developers a starting point to design their own applications. Finally, we provide a summary of our findings and make recommendations for developers to efficiently leverage the maximum throughput of storage services.

We do not analyze the local storage feature at each compute resource as it is similar to writing to the local hard disk and thus does not add value to the study. Since we are only interested in assessment of Azure storage in this study, the compute instances do not perform any computationally-intensive task on the data. The data generation is limited to minimum possible and the time spent in data generation is also ignored. We have done another study to analyze the performance of Azure platform for both computational and I/O phases with respect to a scientific application from Geographic Information System and Science (GIS) domain. Motivated readers can refer to [9] to read more details about that study.

The rest of the paper is organized as follows: Section II discusses the previous work reported in literature and gives the background on Windows Azure platform. We present a generic framework for HPC applications on Windows Azure cloud platform in Section III. Our experimental results and analysis is presented is Section IV. Section ?? concludes this paper with comments on future work.

3Source code for AzureBench can be downloaded from http://azurebench.codeplex.com/.

Fig. 1: Google search trends for cloud computing (blue line), Grid computing (yellow line), and Virtualization technology (red line) [10].

II. BACKGROUND AND LITERATURE

A. Cloud Computing for Scientific Applications

Cloud Computing is typically perceived as a set of shared and scalable commodity computing resources, located all over the world and available on-demand over a network. The resources are made available to end users as various services such as “Platform as a service” (PaaS), “Infrastructure as a service” (IaaS), and “Software as a service” (SaaS). Conceptually, the key idea here is to abstract the provisioning mechanism at a level, where users can avail of these resources dynamically without burdening themselves with either the availability or the maintenance. There is yet no universally accepted definition of Cloud computing [2].

The term cloud computing started getting popular around the year 2007, its popularity has only increased since then. Figure 1 illustrates the web search interest growth rate of cloud computing (blue line) as compared to virtualization technology (red line) or grid computing (yellow line), according to Google search Insights [10]. Cloud computing outperformed grid computing in terms of growth rate of web search interest in mid 2008. Over the next few months, cloud computing also encapsulated virtualization and hence a decrease in the growth rate of web search interest for virtualization was seen.

Thanks to the dynamic provisioning of resources, cloud computing has drawn wide interest from researchers, especially those working with data and compute-intensive scientific applications [11]–[13]. Users are billed for utilization, largely based on the time a resource was reserved by a user.

Srirama et al. [14] report how universities can utilize the existing HPC resources as their own private clouds. Ekanayake et al. [15] have created a framework for iterative map-reduce on Azure and have demonstrated, by way of samples, how their framework can be utilized to port map-reduce based applications to Azure.

Commercial vendors, similarly, have also recognized the importance of cloud computing; many of the vendors have already rolled out their cloud computing based services. Rimal et al. [16] have done a comprehensive comparison of various cloud computing vendors including Amazon EC2, Microsoft Azure, Google App Engine, IBM Blue Cloud, Nimbus, 3Tera,
and Gigaspaces among others. Gaming industry has also used cloud platform to host compute intensive aspects of games to enable a rich gaming experience without the need of expensive computing resources [17].

A number of HPC scientific applications have also been ported to cloud. Rehr et al. [13] studied the feasibility of porting two scientific applications - X-ray spectroscopy and electronic structure code FEFF - on Amazon EC2. Hoffa et al. [18] have reported the advantages and shortcomings of using cloud computing for an application called Montage from the field of astronomy and compared the performance of Montage over local and virtual cluster.

B. Windows Azure Platform

Windows Azure platform is a computing and service platform hosted in Microsoft data centers. Its programming primitives consist of two types of processes called web role and worker role for computation, a variety of storage mechanisms, and the Windows Azure Fabric. The typical programming artifacts of Windows Azure platform are shown in Figure 2. The web role acts as a web application accessible over HTTP and HTTPs endpoints and usually is the front end of any Azure cloud based application. Worker roles are the processing entities representing the backend processing for the web application. A web role is hosted in an environment with support for a subset of ASP.NET and Windows Communication Foundation (WCF) technologies [19]. Both web role and worker role processes can have different configurations as shown in Table I.

