WINDOWS AZURE TABLE

Jai Haridas, Niranjan Nilakantan, and Brad Calder
May2009

Table of Contents

1 Introduction

Windows Azure is the foundation of Microsoft’s Cloud Platform. It is the “Operating System for the Cloud” that provides essential building blocks for application developers to write scalable and highly available services. Windows Azure provides:

- Virtualized Computation
- Scalable Storage
- Automated Management
- Rich Developer SDK

Windows Azure Storage allows application developers to store their data in the cloud. The application can access its data from anywhere at any time, store any amount of data and for any length of time, and

be confident that the data is durable and will not be lost. Windows Azure Storage provides a rich set of data abstractions:

- Windows Azure Blob – provides storage for large data items.
- Windows Azure Table – provides structured storage for maintaining service state.
- Windows Azure Queue – provides asynchronous work dispatch to enable service communication.

This document describes Windows Azure Table, which is the structured storage provided by the Windows Azure platform. It supports massively scalable tables in the cloud, which can contain billions of entities and terabytes of data. The system will efficiently scale out by automatically scaling to thousands of servers as traffic grows.

Structured storage is provided in the form of Tables, which contain a set of Entities, which contains a set of named Properties. A few of the highlights of Windows Azure Table are:

- Support for LINQ, ADO .NET Data Services and REST.
- Compile time type checking when using the ADO .NET Data Services client library.
- A rich set of data types for property values.
- Support for unlimited number of tables and entities, with no limit on the table size.
- Strong consistency for single entity transactions.
- Optimistic concurrency for updates and deletes.
- For queries returning large numbers of results or queries that timeout, partial results are returned with a continuation token to allow the query to resume where it left off.

2 Table Data Model

The following summarizes the data model for Windows Azure Table:

- **Storage Account** – An application must use a valid account to access Windows Azure Storage. You can create a new account via the Windows Azure portal web interface. The user will receive a 256-bit secret key once the account is created. This secret key is then used to authenticate user requests to the storage system. Specifically, a HMAC SHA256 signature for the request is created using this secret key. The signature is passed with each request to authenticate the user requests.
  The account name is part of the host name in the URL. The hostname for accessing tables is <accountName>.table.core.windows.net.
- **Table** – contains a set of entities. Table names are scoped by the account. An application may create many tables within a storage account.
- **Entity (Row)** – Entities (an entity is analogous to a "row") are the basic data items stored in a table. An entity contains a set of properties. Each table has two properties, namely the “PartitionKey and RowKey” that form the unique key for the entity.
- **Property (Column)** – This represents a single value in an entity. Property names are case sensitive. A rich type set is supported for property values.

- **PartitionKey** – The first key property of every table. The system uses this key to automatically distribute the table's entities over many storage nodes.
- **RowKey** – A second key property for the table. This is the unique ID of the entity within the partition it belongs to. The PartitionKey combined with the RowKey uniquely identifies an entity in a table.
- **Timestamp** – Every entity has a version maintained by the system.
- **Partition** – A set of entities in a table with the same partition key value.
- **Sort Order** – There is a single index provided for the CTP, where all entities in a table are sorted by PartitionKey and then RowKey. This means that queries specifying these keys will be more efficient, and all results are returned sorted by PartitionKey and then by RowKey.

A table has a flexible schema. Windows Azure Table keeps track of the name and typed value for each property in each entity. An application may simulate a fixed schema on the client side by ensuring that all the entities it creates have the same set of properties.

The following are some additional details about Tables, Entities and Properties:

- Table
    - Table names may contain only alphanumeric characters.
    - A table name may not begin with a numeric character.
    - Table names are case-insensitive.
    - Table names must be from 3 through 63 characters long.
- Property Name
    - Only alphanumeric characters and '_' are allowed.
- An entity can have at most 255 properties including the mandatory system properties – PartitionKey, RowKey and Timestamp. All other properties in an entity have a name defined by the application.
- PartitionKey and RowKey are of string type, and each key is limited to 1KB in size.
- Timestamp is a read-only system maintained property which should be treated as an opaque property
- No Fixed Schema – No schema is stored by Windows Azure Table, so all of the properties are stored as <name, typed value> pairs. This means that two entities in the same table can have very different properties. A table can even have two entities with the same property name, but different types for the property value . However, property names must be unique within a single entity.
- Combined size of all data in an entity cannot exceed 1MB. This size includes the size of the property names as well as the size of the property values or their types, which includes the two mandatory key properties (PartitionKey and RowKey).
- Supported property types are: Binary, Bool, DateTime, Double, GUID, Int, Int64, String. See the table below for limits.

We use the default limits for Http.sys which enforces a 260 length limit on the URI segment. This results in a limitation on the size of partition and row key since GetRow, Delete, Update, Merge require that the partition and row key be specified as part of a single URI segment. For example, the following URI specifies a single entity with PartitionKey “pk” and RowKey “rk”: http://myaccount.windows.core.net/Customers(PartitionKey="pk",RowKey="rk"). Due to the Http.sys limitation, the highlighted portion cannot exceed 260 characters. To get around this problem, operations that have this limitation can still be performed using Entity Group Transactions, since in Entity Group Transactions the URI identifying the resource is part of the request body (see section 4.9 for more information).

Property Type	Details
Binary	An array of bytes up to 64 KB in size.
Bool	A Boolean value.
DateTime	A 64-bit value expressed as UTC time. The supported range of values is 1/1/1601 to 12/31/9999.
Double	A 64-bit floating point value.
GUID	A 128-bit globally unique identifier.
Int	A 32-bit integer.
Int64	A 64-bit integer.
String	A UTF-16-encoded value. String values may be up to 64 KB in size.

3 Partitioning Tables

Windows Azure Table allows tables to scale out to thousands of storage nodes by distributing the entities in the table. When distributing the entities, it is desirable to ensure that a set of entities always stay together on a storage node. An application controls this set by choosing an appropriate value for the PartitionKey property in each entity.

Applications need to understand their workload to a given partition, and stress with the simulated peak workload during testing, to make sure they will get the desired results.

Figure 1 Example of Partitions

The Figure above shows a table that contains multiple versions of documents. Each entity in this table corresponds to a specific version of a specific document. In this example, the partition key of the table is the document name, and the row key is the version string. The document name along with the version uniquely identifies a specific entity in the table. In this example, all versions of the same document form a single partition.

3.1 Impact of Partitioning

We now describe the purposes of the Table partitions and how to go about choosing a partition key.

3.1.1 Scalability of the table

The storage system achieves good scalability by distributing the partitions across many storage nodes. The system monitors the usage patterns of the partitions, and automatically balances these partitions across all the storage nodes. This allows the system and your application to scale to meet the traffic needs of your table. That is, if there is a lot of traffic to some of your partitions, the system will automatically spread them out to many storage nodes, so that the traffic load will be spread across many servers. However, a partition i.e. all entities with same partition key, will be served by a single node. Even so, the amount of data stored within a partition is not limited by the storage capacity of one storage node.

3.1.2 Entity Group Transactions

For the entities stored within the same table and same partition (i.e., they have the same partition key value), the application can atomically perform a transaction involving those entities. This allows the application to atomically perform multiple Create/Update/Delete operations across multiple entities in a single batch request to the storage system, as long as all the entities have the same partition key value and are in the same table. Either all the entity operations succeed in the single transaction or they all fail, and snapshot isolation is provided for the execution of the transaction. In addition, all other queries executing in parallel at the same time will not see the result of the transaction, since they will be working off a prior snapshot. Queries will only see the result of the transaction, once it has fully successfully committed.

