Scaling a Microsoft Azure Cloud Service

Implementing a Horizontally Scalable Scientific Computing Service for BLAST

This paper demonstrates how to use Microsoft Azure to implement a horizontally scalable scientific computing service. It uses Basic Local Alignment Search Tool (BLAST) search queries as an example. BLAST is an application that searches biological databases to find regions of similarity between nucleotide or protein sequences. A BLAST search enables a researcher to compare a query sequence with a library or database of sequences, and identify library sequences that resemble the query sequence above a certain threshold.

The BLAST sample application (http://blast2.cloudapp.net) is a scalable search service with a web front end. It uses a Microsoft Azure Service Bus Queue and Microsoft Azure Table service within a scalable Microsoft Azure Cloud Service that hosts a front-end web portal and back-end BLAST worker instances. With a Microsoft Azure Cloud Service, each worker instance runs in parallel with other instances. The processing power of the application can be increased simply by asking Microsoft Azure for additional instances. This kind of architecture is sometimes called the scale-out pattern. It is ideal for “embarrassingly parallel” applications where there are no data dependencies between loop iterations. Running multiple BLAST queries at the same time is an example of this situation.

Overview

The following diagram is an overview of how the BLAST sample application works:

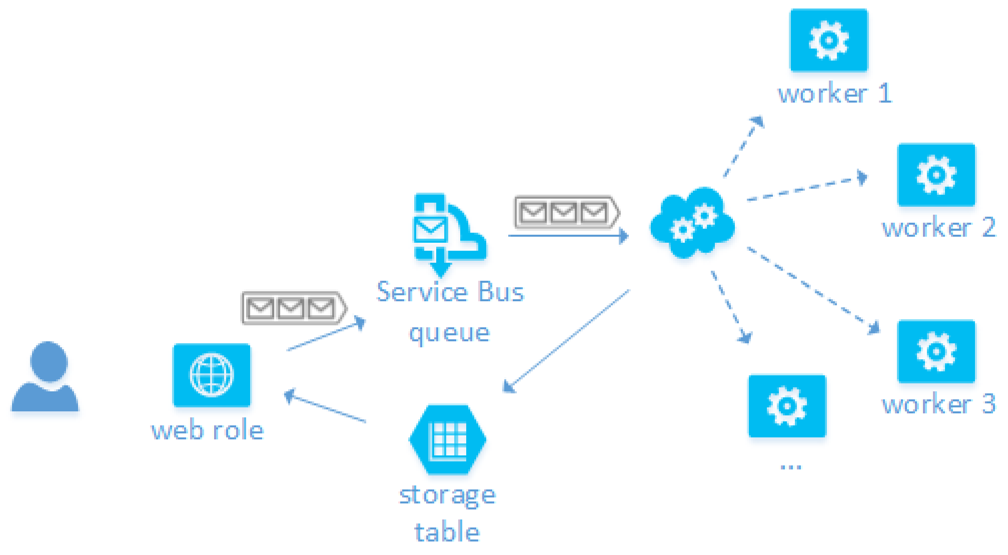


Figure 1: Diagram of BLAST sample application in action

The researcher enters genomic queries in a web application that is hosted in a Microsoft Azure web role instance. A web role instance is a virtual machine (VM) that runs in Microsoft Azure. However, unlike a standard VM that comes with many administrative responsibilities, Microsoft Azure manages the web role instance, performing tasks such as installing operating system patches and automatically rolling out new patched images. Microsoft Azure also monitors the web role instance, restarting it if it fails.

Scaling a Microsoft Azure Cloud Service

The BLAST web application adds each new genomic query request from the user to a queue in the Microsoft Azure Service Bus. Microsoft Azure Service Bus is a service that lets you reliably pass messages between distributed components that run on the premises or in the cloud. One mechanism for communication is the Service Bus queue, which is used by the BLAST example. A Service Bus queue offers first in, first out guaranteed message delivery and supports a range of standard protocols and APIs to put messages in the queue and remove messages from the queue. (For more information, see the Service Bus documentation.)

The back end of the BLAST search application consists of one or more Microsoft Azure worker role instances. Worker role instances are VMs that run in Microsoft Azure. Worker role instances do not ordinarily serve web pages; instead, they perform computations. Like web role instances, worker role instances are managed by Microsoft Azure. Worker role instances run Windows Server as the operating system. You can configure the system to include as many worker role instances as needed.