The storage objects are organized as services and can be accessed by both web roles and worker roles. There are three primary types of storage services: Queues, Blobs, and Tables. Additionally, local storage can be configured for role instances. Azure platform also provides a caching service to temporarily hold data in memory across different servers. In this paper, we only concentrate on the three primary storage services.

Queues are similar to the traditional queue data structure, but first-in first-out (FIFO) functionality is not always guaranteed. Queues are prominently used for communication among instances of web roles and worker roles. A reference to a task is usually put as a message on the queue and a set of worker role instances are deployed to process them. The blob storage is a persistent storage service like a traditional file; the data can be stored as a collection of small blocks of size up to 4 MB or large pages of size up to 1 TB. Azure tables arrange data into a structured organization and thus are useful for query-based data management.

![Table I: Listing of virtual machine configurations available for web role and worker role instances with Windows Azure](image)

III. AN APPLICATION FRAMEWORK FOR SCIENTIFIC APPLICATIONS ON WINDOWS AZURE CLOUD PLATFORM

![Figure 3: A Generic Application Framework for Scientific Applications on Windows Azure Cloud Platform](image)

Figure 3 shows a generic framework for application development on Windows Azure cloud platform. Application workflow for Azure cloud based applications typically starts with a web-interface where users have an option to specify the parameters for background processing. Moreover, this interface should be interactive to update users with current state of the system, especially for time consuming applications. This is typically achieved by employing a web-role, although
some applications use command line interface where this component could be missing from the application framework.

VM configuration for web role depends on the intensity of the tasks to be handled by the web role. For applications where web role performs computationally-intensive operations, a fat VM configuration should be chosen. Similarly, if the web role needs to access large data items from cloud storage, it could be a fat VM to upload/download data to/from the storage using multiple threads.

To communicate task with worker roles, web role puts a message on a Task assignment queue as shown in Figure 3. If there are distinct input parameter sets, there could be multiple task assignment queues for each set of parameters.

Worker role instances keep checking this queue and as soon as they locate a message there, they start background processing based on the content of the message in the queue. Worker roles communicate with the storage services to acquire the data required for processing.

One or more queues can be utilized to communicate among worker role instances. Since one role instance cannot automatically query the state of other role instances in Windows Azure, the communication depends on storage services - typically Queue storage. Azure platform also supports TCP endpoints that can be configured to facilitate an application to listen on an assigned TCP port for incoming requests. TCP messages can be sent/received among Azure roles or can be used for communication with external services - these messages are not currently studied in this paper.

For a time-consuming interactive application, it is essential to update the user interface. To achieve this, a worker role instance can put a message on a queue after every phase of processing completes. The web role can read the number of messages in this queue and accordingly update the user interface. This queue is shown as Termination Indicator Queue in Figure 3.

The effectiveness of this framework has been proven in several applications, such as our own GIS application Crayons [9] and Twister4Azure [15].

IV. BENCHMARK EXPERIMENTS AND TIMING CHARACTERISTICS

Windows Azure storage services partition the stored data across several servers to provide enhanced scalability. The absolute limit on a storage account is 100 TB. However, there are additional limits on scalability targets. Windows Azure storage services can handle up to 5,000 transactions (entities/messages/blobs) per second. Moreover, there is a maximum bandwidth support for up to 3 GB per second for a single storage account. Exceeding any of the specified limits result in the failure of a role instance.

In this section, we detail our analysis with the performance test of all three Azure storage services: Blob storage, Table storage, and Queue storage.

A. Blob Storage

Blob storage in Windows Azure is similar to the traditional file system. Blob storage service, organized into a hierarchy, can be used to store large amount of unstructured data. One storage account can have multiple blob containers, and one container can store multiple blobs. Blobs are partitioned based on “container name + blob name” combination, i.e., each individual blob can be stored at a different server for maximum throughput. The throughput of a blob is up to 60 MB per second.