Entity Group Transaction will require the use of the version header with the version set to `"2009-04-14"` or later. See section 4.1 for more details.

3.1.3 Entity Locality

The entities within the same partition are stored together. This allows efficient querying within a partition. Furthermore, your application can benefit from efficient caching and other performance optimizations that are provided by data locality within a partition.

In the above example, all versions of the same document form a single partition. Therefore, retrieval of "all of the versions of a given document" will be efficient, since we are accessing a single partition. On the other hand, a query for "all versions of documents modified before 5/30/2007" is not limited to a single partition. Since the query has to examine all partitions potentially across several storage nodes such a query would incur a higher cost.

3.2 Choosing a Partition Key

Choosing a partition key is important for an application to be able to scale well. There is a tradeoff here between trying to benefit from entity locality, where you get efficient queries over entities in the same partition, and the scalability of your table, where the more partitions your table has the easier it is for Windows Azure Table to spread the load out over many servers.

3.2.1 Entity Group Transactions

If your application needs to use entity group transactions, a PartitionKey needs to be chosen so that its granularity can capture all of the entities you need to perform atomic transactions over. Depending upon your query needs, one general rule of thumb, is to choose the ParitionKey such that it only groups together the entities that need to be grouped together to perform entity group transactions over them. This groups together the entities that need to be operated on atomically, while potentially creating many partitions to allow Windows Azure Table to load balance those partitions across our servers to meet your Table's traffic needs.

3.2.2 Efficient Queries

We recommend that high-frequency, latency-critical queries use the PartitionKey as a query filter condition. Using the ParititionKey in the query filter limits the query execution to a single or a subset of partitions (depending upon the condition used), thereby improving query performance.
If the PartitionKey is not part of the query, then the query has to be done over all of the partitions for the table to find the entities being looked for, which is not as efficient.

The following are some rough guidelines and suggestions for how choose a PartitionKey for your table for efficient querying:

1. First determine the important properties for your table. These are the properties frequently used as query filters.
2. Pick the potential keys from these important properties.
    a. It is important to identify the dominant query for your application workload. From your dominant query, pick the properties that are used in the query filters.
    b. This is your initial set of key properties.
    c. Order the key properties by order of importance in your query.
3. Do the key properties uniquely identify the entity? If not, include a unique identifier in the set of keys.
4. If you have only 1 key property, use it as the PartitionKey.
5. If you have only 2 key properties, use the first as the ParitionKey, and the second as the RowKey.
6. If you have more than 2 key properties, you can try to concatenate them into two groups – the first concatenated group is the PartitionKey, and the second one is the RowKey. With this approach, your application would need to understand that the PartitionKey for example contained two keys separated by a "-".

3.2.3 Scalability

Now that the application has its potential set of keys, you need to make sure that the partitioning chosen is scalable:

1. Given the PartitionKey above, will it result in partitions that will become too hot, based on your applications access patterns, to be served efficiently from a single server? One way to determine if this would result in a hot partition is to implement a Table partition stress test. For this test, create a sample table using your keys and then exert peak stress for your given workload on a single partition to ensure that the table partition can provide the desired throughput for your application.
2. If the Table partition stress test passes, then you are done.
3. If the Table partition stress test does not pass, select a more fine-grained PartitionKey. This could be done either by choosing a different PartitionKey or modifying the existing PartitionKey (for example, by concatenating it with the next key property). The purpose of this is to create more partitions so that a single partition does not become too large or too hot.
4. We designed the system to scale and be able to handle a large amount of traffic. However, an extremely high rate of requests may lead to request timeouts, while the system load balances. In that case, reducing your request rate may decrease or eliminate errors of this type. In general, most users will not experience these errors regularly; however, if you are experiencing high or unexpected Timeout errors, contact us at the MSDN Windows Azure forum to discuss how to optimize your use of Windows Azure Table and prevent these types of errors in your application.

3.2.4 Flexible Partitioning

You may also need to consider the extensibility of the keys you choose, especially if the user traffic characteristics are still unclear when the keys are chosen. In that case, it will be important to choose keys that can be easily extended to allow finer partitioning, so that if the partitioning turns out to be too coarse-grained, you can still extend your current partitioning scheme. A detailed example is discussed later in this document.

4 Programming Tables

The following basic operations are supported on tables and entities

- Create a table or entity
- Retrieve a table or entity, with filters
- Update an entity (but not a table)
- Delete a table or entity.
- Entity Group Transactions that support transactions across entities in the same table and partition.

To use tables in a .NET application, you can simply use ADO.NET Data Services.

The following table summarizes the APIs. Since the .NET ADO.NET Data Services api results in transmission of REST packets, applications can choose to use REST directly. Besides allowing non .NET languages access to the store, REST also allows fine grained control on serialization/deserialization of entities which is useful when dealing with scenarios such as a single table containing different kinds of entities or dealing wanting to have more properties for your object than is allowed for a given entity.

Operation	ADO.NET Data Services	HTTP Verb	Resource	Description
Query	LINQ Query	GET	Table	Returns the list of all tables in this storage account. If a filter is present, it returns the tables matching the filter
			Entity	Returns all entities in the specified table or a subset of entities if filter criteria are specified.
Update entire entity	UpdateObject & SaveChanges(`SaveChangesOptions.ReplaceOnUpdate`)	PUT	Entity	Updates property values within an entity. A PUT operation replaces the entire entity and can be used to remove properties.
Update partial entity	UpdateObject & SaveChanges()	MERGE	Entity	Updates property values within an entity.

Operation	ADO.NET Data Services	HTTP Verb	Resource	Description
Create new entity	AddObject & SaveChanges()	POST	Table	Creates a new table in this storage account.
			Entity	Inserts a new entity into the named table.
Delete entity	DeleteObject & SaveChanges()	DELETE	Table	Deletes a table in this storage account.
			Entity	Deletes an entity from the named table.
Entity group transaction	SaveChanges(SaveChangesOptions.Batch)	POST	$batch	Entity group transaction support is provided through a batch operation across entities having the same partition key in a single table. In ADO.NET Data Services, the option to SaveChanges dictates that the request needs to be sent as a single transaction.

Advanced operations on tables include the following, which will be discussed in more detail below

- Pagination
- Handling conflicts due to concurrent updates

4.1 Versioning

For all of the Windows Azure Storage solutions, we have introduced a new HTTP header called "x-ms-version". All changes to the storage APIs will be versioned by this header. This allows prior versions of commands executed against the storage system to continue to work, as we extend the capabilities of the existing commands and introduce new commands.

The x-ms-version should be specified for all requests coming to Windows Azure Storage. If there is an anonymous request without a version, then the oldest support version of that command will be executed by the storage system.

By PDC 2009, we plan to require the x-ms-version to be specified by all non-anonymous commands. Until then, if no version specified for a given request, we assume that the version of the command the request wants to execute is the CTP version of the Windows Azure Storage APIs from PDC 2008. If a request comes in an invalid x-ms-version, it will be rejected.