In the BLAST sample, idle worker role instances pull query requests from the Service Bus queue for processing. When the query is finished, the instances record the results into Microsoft Azure Table service. Table service is a fault-tolerant, ISO 27001 certified NoSQL key/value store. Table service can be useful for applications that must store large amounts of non-relational data.

The BLAST web application reads the results of the completed queries from Table service and presents them to the user.

Configuring and Publishing the BLAST Application

Microsoft Azure provides convenient options for creating and managing the services it provides. You can perform these tasks with:

- The Microsoft Azure Management Portal
- A command-line interface
- Calls to the Microsoft Azure API from within a program
- The built-in automation features of Microsoft Visual Studio.

To configure and publish the BLAST sample application, it's easiest to perform some steps from the Microsoft Azure management portal and then let Microsoft Visual Studio publish the application for you to a Microsoft Azure Cloud Service.

In the Microsoft Azure management portal, you will:

- Create a Microsoft Azure Service Bus namespace that provides a unique name prefix for services such as queues
- Create a Service Bus queue
- Create a Microsoft Azure Cloud Service
- Create a Microsoft Azure storage account

These activities create credentials that will allow your application to identify unique Microsoft Azure service endpoints and gain access to them.

Next, you can use Visual Studio to provision web role and worker role instances, and publish the application code to these instances.

Scaling a Microsoft Azure Cloud Service

You can find detailed information on how to configure and publish the BLAST example application in the companion paper, *Installing the Microsoft Azure BLAST Example*.

The rest of this paper gives an overview of how Microsoft Azure services work together to perform BLAST queries once the application is running. The sections discuss:

- Adding jobs to the Service Bus queue
- Monitoring the queue for new requests
- Storing the results of the BLAST search in Microsoft Azure Table service
- Initializing the worker role with data from Microsoft Azure Blob service

Note: The BLAST sample uses C# and targets the Microsoft .NET Framework, but you can also use other platforms with Microsoft Azure. For an example, see the paper, *Building Scalable Services in Microsoft Azure with Python*, to learn how to perform the back-end BLAST computation by using the Python language.

Adding Jobs to the Service Bus Queue

The BLAST sample's web front end defines a class named **SearchTaskPUProxy** that acts as an intermediary between the web application and the Microsoft Azure Service Bus Queue. Here is the relevant section of the code.

```
public class SearchTaskPUProxy
{
  private QueueClient mTaskQueue;
  private const string QueueName = "JobQueue";

  // ...
}
```

The **SearchTaskPUProxy** class has an instance variable **mTaskQueue** that contains a Microsoft Azure **QueueClient** object. The **QueueClient** class is provided by the Microsoft Azure API. The value of the instance variable represents a connection to the particular Service Bus queue that contains the application's pending BLAST query requests. The constructor of the class uses Microsoft Azure APIs to initialize the **QueueClient** object. Here is the code.

```
public SearchTaskPUProxy()
{
  string connectionString =
       CloudConfigurationManager.GetSetting("Microsoft.ServiceBus.ConnectionString");

  // ...

  mTaskQueue = QueueClient.CreateFromConnectionString(connectionString, QueueName);

  // ...
}
```

The following code example shows how the web application queries the **CloudConfigurationManager** class's **GetSetting** static method for the connection string of the Microsoft Azure Service Bus. The connection string includes access credentials and the service's namespace identifier. The code uses the connection string to create a **QueueClient** instance that is connected to the queue named **JobQueue**.

When the user asks for a new BLAST search, the web application invokes the **QueueClient** object's **Send** method. This is shown in the following code example.

```
public void QueueJob(Entities.SearchTask task)
{
  if (string.IsNullOrEmpty(task.Id))
  {
    task.Id = Guid.NewGuid().ToString("N");
  }
  task.LastTimestamp = DateTime.UtcNow.Ticks;
  task.State = "QUEUED";
  task.LastMessage = "Queued";
  mTaskQueue.Send(new BrokeredMessage(task));
  mProcessingUnit.Create(task);
}
```

The argument to the application's **QueueJob** method is an instance of the **SearchTask** class. The BLAST sample defines this class in order to format the information contained in each queued search request. Here is the definition of the **SearchTask** class:

```
public class SearchTask
{
   public string Id { get; set; }
   public string Name { get; set; }
   public string InputFile { get; set; }
   public long LastTimestamp { get; set; }
}
```

To send a request for running a BLAST job through the queue, the application first creates a **SearchTask** object, then passes it to the queue. When a worker instance eventually retrieves the search task for processing, it deserializes the item and creates its own instance of the **SearchTask** class. The **SearchTask** class is defined in a library that is loaded by both the web front-end and the worker back-end processes. Using a class is optional; you could also use simple strings, or use some of the properties available in the Microsoft Azure **BrokeredMessage** class.