Algorithm 1 Azurebench blob benchmarks

```plaintext
syncCount := 0
for repeat := 1 → 10 do
    BlockBlob := "AzureBenchBlockBlob"
    PageBlob := "AzureBenchPageBlob"
    Total blocks/pages in a blob count := \( \frac{100\text{MB}}{1\text{MB}} \)
    /* Page blob upload */
    content := randomData(1MB)
    for pageid := 1 → \( \frac{\text{count}}{\text{worker}} \) do
        PutPage(PageBlob, content)
    end for
    /* Block Blob Upload */
    content := randomData(1MB)
    for blockid := 1 → \( \frac{\text{count}}{\text{worker}} \) do
        PutBlock(BlockBlob, content)
    end for
    PutBlockList(blockIdList)
    Synchronize(+ + syncCount)
end for
/* Downloading pages from a Page blob randomly */
for pageID := 1 → count do
    pageOffset := randomNumber(1, count)
    Page := GetPage(PageBlob, pageOffset)
end for
/* Downloading blocks from a Block blob */
for blockID := 1 → count do
    Block := GetBlock(BlockBlob, blockID)
end for
Synchronize(+ + syncCount)
/* Download entire Page Blob */
Download PageBlob using PageBlob.openRead()
/* Download entire Block Blob */
Download blob using BlockBlob.DownloadText()
Synchronize(+ + syncCount)
DeletePageBlob(PageBlob)
DeleteBlockBlob(BlockBlob)
end for
```

There are two types of blobs in Windows Azure: Block blobs and Page blobs. Block blobs can be created in two ways - Block blobs less than 64 MB in size can be directly uploaded to blob storage as a single entity, and Block blobs greater than 64 MB can be uploaded as a set of multiple blocks of size up to 4 MB each. There can be a total of 50,000 such blocks in a blob. Thus, the maximum size of a Block blob cannot exceed 200 GB.

The Page blob artifact was not there in the Blob storage initially; it was later introduced to facilitate random read/write
operations on blobs. A Page blob is created and initialized with a maximum size; pages can be added at any location in the blob by specifying the offset. The offset boundary should be divisible by 512, and the total data that can be updated in one operation is 4 MB. A Page blob can store up to 1 TB of data.

Algorithm 1 shows the skeleton of our benchmark code for Azure Blob storage (Azure APIs highlighted in bold italics). Each worker role starts with uploading one 100 MB blob to cloud storage in 100 chunks (blocks or pages) of 1 MB each.

To ensure that the process of downloading starts only after the process of uploading has finished, worker roles need to synchronize. Synchronizing among worker roles in Azure platform is an interesting process by itself. There is no API in the Azure software development kit that provides a traditional barrier like functionality. However, a queue can be used as a shared memory resource to implement explicit synchronization among multiple worker role instances. Each worker can put a message on a designated queue that acts as a barrier. When the number of messages in the queue is equal to the number of workers, it is safe to assume that all workers have touched the barrier and hence all of them can cross it.

However, what makes it interesting is that if the workers delete the messages after exiting the **While** loop, those workers that have put the message in the queue, but yet to exit the loop, will never meet the loop termination condition. On the other hand, if the workers do not delete the messages, the number of workers will never match the number of messages in the queue after first synchronization cycle. Therefore, in our case, for each synchronization phase, we pass a variable that accounts for the messages left in the queue during previous synchronization phases. Moreover, since a large number of requests to get the message count can throttle the queue, each worker sleeps for a second before issuing the next request. The time reported in the experiments does not include the time spent in synchronization. Our synchronization mechanism is illustrated in Algorithm ??.

**Algorithm 2** Synchronization among worker role instances

<table>
<thead>
<tr>
<th>Input: syncCount</th>
</tr>
</thead>
<tbody>
<tr>
<td>syncQueue := “Termination_Indicator_Queue”</td>
</tr>
<tr>
<td>arrived := 0</td>
</tr>
<tr>
<td>while arrived &lt; (workers * syncCount) do</td>
</tr>
<tr>
<td>arrived := GetMsgCount(syncQueue)</td>
</tr>
<tr>
<td>Sleep(1 second)</td>
</tr>
<tr>
<td>end while</td>
</tr>
</tbody>
</table>

Figure ?? shows the performance analysis of Azure platform’s Blob storage service. The total uploaded data to the Blob storage is 2 GB - 1 GB for each Block and Page blob. The downloaded data, however, is 2 GB per worker role instance. Since each worker downloads the blobs from the Blob storage, the download time increases with increasing number of worker role instances for both Block and Page blobs as shown in Figure ??.

The upload time reduces with the increasing number of workers, as the amount of the data to be uploaded per worker reduces. Moreover, the increasing throughput for the process of uploading suggests that the Blob storage scales well even when multiple clients are trying to upload data.