The current supported version is "x-ms-version: 2009-04-14". This can be used for all commands and requests sent to Windows Azure Storage. The new functionality we introduce for Windows Azure

Tables with this version is Entity Group Transactions, and specifying this version header is required to use this new feature.

```
// add the version header using SendingRequest event. This is the same
// place where the date header would have been added. However, the
// version header is not part of the canonicalized string used for
// creating the signature
context.SendingRequest +=
    new EventHandler<SendingRequestEventArgs>(
        delegate(object sender, SendingRequestEventArgs requestArgs)
        {
            HttpWebRequest request = requestArgs.Request as HttpWebRequest;
            request.Headers.Add(
                "x-ms-date",
                DateTime.UtcNow.ToString("R", CultureInfo.InvariantCulture));
            request.Headers.Add("x-ms-version", "2009-04-14");

            // ... add authorization header using shared key lite
        });
```

4.2 Running Example

In the examples below, we describe operations on a "Blogs" table. This table is used to hold blogs for a MicroBlogging application.

The MicroBlogging application has two tables – Channels and Blogs. There is a list of Channels, and blogs are posted to a particular channel. For this application, users would subscribe to channels and they would get the new blogs for those channels every day.

In this example, we only focus on the Blogs table, and give examples of the following steps for the Blogs table:

1. Define the schema for the table
2. Create the table
3. Insert a blog into the table
4. Get the list of blogs from the table
5. Update a blog in the table
6. Delete a blog from the table
7. Insert multiple blogs in a table

4.3 Defining the Entity Class for the Table

The schema for a table is defined as a C# class. This is the model used by ADO.NET Data Services. The schema is known only to the client application, and simplifies data access. The server does not enforce this schema.

The following shows the entity definition for the Blog entities to be stored in the Blogs table. Each blog entity has the following information.

1. A channel name – the blog has been posted to this channel.

2. The posted date.
3. Text – the content of the blog body.
4. Rating – the popularity of this blog.

For this table “Blogs”, we choose the channel name to be the PartitionKey and the posted date to be the RowKey. The PartitionKey and RowKey are the keys in the “Blogs” table and this is indicated by declaring the keys using an attribute on the class - DataServiceKey. The “Blogs” table is partitioned by the ChannelName. This allows the application to efficiently retrieve the latest blogs for a channel to which a user has subscribed. In addition to the keys, user specific attributes are declared as properties. All properties with a public getter and setter are stored in Windows Azure table. So in the below example:

- Text and Rating are stored for the entity instance in Azure table.
- RatingAsString is not because it does not have a setter defined.
- Id is not stored since the accessors are not public.

```
[DataServiceKey("PartitionKey", "RowKey")]
public class Blog
{
    // ChannelName
    public string PartitionKey { get; set; }
    // PostedDate
    public string RowKey { get; set; }

    // User defined properties
    public string Text { get; set; }
    public int Rating { get; set; }
    public string RatingAsString { get; }
    protected string Id { get; set; }
}
```

4.4 Creating a Table

Next we show how to create the “Blogs” table for your storage account. Creating a table is the same as creating an entity in a master table called “Tables”. Every storage account has this master table already defined, and every table used by a storage account must register the table name with this master table. The class definition for this master table is shown below where the TableName property represents the name of the table that is to be created.

```
[DataServiceKey("TableName")]
public class TableStorageTable
{
    public string TableName { get; set; }
}
```

The actual creation of the table happens as follows.

```
// Service Uri is “http://<Account>.table.core.windows.net/”
DataServiceContext context = new DataServiceContext(serviceUri);
TableStorageTable table = new TableStorageTable("Blogs");
```

```
// Create a new table by adding a new entity instance to the master
// table called "Tables"
context.AddObject("Tables", table);

// SaveChanges results in a call to the server
DataServiceResponse response = context.SaveChanges();
```

The serviceUri is the uri for the table service which is http://<Account name goes here>.table.core.windows.net/. The DataServiceContext is one of the main classes in ADO.NET data service which represents the runtime context for a service. It provides APIs to insert, update, delete and query entities either using LINQ or RESTful URIs and maintains state on the client end.

4.5 Inserting a Blog

To insert an entity, an application must do the following.

1. Create a new C# object and set all the properties.
2. Create an instance of DataServiceContext, which represents a connection to the server in ADO .NET data services for your storage account.
3. Add the C# object to the context.
4. Call the SaveChanges method on the DataServiceContext to send the request to the server. At this point an HTTP request is sent to the server with the entity in the ATOM XML format.

The following is the code examples for the above:

```
Blog blog = new Blog {
    PartitionKey = "Channel9",            // ChannelName
    RowKey  = DateTime.UtcNow.ToString(), // PostedDate
    Text    = "Hello",
    Rating  = 3
};

serviceUri = new Uri("http://<account>.table.core.windows.net");
var context = new DataServiceContext(serviceUri);

context.AddObject("Blogs", blog);
DataServiceContext response = context.SaveChanges();
```

4.6 Querying Blogs

Entities are queried using LINQ – the Language Integrated Query in C#. In this example we retrieve all blogs whose rating is 3.

When the query is enumerated (e.g. with a foreach statement), a request is sent to the server. The server sends the results in the ATOM XML format. The ADO .NET Data Services client library deserializes the results into C# objects for the application to use.

```
var serviceUri = new Uri("http://<account>.table.core.windows.net");
DataServiceContext context = new DataServiceContext(serviceUri);
// LINQ query using the DataServiceContext for all blog entities from
// Blogs table where rating = 3
var blogs =
```

```
    from blog in context.CreateQuery<Blog>("Blogs")
    where blogs.Rating == 3
    select blog;

// Enumerating over blogs is when the query is sent to the server
// and executed
foreach (Blog blog in blogs) { ... }
```

4.7 Updating a Blog

To update an entity, the application should do the following.

1. Create a DataContext with MergeOption set to OverwriteChanges or PreserveChanges as described in section 4.10. This ensures that the ETag is maintained correctly for each object that is retrieved.
2. Fetch the entity to be updated into the DataContext using LINQ. Retrieving it from server will ensure that the ETag is updated in the entities tracked by the context and subsequent updates and deletes will use the updated ETag in the if-match header. Modify the C# object that represents the entity.
3. Add the C# object back to the same DataContext for update. Using the same DataContext ensures that we automatically reuse the ETag that was fetched earlier for this object.
4. Call SaveChanges to send the request to the server.

```
Blog blog =
   (from blog in context.CreateQuery<Blog>("Blogs")
    where blog.PartitionKey == "Channel9"
       && blog.RowKey == "Oct-29"
    select blog).FirstOrDefault();

blog.Text = "Hi there";

context.UpdateObject(blog);
DataServiceResponse response = context.SaveChanges();
```

4.8 Deleting a Blog

Deleting an entity is similar to updating the entity. Use the DataServiceContext to retrieve the entity and then invoke DeleteObject method on the context instead of the UpdateObject method.

```
// Get the Blog object for ("Channel9", "Oct-29") in the query above
context.DeleteObject(blog);
DataServiceResponse response = context.SaveChanges();
```

4.9 Entity Group Transactions

Entity group transactions support executing up to 100 CUD commands in a single batch against entities in a single table and having the same partition key value. These commands are executed atomically i.e. all succeed or none.

Except for certain limitations, we follow the protocol that ADO.NET data service uses for batch transactions (see http://msdn.microsoft.com/en-us/library/cc668802.aspx for details). Here are some terms that are batch specific:

- Change set - Group of one or more CUD commands that are executed atomically.
- Batch - A batch contains one or more change set or queries.

The following are the limitations on using a batch command for Windows Azure Table:

- We allow a maximum of 100 operations in a change set.
- We allow only a single change set or query per batch operation. Query cannot be combined with CUD operations in a given batch operation. We only support queries that deterministically returns a single row i.e. the filter has PartitionKey and RowKey specified with '=' as the qualifying filter.
- A single batch can be at most 4MB in size.
- All commands in a change set must belong to a single partition in a single table. This means that the partition key value must be the same for all entities and the operations should execute against a single table.
- Each entity can only appear once in the change set. You cannot have a change set that performs multiple operations on the same entity. The change set is meant to perform multiple operations across different entities in the same partition and table as an atomic transaction.
- The individual commands in a change set are executed in the order in which they appear in the change set.