Monitoring the Queue for New Requests

After the query request is added to the queue by the web front end, it remains in the queue until one of the worker role instances retrieves it for processing. The Service Bus delivers messages to one worker role instance at a time. A message may be delivered more than once if a previous recipient fails before marking the entry as successfully processed.

The application puts code that should run on the worker role instance into a class that it defines as a subclass of the **RoleEntryPoint** class that is provided by Microsoft Azure.

```
public class WorkerRole : RoleEntryPoint
{
   const string QueueName = "JobQueue";
   QueueClient mQueueClient;

   // ...
}
```

The **WorkerRole** class overrides the **OnStart** method of its base class **RoleEntryPoint**. This is shown in the following code.

```
public override bool OnStart()
{
  string serviceBusconnectionString =
       CloudConfigurationManager.GetSetting(
                                  "Microsoft.ServiceBus.ConnectionString");

  // ...

  // Initialize the connection to Service Bus Queue
  mQueueClient = QueueClient.CreateFromConnectionString(connectionString, QueueName);

  return base.OnStart();
}
```

Microsoft Azure invokes the **OnStart** method when the worker role is instantiated. The example code shows the same connection steps that you saw in the web client in the previous section. A **QueueClient** object is created from the connection string. The **WorkerRole** object will take items from the queue instead of adding them.

The **WorkerRole** class also overrides the **Run** method of the **RoleEntryPoint** base class. Here is the code.

```
public override void Run()
{
  mQueueClient.OnMessage((receivedMessage) =>
    {
       SearchTask task = null;
       try
       {
          mCurrentMessage = receivedMessage;
          //Save the current message because we may need to extend the lock on it.
          task  = receivedMessage.GetBody<SearchTask>();

          // ... (invoke BLAST processing) ... please refer to the source code

          //Make the message as completed
          mCurrentMessage.Complete();
       }
       catch
       {
         // Here we mark the message as Complete() instead of Abandon()
         // because we don't want to automatically retry
         // a failed task. User needs to re-submit the job.
         receivedMessage.Complete();
       }
    }, new OnMessageOptions { MaxConcurrentCalls = 1, AutoComplete = false});

  CompletedEvent.WaitOne();
}
```

The **Run** method of the worker role performs the main processing. In the case of the BLAST sample, the Run method blocks execution until the Service Bus queue notifies the **QueueClient** object that a message is ready to process. In that case, the **OnMessage** method invokes the lambda expression (receivedMessage) => { … } that is shown in the previous code sample. The lambda's body converts the message into the required **SearchTask** form and then runs the BLAST search job.

Storing Data in Microsoft Azure Table Service

In order for users to monitor BLAST job requests, the BLAST sample tracks all processed job requests. Because job requests are non-relational data, the data can be efficiently managed in a key/value store. The BLAST sample uses Table service for this purpose.

The application defines a reusable repository class to manage access to Table service. Here is the code.

```
  public class TableStoreageEntityRepo<E> : EntityRepository<string, E> where E : new()
  {
     private CloudTableClient mClient;
     private CloudTable mTable;
     private PropertyInfo mIdField;
     private PropertyInfo mPartitionField;
     // ...
}
```

The **TableStorageEntityRepo** class defines instance variables that contain a CloudTableClient object and a CloudTable object, which are created by the class's constructor. Here is the code.

```
public TableStoreageEntityRepo(string partitionFieldName, string idFieldName,
                               string connectionString, string tableName)
{
  // ...
  this.mClient = CloudStorageAccount.Parse(connectionString).CreateCloudTableClient();
  this.mTable = this.mClient.GetTableReference(tableName);
}
```

The constructor calls the static method CloudStorageAccount.Parse to connect to Table service and to create the CloudTableClient object. The **CloudTableClient**'s GetTableReference method returns a reference to the table that is used by the BLAST sample.

The following code example shows how the BLAST sample inserts new key/value pairs into the table.

```
public override void Create(E entity)
{
  this.mTable.Execute(TableOperation.Insert((ITableEntity) this.makeEntity(entity)),
                      (TableRequestOptions) null,
                      (OperationContext) null);
}
```

The example shows that adding an item to a table has three steps:

1. The application creates an instance that implements the Microsoft Azure ITableEntity interface. The example uses a helper method, **makeEntity**, to create the instance.
2. The application creates a TableOperation object that represents a request for access. This is somewhat analogous to a SQL statement. The **TableOperation** object in this example represents a request to add a row to the table.
3. The application executes the operation that was defined in step 2.