The maximum throughput for blob download process was 165 MB/s, achieved for Block blob download using 96 workers, and the maximum throughput for blob upload process was 60 MB/s, realized for Page upload process using 96 workers. The maximum throughput for a Block blob upload process was only a little over 21 MB/s using 96 workers; the reason why Page blob upload process demonstrates superior upload throughput is the capability of Page blobs to allow fast random access to read/write pages.

To evaluate the performance of random access for Page blob download process, each worker downloads 100 random pages from the Page blob that was uploaded previously. Since Block blobs do not support random access of blocks, we let each worker read one block at a time sequentially.

Figure ?? shows the download time and throughput of
Blob storage when blobs are downloaded by accessing one block/page at a time. The maximum throughput achieved by Page wise blob downloading was more than 71 MB/s using 96 workers. The Block wise blob downloading for the same amount of worker roles was more than 104 MB/s.

B. Queue Storage

In Windows Azure platform based applications, queues are used by both web role and worker roles to communicate with each other or among different instances of the same role. The distinguishing feature of an Azure Queue storage from the traditional queue data structure is its lack of ability to guarantee a FIFO operation. This lack of guarantee for FIFO operation can cause issues if a queue is to be used to signal a special event. For instance, if web role wants to put a message at the end of the task queue to signal the end of work, it might not work as expected. Since FIFO is not guaranteed, the worker roles might read this message before the actual messages for tasks and hence quit processing while there is work in the task pool. To achieve termination signaling, it is recommended to create a dedicated termination indicator queue where worker instances can send messages to signal an event.

A storage account can have unlimited number of uniquely named queues. Each queue can have unlimited number of messages (limited by 100 TB limit of a storage account) and each message has a visibility timeout period. Queues are partitioned based on queue names, i.e. a single queue and all the messages stored in it are stored at a single server. A single queue can only handle up to 500 messages per second. Thus, if an application only interacts with queues, it is essential to employ multiple queues for better scalability.

The messages in a queue can be consumed by any service, but the consumer is expected to delete the message after processing. Once a message is read, it is made invisible from the queue for other consumers. A consumer also has the option to peek a message rather than reading it, in which case the message stays visible in the queue for other consumers. If the consumer does not delete the message after its consumption, it reappears in the queue after a certain time.

Algorithm 3 Azurebench Queue storage benchmarks with a separate queue per worker

```plaintext
QueueName := "AzureBenchQueue + roleID"
Message_Size := 4KB
Message_Count := \lceil \frac{20000}{\text{workers}} \rceil
CreateQueue(QueueName)
for repeat := 0 \rightarrow (\log(\frac{64}{4K}) := 4) do
  for count := 1 \rightarrow Message_Count do
    Message := randomData(Message_Size)
    PutMessage(QueueName, Message)
  end for
  for count := 1 \rightarrow Message_Count do
    Message := PeekMessage(QueueName)
  end for
  for count := 1 \rightarrow Message_Count do
    Message := GetMessage(QueueName)
    DeleteMessage(Message)
  end for
  Message_Size := Message_Size * 2
end for
DeleteQueue(QueueName)
```

Similarly, if a message is left in the queue for longer than a week (the duration was 2 hours for previous APIs), it automatically disappears. The maximum size of a message supported by Azure cloud is 64 KB - it used to be 8 KB prior to October 2011 version of Azure APIs. To store larger data, one can store the actual data in Blob storage and put the blob’s name in the queue as a message. Equipped with these properties, queues can easily facilitate the behavior of a shared task pool with in-built fault tolerance mechanisms.

For our experiments, we test three operations on Azure queues: inserting a message using PutMessage API, reading a message using GetMessage API, and reading a message using PeekMessage API. Concurrent consumers can read a
message from a queue using `PeekMessage` API. However, if a consumer reads a message using `GetMessage` API, only first consumer can read the message as the message becomes invisible from the queue for a certain amount of time, defined at the time of message creation.

We have evaluated Queue storage under two scenarios, (i) each worker works with its own dedicated queue, and (ii) all workers access the same queue. For both experiments, a total number of 20K messages were first inserted in the queue, then read using both APIs, and finally deleted from the queue.