While issuing a transaction, the ETag is sent in each changeset as described in the MSDN document and not as a header for the entire request. On the server, we check if the ETag matches with the value stored on the server. If there is a ETag mismatch, the entire transaction fails.

The following continues the blob example showing the usage of an Entity Group Transaction to atomically create, update and delete multiple entities from a partition in a single batch operation.

```
// newBlogs is the list of new blogs in Channel_19 partition
// deletedBlogs is the list of blogs to delete in Channel_19 partition
// updatedBlogs is the list of blogs to update in Channel_19 partition
// We assume that the deletedBlogs and updatedBlogs have been fetched
// into the context in earlier queries.

// Insert multiple blog objects for partition "Channel_19".
// newBlogs is a list of blogs that have been selected for
// insertion outside of this function
for (int index = 0; index < newBlogs.Length; index++)
{
    context.AddObject(newBlogs[index]);
}

// Delete old blogs objects for Channel_19.
// deletedBlogs is a list of blogs that have been selected for
// deletion outside of this function
for (int index = 0; index < deletedBlogs.Length; index++)
{
```

```
    context.DeleteObject(deletedBlogs[index]);
}

// Update existing blogs objects for Channel_19.
// updatedBlogs is a list of blogs that have been selected to
// increase the rating
for (int index = 0; index < updatedBlogs.Length; index++)
{
    updatedBlogs[index].Rating++;
    context.UpdateObject(updatedBlogs[index]);
}

// All CUD operations above are executed as a single batch request.
DataServiceResponse response  =
         context.SaveChanges(SaveChangesOptions.Batch);
```

Specifying SaveChangeOptions.Batch indicates to the SaveChanges command that all the pending changes are grouped into a single change set and sent as a single Entity Group Transaction to the storage system. The batch transaction is then checked on the server to ensure that it adheres to the above restrictions.
Now, if the above SaveChanges was done without the SaveChangesOptions.Batch, then each entity Create/Update/Delete would be sent as a separate request to the storage system and executed separately.

The following shows executing a single query in a batch command. Note, a batch command can either be an Entity Group Transaction as shown above, or a single query as shown below. This is useful because it allows us to query (Get) a single row using a URI format that represents just a single entity as a workaround for the 260 character limit on URI segment length imposed by HTTP.sys (see Section 2).

```
// Execute a query in a batch- Note the filter has to be present
// and should be of the form of:
// PartitionKey == 'Some Value' && RowKey == 'Some Value'
var q1 = from o in context.CreateQuery<Blogs>("Blogs")
                where o.PartitionKey == "Channel_19" &&
                      o.RowKey == "2"
                select o;
DataServiceResponse response =
       context.ExecuteBatch((DataServiceQuery<RetailStoreV1>)q1);
```

For sample traces, please refer to MSDN documentation.

NOTE: A non-batch solution around this problem is to construct the query using the below syntax which results in the partition and row key filters to be specified as part of the $filter query parameter rather than in the URI segment.

```
from o in context.CreateQuery<Blogs>("Blogs")
 where o.PartitionKey == "Channel_19"
select o into d
where d.RowKey == "2"
 select d;
```

4.9.1 Handling the response

If the batch is accepted and understood by the server, the response to a batch request will always be the return code 202 Accepted. Responses to the individual operations within a batch are returned within the payload of a batch in the response element. Conceptually, the response from a batch matches to the request one-to-one unless an operation within a change set fails. In this case, only a single response is returned for the change set because the operations of change sets are atomic.

- If all operations in a change set succeed, a response is returned for each operation within the change set. If an operation in the set fails, only a single response object is returned, which represents the single response for all operations in the change set.
- HTTP eTags are transferred using the same mechanisms as defined in the optimistic concurrency extension specification to the core protocol with the response element playing the role of HTTP response headers for a specified operation.

4.9.2 Errors

When a batch request fails, none of the operations succeed. However, it is important for an application to know which command caused the entire transaction to fail. To determine this, we return the sequence number of the command in the error message as seen in the highlighted portion of the response below. However, if the error occurred due to authorization or any other reason that is not specific to a single command, then the sequence number is not provided. Hence, the payload in the event of a failure will contain:

1. The key of the operation that failed.
2. The zero based index of the operation in the batch that caused the failure (the index starts at 0).
3. A string describing the failure.

Example Response:

```
HTTP/1.1 202 Accepted
Cache-Control: no-cache
Transfer-Encoding: chunked
Content-Type: multipart/mixed; boundary=batchresponse_7ab1553a-7dd6-44e7-8107-bf1ea1ab1876
Server: Table Service Version 1.0 Microsoft-HTTPAPI/2.0
x-ms-request-id: 45ac953e-a4a5-42ba-9b4d-97bf74a8a32e
Date: Thu, 30 Apr 2009 20:45:13 GMT

6E7
--batchresponse_7ab1553a-7dd6-44e7-8107-bf1ea1ab1876
Content-Type: multipart/mixed; boundary=changesetresponse_6cc856b4-8cb9-41eb-b8d2-
bb73475c6cec

--changesetresponse_6cc856b4-8cb9-41eb-b8d2-bb73475c6cec
Content-Type: application/http
Content-Transfer-Encoding: binary

HTTP/1.1 400 Bad Request
Content-ID: 4
Content-Type: application/xml
Cache-Control: no-cache
DataServiceVersion: 1.0;

<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<error xmlns="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata">
  <code>InvalidInput</code>
  <message xml:lang="en-US">1:One of the request inputs is not valid.</message>
</error>
--changesetresponse_6cc856b4-8cb9-41eb-b8d2-bb73475c6cec--
--batchresponse_7ab1553a-7dd6-44e7-8107-bf1ea1ab1876--
```

```
0
```

When an exception occurs, you can extract the sequence number (highlighted above) of the command that caused the transaction to fail as follows:

```
try
{
    // ... save changes
}
catch (InvalidOperationException e)
{
    DataServiceClientException dsce = e.InnerException as DataServiceClientException;
    int? commandIndex;
    string errorMessage;

    ParseErrorDetails(dsce, out commandIndex, out errorMessage);
}

void ParseErrorDetails(
    DataServiceClientException e,
    out string errorCode,
    out int? commandIndex,
    out string errorMessage)
{

    GetErrorInformation(e.Message, out errorCode, out errorMessage);

    commandIndex = null;
    int indexOfSeparator = errorMessage.IndexOf(':');
    if (indexOfSeparator > 0)
    {
        int temp;
        if (Int32.TryParse(errorMessage.Substring(0, indexOfSeparator), out temp))
        {
            commandIndex = temp;
            errorMessage = errorMessage.Substring(indexOfSeparator + 1);
        }
    }
}

void GetErrorInformation(
    string xmlErrorMessage,
    out string errorCode,
    out string message)
{
    message = null;
    errorCode = null;

    XName xnErrorCode = XName.Get(
        "code",
        "http://schemas.microsoft.com/ado/2007/08/dataservices/metadata");
    XName xnMessage = XName.Get(
        "message",
        "http://schemas.microsoft.com/ado/2007/08/dataservices/metadata");

    using (StringReader reader = new StringReader(xmlErrorMessage))
    {
        XDocument xDocument = null;
        try
        {
            xDocument = XDocument.Load(reader);
        }
        catch (XmlException)
        {
            // The XML could not be parsed. This could happen either because the connection
            // could not be made to the server, or if the response did not contain the
            // error details (for example, if the response status code was neither a failure
            // nor a success, but a 3XX code such as NotModified.
            return;
```

```
        }

        XElement errorCodeElement =
            xDocument.Descendants(xnErrorCode).FirstOrDefault();

        if (errorCodeElement == null)
        {
            return;
        }

        errorCode = errorCodeElement.Value;

        XElement messageElement =
            xDocument.Descendants(xnMessage).FirstOrDefault();

        if (messageElement != null)
        {
            message = messageElement.Value;
        }

    }
}
```