Storing Data in Microsoft Azure Blob Service

A BLAST search involves reading a set of large files. To perform efficient searches, each worker role instance needs a local copy of the genomic database that is used by BLAST.

The BLAST sample handles this requirement by storing the files in a central location by using Blob service. This storage service is designed to accommodate files up to 200 GB in size, so the BLAST files, the largest of which are around 1 GB, can be easily handled. The files in blob storage are considered the master copy.

When a worker role instance begins processing its first query, it copies the BLAST file set from the master copy and saves it locally. As a result, newly deployed instances have access to the same files as other running instances. The copy operation only occurs once for each worker role instance.

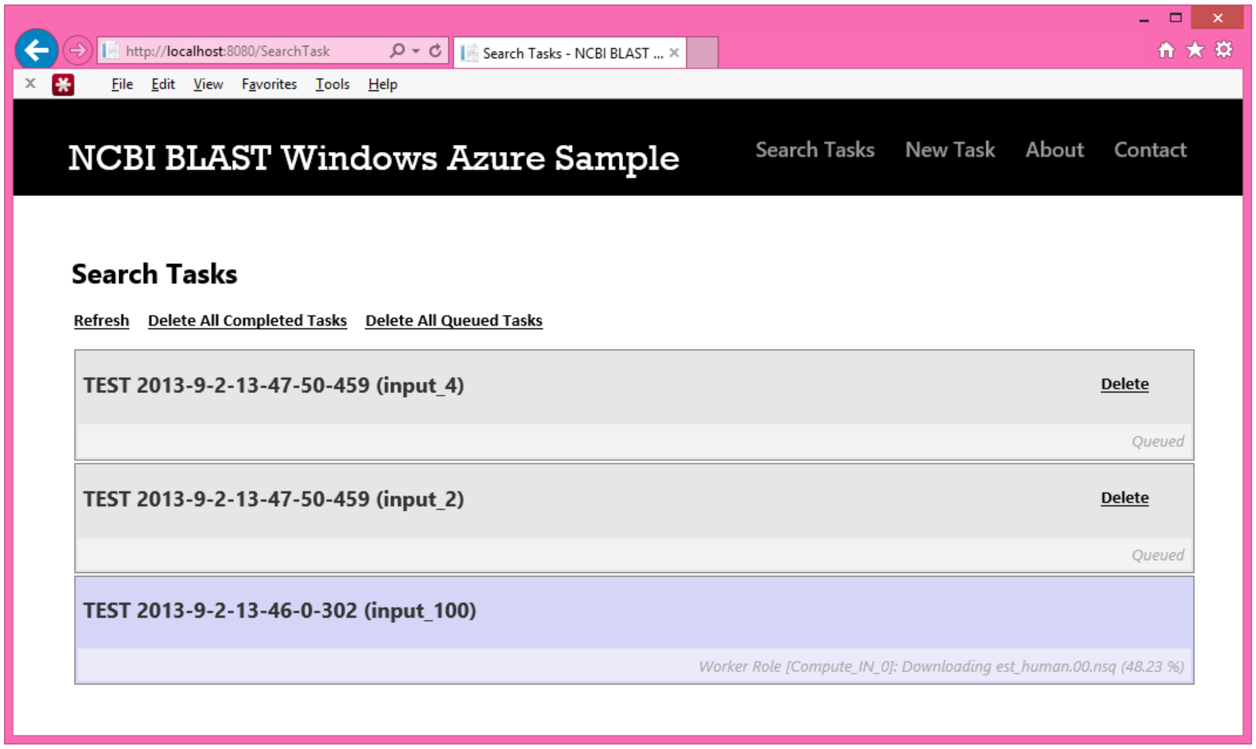
The following code shows the simplest way to create a local copy of data stored in Blob service.

```
using (var fileStream = System.IO.File.OpenWrite(@"c:\blastfiles"))

{

   blob.DownloadToStream(fileStream);

}
```