Algorithm ?? represents the first scenario - each worker has its own queue. In this experiment, we evaluate the performance of the queue storage with varying size of messages - 4 KB, 8KB, 16 KB, 32 KB, and 64 KB. Interestingly, 48 KB (49152 Bytes to be precise) is the maximum usable size of an Azure queue message, rest of the message content is metadata. The total real data uploaded (and downloaded) to (from) Queue storage in an experiment with message size of 48 KB is around 1.2 GB.

Figure ?? shows how Windows Azure platform’s Queue storage scales with varying load and varying number of worker role instances. Impressively, the Queue storage scales very well for varying message sizes (4 KB through 64 KB) and varying number of worker role instances for all three operations - insertion, reading using `PeekMessage`, and reading using `GetMessage`. We also tried the same experiment with 200 messages in total and the results were still the same. The experiments at different times also demonstrated similar behavior.

Windows Azure platform maintains three replicas of each storage object with strong consistency [20]. Figure ?? shows the time to put a message on the queue. For `Put Message` operation, the queue needs to be synchronized among replicated copies across different servers. Figure ?? shows the behavior of `Peek message` operation. It is the fastest of all three operations, as there is no synchronization needed on the server end. The `Get Message` operation, as shown in Figure ??, is the most expensive operation as in this case, in addition to synchronization, the message also becomes invisible from the queue for all other worker role instances, and hence extra state needs to be maintained across all copies. Moreover, in our case the `Get Message` operation also includes deletion of the respective message.

One interesting case is of message size 16 KB. Surprisingly, the `Get` operation for this sized messages took significantly more time than other message sizes (both smaller and larger ones). We do not know the reason behind this, but this was consistently seen in all repeated experiments.

Algorithm ?? illustrates the steps of evaluation where multiple workers concurrently interact with a single queue. Each worker accesses the queue once and then spends a certain amount of time before going back to the queue again. This behavior simulates a real world application, where the application accesses the queue intermittently during the course of execution. We varied the time taken by a worker before going back to the queue from 1 second to 5 seconds; the reported time only includes the time spent in communication with the queue. We ensured that the total number of transactions remain same irrespective of number of workers. Thus, workers proportionately carried out fewer transactions on the shared queue as their number increased. The message size for this experiment was kept constant at 32 KB. Additionally, in order to ensure that the number of transactions between the workers and the queue never exceed the bandwidth limit of 500 messages per second, each operation is split into multiple
Algorithm 4 Azurebench queue benchmarks with a single queue shared among multiple workers

\[ \text{QueueName} := \text{"AzureBenchQueue"} \]
\[ \text{Message\_Size} := 32\text{KB} \]
\[ \text{Message\_Count} := (500) \]
\[ \text{rounds} := \frac{20,000}{500} \]
\[ \text{thinkTime} := 1\text{ second} \]
\[ \text{for repeat} := 1 \rightarrow ((\frac{5\text{ seconds}}{1\text{ second}}) := 5) \text{ do} \]
\[ \text{for round} := 1 \rightarrow \text{rounds} \text{ do} \]
\[ \text{for count} := 1 \rightarrow \text{Message\_Count} \text{ do} \]
\[ \text{Message} := \text{randomData}(\text{Message\_Size}) \]
\[ \text{PutMessage}(\text{QueueName}, \text{Message}) \]
\[ \text{end for} \]
\[ \text{think}(\text{thinkTime}) \]
\[ \text{for count} := 1 \rightarrow \text{Message\_Count} \text{ do} \]
\[ \text{Message} := \text{PeekMessage}(\text{QueueName}) \]
\[ \text{end for} \]
\[ \text{think}(\text{thinkTime}) \]
\[ \text{for count} := 1 \rightarrow \text{Message\_Count} \text{ do} \]
\[ \text{Message} := \text{GetMessage}(\text{QueueName}) \]
\[ \text{DeleteMessage}(\text{Message}) \]
\[ \text{end for} \]
\[ \text{think}(\text{thinkTime}) \]
\[ \text{thinkTime} := \text{thinkTime} + 1\text{ second} \]
\[ \text{end for} \]

Figure 7 shows the behavior of Queue storage when multiple workers are accessing a queue in parallel. Parallel access of a queue increases the contention at the queue, hence the time taken by each operation is greater than the time taken when each worker accesses its own queue (Figure ??). The think time also plays a vital role in realizing the performance of a queue. It can be seen from Figure ??, that the time taken by an operation reduces as the think time increases; in some cases, the time reduces by a factor of almost two. As the number of workers starts increasing, the time starts decreasing. This is not due to any reduction in overall parallel access to the shared queue. This demonstrates that the queue implementation scales very well at these access frequencies.