4.10 Best Practices when using DataServiceContext

Here are some best practices to be aware of while using DataServiceContext:

- The DataServiceContext object is not thread safe, so it is not meant to be shared amongst different threads. The DataServiceContext is not meant to have a long lifetime. Instead of having a single DataServiceContext for the life of a thread, it is recommended to create a DataServiceContext object each time you need to do a set of transactions against WindowsAzureTable, and then delete the object when done.
- When using the same DataServiceContext instance across all inserts/updates/deletes, if there is a failure when doing SaveChanges, the operation that failed is kept track of in the DataServiceContext and re-tried the next time SaveChanges is invoked.
- DataServiceContext has a MergeOption which is used to control how the DataServiceContext handles the tracked entities. The possible values are:
    - AppendOnly: This is the default value where the DataServiceContext does not load the entity instance from the server if it is already present in its cache.
    - OverwriteChanges: DataServiceContext always loads the entity instance from the server hence keeping it up-to-date and overwriting the previously tracked entity.
    - PreserveChanges: When an entity instance exists in the DataServiceContext, it is not loaded from persistent storage. Any property changes made to objects in the DataServiceContext are preserved but the ETag is updated hence making it a good option when recovering from optimistic concurrency errors.
    - NoTracking: Entity instances are not tracked by the DataServiceContext. To update an entity on a context that is not tracking will require the use of AttachTo to update the etag to use for the update and hence is not recommended.

```
context.AttachTo("Blogs", blog, "etag to use");
context.UpdateObject(blog);
context.SaveChanges();
```

When MergeOption on the context is set to AppendOnly and the entity is already tracked by a previous retrieve or add operation with the DataServiceContext object, then retrieving an entity again from server will not update the tracked entity in the context. So, if the entity on server has changed, it will result in PreCondition failure on subsequent updates/deletes. See code sample in section 5 where we set the MergeOption to be PreserveChanges which always loads the entity from the server.

- Due to a known performance issue with the ADO.NET Data Services client library, it is recommended that you use the table name for the class definition or define the ResolveType delegate on the DataServiceContext. If this is not used and when the entity class name is not the same as table name, query performance degrades with number of entities returned in the result. An example of using ResolveType delegate is shown below:

```
public void Query(DataServiceContext context)
{
    // set the ResolveType to a method that will return the appropriate
    // type to create
    context.ResolveType = this.ResolveEntityType;
    ...
}

public Type ResolveEntityType(string name)
{
    // if the context handles just one type, you can return it without checking the
    // value of "name".  Otherwise, check for the name and return the appropriate
    // type (maybe a map of Dictionary<string, Type> will be useful)
    Type type  = typeof(Customer);
    return type;
}
```

4.11 Using the REST API

All of the operations above result in HTTP messages to and from the server. An application could choose to work at the HTTP/REST level instead of using the .NET client library.

5 Concurrent Updates

Updating an entity requires the following operations.

1. Get the entity from the server
2. Update the object locally, and send it back to the server.

Let us assume that two concurrent processes are trying to update the same entity. Since steps 1 and 2 are not atomic, one of them could end up modifying a stale version of the entity. Windows Azure Table uses optimistic concurrency to address this problem.

1. Each entity has a system maintained version that is modified by the server on every update.
2. When an entity is retrieved, the server sends this version to the client as an HTTP ETag.
3. When the client sends an UPDATE request to the server, it sends this ETag to the server as an If-Match header.

4. If the version of the entity on the server is the same as the ETag in the If-Match header, the change is accepted, and the entity stored gets a new version on the server. The new version is returned back to the client as an ETag header.
5. If the version of the entity on the server is different than the ETag in the If-Match header, the change is rejected, and a "precondition failed" HTTP error is returned to the client.

When a client application gets a "precondition failed" error, its typical behavior will be to retry the operation as shown below in the code.

1. It should retrieve the object again, thus getting the latest version
2. Apply the update locally and send it back to the server.

If the application uses the .NET client library, the HTTP error code is presented to the application as an exception (DataServiceRequestException).

In the example below, two different clients execute the same code to change the text. The two clients are trying to set the property "Text" to different values. The following is a possible sequence of events illustrating how concurrent updates are handled.

1. Both clients fetch the entity. This also fetches the ETag for the entity e.g. "v1". Both clients think the prior version is v1.
2. Each client updates the Text property locally.
3. Each client calls UpdateObject and SaveChanges.
4. Each client sends an HTTP request to the server with an "If-Match: v1" header.
5. One of the clients reaches the server first.
    a. The server compares the "If-Match" header with the entity's version. They happen to be equal.
    b. The server applies the change.
    c. The entity gets a new version "v2" on the server.
    d. A new "ETag:v2" header is sent as a response to the client.
6. The other client reaches the server next. By this time, the first client's changes have been applied.
    a. The server compares the "If-Match" header with the entity's version. They are not equal, since the entity's version has changed to "v2", while the request's version is "v1".
    b. The server rejects the request.

```
// Set the merge option to preserve the updates but allow etag to be updated
// The default is AppendOnly which does not overwrite an already tracked entity
// with the values retrieved from server which will result in an invalid etag being used
//  if the entity on the server has changed.
context.MergeOption = MergeOption.PreserveChanges;
Blog blog =
    (from blog in context.CreateQuery<Blog>("Blogs")
     where blog.PartitionKey == "Channel9"
        && blog.RowKey == "Oct-29"
     select blog).FirstOrDefault();
```

```
blog.Text = "Hi there again";
try
{
         context.UpdateObject(blog);
         DataServiceResponse response = context.SaveChanges();
}
catch (DataServiceRequestException e)
{
   OperationResponse response = e.Response.First();

   if (response.StatusCode == (int)HttpStatusCode.PreconditionFailed)
   {
         // query the object again to retrieve the latest etag
         // and update it
    }
}
```

5.1 Unconditional updates

If an application wants to update an entity unconditionally it should do the following

1. Create a new DataServiceContext or if using an existing context, detach the object as seen in the example below.
2. Attach the entity to the context and use "*" as the new ETag value.
3. Update the entity.
4. Invoke SaveChanges.

```
// set the merge option to overwrite to allow the tracked entity to be updated
context.Detach(blog);

// Attach the entity to the context using the table name, entity to
// update and "*" as the etag value to use.
context.AttachTo("Blogs", blog, "*");
blog.Text = "Hi there again";
 try
 {
     context.UpdateObject(blog);
     DataServiceResponse response = context.SaveChanges();
 }
catch (DataServiceRequestException e)
{
     // Error handling - but it cannot throw a PreCondition failure
}
```

6 Pagination of query results

For queries that may return a large number of results, the system provides two mechanisms:

1. The ability to get the first N entities using the LINQ Take(N) function.
2. A continuation token that indicates where the next set of results starts.