The BLAST sample's approach is more complicated because it shows real-time progress updates in the web UI during the download. This approach is shown in the following code.

```
using (Stream blobStream = blob.OpenRead())
{
   using (FileStream fileStream = new FileStream(filepath, FileMode.Create))
   {
      byte[] buffer = new byte[BUFFER_SIZE];
      int read;
      long count = 0;
      while ((read = blobStream.Read(buffer, 0, BUFFER_SIZE)) > 0)
      {
         fileStream.Write(buffer, 0, read);
         count += read;
         double percent = blob.Properties.Length;
         if (percent > 0)
            percent = count / percent;
         else
            percent = 0;
         raiseEvent(taskId, PENDING,
                    string.Format("Downloading {0} ({1:P})", blob.Name, percent));
      }
   }
}
```

The **raiseEvent** method sends signals to the web user interface by using the ASP.NET library **SignalR**. With this library, the method uses either HTTP long polling or the **WebSockets** protocol (if it is available) to request information. You can see the information sent by **SignalR** in the following screenshot. Notice the 48.23 percent download progress indicator at the bottom-right corner of the image.


http://localhost:8080/SearchTask
Search Tasks - NCBI BLAST ...
File Edit View Favorites Tools Help
NCBI BLAST Windows Azure Sample
Search Tasks
New Task
About
Contact

Search Tasks

Refresh Delete All Completed Tasks Delete All Queued Tasks

TEST 2013-9-2-13-47-50-459 (input_4) Delete
Queued

TEST 2013-9-2-13-47-50-459 (input_2) Delete
Queued

TEST 2013-9-2-13-46-0-302 (input_100)
Worker Role [Compute_IN_0]: Downloading est_human.00.nsq (48.23 %)

Figure 2: Information sent by SignalR

Summary of Websites

Here is a list of the websites listed in this paper.

The source code for the BLAST sample described in this paper can be found at: https://github.com/MSRConnections/Azure4Research-TechnicalPapers/tree/master/Scaling_a_Windows_Azure_Cloud_Service_BLAST.

For instruction on how to download and install the BLAST sample, see *Installing the Microsoft Azure BLAST Example* (http://research.microsoft.com/en-us/projects/azure/technical-papers.aspx).

For information about Microsoft Azure Service Bus, see the Service Bus documentation. (http://azure.microsoft.com/en-us/documentation/services/service-bus/)

For more information about BLAST go to the BLAST home page. (http://blast.ncbi.nlm.nih.gov/Blast.cgi?CMD=Web&PAGE_TYPE=BlastHome)

ASP.NET SignalR is a library that provides real-time web functionality easy. For more information, see SignalR (http://www.asp.net/signalr)

For information about the Microsoft Azure classes and methods discussed in this paper, see the following resources:

- CloudConfigurationManager class (http://msdn.microsoft.com/en-us/library/microsoft.windowsazure.cloudconfigurationmanager.aspx)
- GetSetting method (http://msdn.microsoft.com/en-us/library/microsoft.windowsazure.cloudconfigurationmanager.getsetting.aspx)

- QueueClient class (http://msdn.microsoft.com/en-us/library/microsoft.servicebus.messaging.queueclient.aspx)
- Send method (http://msdn.microsoft.com/en-us/library/microsoft.servicebus.messaging.queueclient.send.aspx)
- RoleEntryPoint class (http://msdn.microsoft.com/en-us/library/windowsazure/microsoft.windowsazure.serviceruntime.roleentrypoint.aspx)
- OnStart method (http://msdn.microsoft.com/en-us/library/windowsazure/microsoft.windowsazure.serviceruntime.roleentrypoint.onstart.aspx)
- Run method (http://msdn.microsoft.com/en-us/library/windowsazure/microsoft.windowsazure.serviceruntime.roleentrypoint.run.aspx)
- OnMessage method (http://msdn.microsoft.com/en-us/library/windowsazure/microsoft.servicebus.messaging.queueclient.onmessage.aspx)
- CloudStorageAccount.Parse method (http://msdn.microsoft.com/en-us/library/microsoft.windowsazure.cloudstorageaccount.parse.aspx)
- CloudTableClient class (http://msdn.microsoft.com/en-us/library/microsoft.windowsazure.storageclient.cloudtableclient.aspx)
- GetTableReference method (http://msdn.microsoft.com/en-us/library/microsoft.windowsazure.storage.table.cloudtableclient.gettablereference.aspx)
- ITableEntity interface (http://msdn.microsoft.com/en-us/library/microsoft.windowsazure.storage.table.itableentity.aspx)
- TableOperation object (http://msdn.microsoft.com/en-us/library/microsoft.windowsazure.storage.table.tableoperation.aspx)