The first scenario, where we use separate queues for each worker role instance, each worker can put a parallel request as the queues are partitioned based on queue names. This is the reason why we see super-linear speedup in many cases. Consequently, we recommend usage of multiple queues as and when possible to make efficient use of Queue storage.

C. Table Storage

Table storage provides semi-structured data storage in Azure cloud platform. A table is comprised of entities of up to 1 MB in size; each entity is composed of up to 255 properties. A table can be queried based on the default properties - Row Key and Partition Key - which apparently also form the unique key
for an entity. Unlike traditional database tables, Azure Table storage does not have a schema. All of the properties of a table are stored as \((\text{Name}, \text{Value})\) pairs, i.e. two entities in the same table can have different properties.

Tables are partitioned on the partition keys, i.e. entities of a table that belong to the same partition are stored together on a server. A single partition can support access to a maximum of 500 entities per second. Therefore, a good partitioning of a table can significantly boost the performance of Table storage.

Algorithm ?? shows the structure of our benchmark tests for Table storage. Each worker role instance inserts 500 entities in the table, all of which are stored in a separate partition in the same table. Once the insertion completes, the worker role queries the same entities 500 times. After the querying phase ends, the worker role updates all of the 500 entities with newer data. Finally, all of these entities are deleted. The exact experiment is repeated 5 times with varying entity sizes. We have experimented with entity sizes of 4 KB, 8 KB, 16 KB, 32 KB, and 64 KB.

Figure ?? shows the performance of Table storage service. The timings are almost constant till 4 concurrent clients for all entity sizes across all four operations. It can be seen from Figure ?? that updating a table is the most time consuming process. We only tested the unconditional updates by using the wild card character * for \(\text{ETag}\) during update queries. The least expensive process is querying a table, as shown in Figure ??.

For entity sizes 32 KB and 64 KB, the time taken for all of the four operations increases drastically with increasing number of worker role instances.

We initially started with inserting about 1000 entities each and experienced a small number of \textit{server busy} exceptions during the experiments, which is an indication of hitting the 500 transactions per second limit. Therefore, we tried with only 500 transactions and everything worked without any exception.

Similar to the previous case of Blob and Queue storage, when we run into such exceptions, the worker sleeps for a second before retrying the same operation.

Many scientific applications require a storage service that offers query like behavior, as well as well organized data storage than a simple file system. Azure Table storage provides this functionality with an impressive throughput. However, a table can only handle entities (rows) that are at most 1 MB in size and have up to 255 properties (columns); we only had one column per row for our experiments.

The name Table storage could confuse a beginner developer to expect a SQL like functionality, where the size of the table should be controlled by the limit of the storage account size - both in terms of number of entities as well as properties. Moreover, if the fundamental storage unit of an application can expand beyond 1 MB, than table is not the best choice for storage - a blob should be considered. A blob can store up to 64 MB of data, or even more by arranging data into multiple blocks or pages.

Figure ?? shows the per operation time for Queue and Table storage services. The reported time is the average time taken

![Fig. 8: Table Storage](image)
by an operation, i.e. the division of total time taken by all the worker roles to finish that operation, and the number of workers. It is evident from Figure ?? that the Queue storage scales better than the Table storage as the number of workers increases.

V. CONCLUSION AND FUTURE WORK

In this paper we have presented AzureBench - an open source benchmark suite for Windows Azure platform’s storage services - along with experimental details to analyze the performance capabilities of Azure cloud platform. We have shown a comprehensive performance evaluation of Windows Azure platform’s storage services - Table, Blob, and Queues. We also present a generic framework along with pointers for HPC application development on Azure.

As a future work, we are going to explore additional services provided by Windows Azure platform, such as local drives, caches, and SQL Azure database. We will also include resource provisioning times and application deployment timings. Finally, we will incorporate benchmarking suited for other cloud offerings by different vendors.

REFERENCES


Fig. 9: Per operation time for Table (insert, query, update, and delete) and Queue storage (put, peek, and get) services