6.1 Getting Top N Entities

The system supports returning the top N entities that match the query. For example, in .NET, you can use the following LINQ Take(N) function to retrieve the top N entities (top 100 entities in this example).

```
serviceUri = new Uri("http://<account>.table.core.windows.net");
DataServiceContext svc = new DataServiceContext(serviceUri);

var allBlogs = context.CreateQuery<Blog>("Blogs");

foreach (Blog blog in allBlogs.Take(100))
{
   // do something with each blog
}
```

The same functionality is supported via REST interface with $top=N query string option. For example, the query “GET http://<serviceUri>/Blogs?$top=100" would return the top 100 entities that matches the query. The filtering is done on the server end such that at most only 100 entities are sent in the response to the client.

6.2 Continuation Tokens

The number of entities returned in the query may be limited for one of the following reasons :

1. The request specified a maximum number of entities to be returned.
2. The number of entities is greater than the maximum number of entities allowed in the response by the server (currently 1000).
3. The total size of the entities in the response is greater than the maximum size of a response (currently 4MB including the property names but excluding the xml tags used for REST).
4. The query executed for more than the server defined timeout (currently 60 seconds).

In each of these cases, the response includes a continuation token as custom headers. For a query over your entities, the custom headers representing a continuation token are :

- x-ms-continuation-NextPartitionKey
- x-ms-continuation-NextRowKey

The client should pass both these values, if they exist, back into the next query as HTTP query options, with the rest of the query remaining the same. The client will then get the next set of entities starting at the continuation token.

The next query looks as follows
http://<serviceUri>/Blogs?<originalQuery>&NextPartitonKey=<someValue>&NextRowKey=<someOtherValue>

The client would repeat this until there is no continuation token. At this point, all the matching results would have been retrieved.

The continuation token must be treated as an opaque value. It reflects the starting point for the next query, and may not correspond to an actual entity in the table. If a new entity is added such that Key(new entity) > Key(last entity retrieved in a query), but Key(new entity) < “Continuation token”, then this new entity will not be returned when the query is reissued using the continuation token. However, new entities added such that Key(new entity) > Continuation token, will be returned in a subsequent query issued using the continuation token.

7 Consistency Model

Now we describe the consistency model provided by Windows Azure Table.

7.1 Single Table Consistency

Within a single table, the system provides ACID transaction guarantees for all the insert/update/delete transactions for a single entity.

Within a single partition, snapshot isolation is provided for queries. A query will have a consistent view of the partition from the start time of the query and throughout the transaction. The snapshot has the following properties.

1. There will be no dirty reads. A transaction will not see uncommitted changes from other transactions which execute concurrently. It will only see the changes committed before the start of this query's execution at the server.
2. The snapshot isolation mechanism allows reads to happen concurrently with an update to the partition without blocking that update.

Snapshot isolation is only supported within a partition and within a single query. The system does not support snapshot isolation across table partitions or across different continuations of a query.

7.2 Cross-Table Consistency

Applications are responsible for maintaining consistency across multiple tables.

In the MicroBlogging example, we had two tables – Channels and Blogs. The application is responsible for maintaining the consistency between the Channels table and the Blogs table. For example, when a channel is removed from the Channels table, the application should delete the corresponding blogs from the Blogs table.

Failures can occur while the application is synchronizing the state of multiple tables. The application needs to be designed to handle such failures, and should be able to resume from where it left off.

In the previous example, when a channel is removed from the channel table, the application also needs to delete all the blogs for that channel in the Blogs table. During this process, the application may have failures. To tolerate such failures, the application can persist the transaction in Windows Azure Queues, so that a worker can resume the deletion of the channel and all its blogs if there is a failure.

Let us continue with the example where we have the "Channels" and "Blogs" tables. An entity in the "Channels" table has the following properties: {Name as PartitionKey, empty string as RowKey, Owner, CreatedOn} and an entity in "Blogs" has {Channel Name as PartitionKey, CreatedOn as RowKey, Title, Blog, UserId}. Now when a channel is deleted, we will need to ensure that all blogs associated with the channel are also deleted. Here are the steps we will take:

1. Create a queue to maintain cross table consistency – let us call it "DeleteChannelAndBlogs"

2. When a delete request arrives for a channel at a web front end role, enqueue an entry into the above queue specifying the channel name
3. Create worker roles that listen for entries added to the “DeleteChannelAndBlogs” queue.
4. A worker role dequeues an entry from DeleteChannelAndBlogs queue setting the invisibility time for the queue entry retrieved to N seconds. This retrieves an entry specifying a channel name to be deleted. If the queue entry is deleted by the worker role within N seconds then the queue entry will be deleted from the queue. If not, then the queue entry will become visible again for a worker to consume. When a worker dequeues an entry it does the following:
    a. Mark the channel as invalid in the “Channels” table, so that from this point on, no one reads from it.
    b. Delete all entities in the Blogs table with PartitionKey = “channel name” received in the queue entry.
    c. Delete the channel from the “Channels” table.
    d. Delete the queue entry from the queue.
    e. Return.

If any failure occurs during 4, for example, the worker process crashes and we did not delete the entry from the queue, this message will be dequeued by a worker process again at a later time once the queue entry becomes visible again (i.e. time defined by the visibility timeout), and the delete process in step 4 will resume. Please see the Windows Azure Queue documentation for more queue details.

8 Tips and Tricks

In this section we will discuss some common scenarios that an application may run into and a possible solution(s) that one can use.

8.1 Retrieve latest items (simulating descending order)

In applications that manage email, news, blogging etc., the most recently added item is usually required to be displayed first. Given that all rows are lexically sorted based on PartitionKey and RowKey, it becomes a little tricky to get the most recently created item first.

To achieve this, we can set the RowKey to be a fixed length string equivalent of DateTime.MaxValue.Ticks - DateTime.UtcNow.Ticks. This allows the RowKey to sort the items by an offsetted time from newest items to older items.

For example:

```
Blog blog=  new Blog();
// Note the fixed length of 19 being used since the max tick value is 19 digits long.
string rowKeyToUse = string.Format("{0:D19}",
                DateTime.MaxValue.Ticks - DateTime.UtcNow.Ticks);
blog.RowKey = rowKeyToUse;
```

So a blog b1 dated 10/1/2008 10:00:00 AM will have 252179445599999999 as the RowKey, and b2 dated 10/2/2008 10:00:00 AM will have 252179359199999999 as the RowKey and hence b2 will preceede b1.

To retrieve all blogs dated after 10/1/2008 10:00:00 AM, we will use the follwing query:

```
string rowKeyToUse = string.Format("{0:D19}",
        DateTime.MaxValue.Ticks - DateTime.UtcNow.Ticks);
var blogs =
    from blog in context.CreateQuery<Blog>("Blogs")
    where blog.PartitionKey == "Football"
         && blog.RowKey.CompareTo(rowKeyToUse) > 0
    select blog;
```

NOTE: The fixed length string enforcement should be sufficiently large to handle a large range. For example: The maximum length of DateTime.MaxValue.Ticks - DateTime.UtcNow.Ticks is 19 digits. However, if we start handling dates starting 2/15/6831 2:13:20 PM, the length is 18 digits. Sorting will give us undesired results if we do not pad the values with '0' at the beginning since sorting is lexicographic i.e. 92 > 10000. However if we padded it appropriately, 10000 > 00092 .

Since for the CTP release, we allow only one RowKey. Even so, the RowKey can actually be used to represent multiple keys through string concatenation.. For example, let us assume that a blog can be rated from 0-5 and we want to ensure that blogs are listed such that they are sorted by (Rating in desc order, Created time in desc order). This means that they are storted first by rating, and then sorted within each rating by the created time. Here we can set the row key to <Rating>+<Ticks to force items to be sorted from newest to older items>. So taking the same example as above, in the two blogs above, let us assume blog1 was rated 5 and blog2 was rated 1. Having the Rating precede the ticks will allow blog1 to be listed before blog2 despite the fact that blog2 was blogged after blog1.

Since the PartitionKey and RowKey have to be strings, you may use the following convention to use other types in these properties. This ensure that values are sorted in the expected manner. For example,

- Integer – Store it as a fixed sized value with leading zeroes.
- DateTime – store it as yyyy-mm-dd, or yyyy-mm-dd-hh-ss if more precision is needed and oldest items precede newer items or use ticks as explained above if newer entities need to be listed before older ones.

8.2 Retrieve using prefix

Since prefix matching is not supported, it can be mimicked by using >= & < filter on the property. For example, if each PDC had a channel of its own and all PDC blogs are to be retrieved, then the query consists of retrieving all blogs from all channels where Partitionkey starts with “PDC”. This can be achieved with the following query:

```
var blogs =
    from blog in context.CreateQuery<Blog>("Blogs")
```

```
    where blog.PartitionKey.CompareTo("PDC") >= 0
         && blog.PartitionKey.CompareTo("PDD") < 0
select blog;
```

8.3 Data Partitioning Example

Data partitioning involves grouping the entities of a table into partitions such that a partition becomes the smallest unit controlled for load balancing. Windows Azure Table uses the PartitionKey to group entities into partitions. In this section we describe how to partition your data for best performance.

8.3.1 Micro Blogging Case Study

Let us take the example of a micro blogging application where users can create channels (or categories) and create blogs in them. When a user selects a channel, a paged view of blogs is displayed.

We will assume that a blog in the Blogs table is uniquely identified by the channel name and the date the blog was created. The dominant query we are optimizing the table for is "Get me the 10 most recent blogs for a channel".

In this example we will look at various options for choosing the PartitionKey where we will vary the partition size from very large to the partition size being just a single entity. This varies the size and the number of partitions.

8.3.1.1 Having a few large partitions

An obvious option for PartitionKey is the channel name. We will also use the method described in section 8.1 to ensure that blogs are sorted by the RowKey from newest inserted blog to oldest blog in the channel. The following is the entity class definition for the Blogs table with this partitioning approach:

```
[DataServiceKey("PartitionKey", "RowKey")]
public class Blog
{
    // Channel name
    public string PartitionKey { get; set; }

    // To sort RowKey from newest to oldest blog,
    // RowKey is DateTime.MaxValue.Ticks - DateTime.UtcNow.Ticks
    public string RowKey { get; set; }

    // User defined properties
    public string Text { get; set; }
    public DateTime CreatedOn { get; set; }
    public int Rating { get; set; }
}
```

This allows us to efficiently query all the blogs in the "Football" channel and display them, and they will be sorted by newest inserted blog to oldest based on how the RowKey was set above.

```
var blogs =
    from blog in context.CreateQuery<Blog>("Blogs")
    where blog.PartitionKey == "Football"
    select blog;

foreach (Blog blog in blogs) { ... }
```

With this partitioning scheme, the unit for load balancing is the channel. Therefore, blogs to different channels are in different partitions and these partitions can be spread across different servers to serve the traffic needs of the “Blogs” table.

8.3.1.2 Having many small partitions

Now if a channel can potentially contain 10s of millions of blogs with many hot sub discussion areas, it may not be suitable to have just the channel name as the key for partitioning as it leads to a single partition containing 10s of millions of blogs and these hot discussion areas cannot be load balanced to different servers independent of each other.

One possible solution to avoid having monolithic partitions would be to make the PartitionKey a combination of the channel name and the datetime on which the blog was created. With this the PartitionKey would uniquely define each entity, and each partition in the table would thus contain a single entity. This is great for load balancing, since it gives the storage system complete freedom to load balance all of the entities as it chooses across the servers to meet the applications needs. Depending on how the partitions are distributed across servers, this may make queries like “Get me all the blogs in the past day or past week” potentially expensive, as it could require accessing several storage nodes which can be expensive.

8.3.1.3 An intermediate partition size

An alternative would be to group the blogs based on channel plus a coarse time-period (weeks/months/days) based on the popularity of the channels.. If the channel gets hot during certain months or weeks or days, then it will be beneficial to partition the blogs using time on which the blog was created. In this way, the PartitionKey includes both the channel name and the time period information. This allows finer grained partitioning and the system has more flexibility to spread the load across many servers to meet the traffic needs. Meanwhile, since the time period is encoded into the PartitionKey, queries like "Get me all the blogs in channel X in the past day" will be more efficient, since only a small targeted set of partitions need to be queried upon.

For examplein r the “Football” channel, the hot periods may be during the season which is September through February. It will be better for the application to partition the table by the month or week in addition to the channel name. If “week” was used in the PartitionKey, hot partitions such as that for Super Bowl can be load balanced to maintain high availability and serve queries efficiently.

Since Windows Azure Table supports a single PartitionKey, the micro blogging application can use <Channel_Name>_<9999 - Year>_<5 - Week#> or <Channel_Name>_<9999 - Year>_<13 - Month> as the PartitionKey for this table. The choice between using <9999 - Year>_<13 - Month> or <9999 - Year>_<5 - Week#> depends on whether the blogs would be best served if load balanced on a weekly or monthly basis. In this example, we are subtracting the date component from an upper bound, in order to have the entities sorted from newest to oldest. The following is the example entity class for this partitioning approach:

```
[DataServiceKey("PartitionKey", "RowKey")]
public class Blog
{
    // <Channel name>_<9999-YYYY>_<13-Month>
    public string PartitionKey { get; set; }


    // To sort RowKey from newest to oldest blog,
    // RowKey is DateTime.MaxValue.Ticks - DateTime.UtcNow.Ticks
    public string RowKey { get; set; }

    // User defined properties.
    public string Text { get; set; }
    public DateTime CreatedOn { get; set; }
    public int Rating { get; set; }
}
```

This allows us to query the following and display the blogs for a given year and month.

```
// for 10/2008, we will use <Channel name>_7991_03.  7991 = 9999 – 2008; 03 = 13 - 10
var blogs =
    from blog in context.CreateQuery<Blog>("Blogs")
    where blog.PartitionKey == "Football_7991_03"
     select blog;
foreach (Blog blog in blogs) { ... }
```

If all blogs for the year are desired, then a query such as the one below can be used.

```
var blogs =
    (from blog in context.CreateQuery<Blog>("Blogs")
    where blog.PartitionKey.CompareTo("Football_7991") <= 0
       && blog.PartitionKey.CompareTo("Football_7990") > 0
     select blog).Take(1000);

foreach (Blog blog in blogs) { ... }
```

NOTE: In all the code samples in the case study, we have left out continuation tokens for brevity. An actual application may have to execute multiple queries using the continuation token to get all the results.

8.3.2 Dynamically selecting the granularity of the PartitionKey

In the above example, it may be efficient to use a dynamic way of partitioning. Using the same example above, we could start with <Channel_Name>_<9999 - Year>_<13 - Month>. But during the anticipated

"hot traffic" months, we may choose to reduce the granularity of the partition size by using <Channel_Name>_<9999 - Year>_<13 - Month>_<5 - Week>. The application would need to be aware of this dynamic partitioning scheme. For example: If the following rule is used to partition blogs:

1. Between March to July i.e. off season use <Channel_Name>_<9999 - Year>_<13 - Month>
2. Rest of the months (i.e. during the NFL football season) use <Channel_Name>_<9999 - Year>_<13 - Month>_<5 - Week>

Now to retrieve all blogs blogged on 7/3/2008 UTC, the application can use the following query:

```
// long rowKeyStart = DateTime.MaxValue.Ticks - d1.Ticks  where d1 =  7/3/2008 00:00:00
// long rowKeyEnd = DateTime.MaxValue.Ticks  - d2.Ticks where d2 =  7/4/2008 00:00:00
var blogs =
    (from blog in context.CreateQuery<Blog>("Blogs")
    where blog.PartitionKey == "Football_7991_06"
       && blog.RowKey.CompareTo(rowKeyStart) >= 0 &&
       && blog.RowKey.CompareTo(rowKeyEnd) < 0
     select blog).Take(1000);

foreach (Blog blog in blogs) { }
```

However, to retrieve blogs created on 10/3/2008 UTC, the application can use the following query:

```
// long rowKeyStart = DateTime.MaxValue.Ticks - d1.Ticks  where d1 =  10/3/2008 00:00:00
// long rowKeyEnd = DateTime.MaxValue.Ticks  - d2.Ticks where d2 =  10/4/2008 00:00:00
var blogs =
    (from blog in context.CreateQuery<Blog>("Blogs")
    where blog.PartitionKey == "Football_7991_03_04"
       && blog.RowKey.CompareTo(rowKeyStart) >= 0 &&
       && blog.RowKey.CompareTo(rowKeyEnd) < 0
    select blog).Take(1000);

 foreach (Blog blog in blogs) { }
```

This query is more efficient with using the "week" in the partitioning scheme, since we reduced the scope to search blogs created only during the first week of a month when we anticipated heavy blogging.

8.3.3 Different Entity Kinds in the Same Table

In many applications, we come across data that represents different entities and yet needs to be retrieved together and processed together. Take for example, a social network site, where a user can have blogs, photos, etc. The user interface usually groups all this information together for a particular user. For example, user Joe's social network home page, would show the most recent changes by Joe (i.e., most recent blog posts, photos, videos, etc.). It has also been determined that since the home page shows the most recent changes, we want to optimize our queries to efficiently retrieve the most recent changes to the home page. Given these requirements, we can ensure that the data is clustered for efficient retrieval by having a single table that stores all the entities, along with a "Kind" for each entity, with all of the entities sorted from newest to oldest. We can partition the data according to user id and the RowKey can be used to differentiate the different entities (i.e. <Time (newest to oldest)>_<Entity Kind such as Photo, Blog, Movie, etc.>).

This also allows you to use entity group transactions to create/update/delete all related entities belonging to a single user using just a single transaction (See section 4.9 above).

Note: Alternatively, <Entity kind>_<Time (newest to oldest)> could be used as the RowKey if the main scenario is to retrieve entities by the kind it belongs to. But since we want to optimize the home page access for recent changes, we will have the Time precede the Kind in this example.

```
 // This union of all entities will be used only for querying. While inserting and updating
 // entities, we will use separate classes for each entity kind.
[DataServiceKey("PartitionKey", "RowKey")]
public class SocialNetworkEntity
{
    // User Id
    public string PartitionKey { get; set; }

    // To sort the time from newest to oldest
    // <DateTime.MaxValue.Ticks - DateTime.UtcNow.Ticks>_EntityType
    // The ticks can be at most 19 digits
    public string RowKey { get; set; }

    // User defined properties will be union of all entity properties.
     // Entity type repeated to allow easy search
    public string EntityType { get; set; }

     // Blog properties
    public string Text { get; set; }
    public string Category { get; set; }

     // Photo properties
    public string Uri { get; set; }
    public string ThumbnailUri { get; set; }
     // Common to blog and photo
    public string Caption { get; set; }

     // Common to blog and photo
    public DateTime CreatedOn { get; set; }
    public int Rating { get; set; }

     // This class can be responsible for returning the appropriate
     // kind so that business logic does not have to deal with
     // the union. The setter is private which prevents it from
     // being stored in the store.
     public BlogEntity Blog { get; private set }
     public PhotoEntity Photo { get; private set }
}
```

NOTE: The class with union of all properties across all entities clustered in one table is required only for querying in ADO.Net Services. If using REST, we do not need this since we have total control on serialization and deserialization. This means that when seeing the Kind on the REST result, we can then populate the correct type with the data fields returned by the REST request.

Now the following query can be used to retrieve the entities to be displayed on home page:

```
var entities =
     (from entity in
```

```
            context.CreateQuery<SocialNetworkEntity>("SocialNetworkTable")
        where entity.PartitionKey == userId
        select entity).Take(100);
```

And the following query can be used to retrieve only photos to be displayed in the photo page:

```
string startPhotoKey  = string.Format("{0:D19}_Photo", DateTime.MinValue.Ticks);
string endPhotoKey  = string.Format("{0:D19}_Photo", DateTime.MaxValue.Ticks);
var entities =
    (from entity in
         context.CreateQuery<SocialNetworkEntity>("SocialNetworkTable")
    where entity.PartitionKey == userId
     and RowKey.CompareTo(startPhotoKey) >= 0
     and RowKey.CompareTo(endPhotoKey) <= 0
    select entity).Take(100);
```

NOTE: While creating the RowKey, we ensure that the prefix is fixed length number. We use 19 digits to allow the entire range of long to be used.

If the goal of the application is to retrieve entities by their Kind, then we would want Kind to precede the Time in the RowKey. But in this example, the application's focus is to retrieve and display the last N changes to a web page, and the chosen RowKey is optimized for that.

8.3.3.1 Deserializing the Different Entity Kinds

One can also deserialize different entity types using Astoria's ReadingEntity event. In this example, let us assume that a single table stores blogs and comments. The PartitionKey for the table is "user name" and the RowKey is <Row key for the blog>_<CommentId>.

```
/// <summary>
/// Helper class used to maintain the data parsed out of the response payload.
/// </summary>
class GenericType
{
    public string ValueType { get; set; }
    public string Value { get; set; }
    public string PropertyName { get; set; }
    public bool IsNull { get; set; }
}

/// <summary>
/// Implements reading entity handler such that it reads the entities and stores all
/// blogs in a dictionary. Each blog holds all related comments. This class will
/// recreate the entities from the payload and store them for later access.
/// </summary>
class BlogReader
{
    private static readonly XNamespace AtomNamespace = "http://www.w3.org/2005/Atom";
    private static readonly XNamespace AstoriaDataNamespace =
                  "http://schemas.microsoft.com/ado/2007/08/dataservices";
    private static readonly XNamespace AstoriaMetadataNamespace =
                  "http://schemas.microsoft.com/ado/2007/08/dataservices/metadata";

    internal Dictionary<string, Blogs> Blogs = new Dictionary<string, Blogs>();
    internal BlogReader(DataServiceContext context)
    {
        context.ReadingEntity +=
                  new EventHandler<ReadingWritingEntityEventArgs>(OnReadingEntity);
        context.ResolveType = this.ResolveType;
```

```
}

public Type ResolveType(string name)
{
    return typeof(BaseEntity);
}

public void OnReadingEntity(object sender, ReadingWritingEntityEventArgs e)
{

   // Read all the properties from the payload for this entity and store them
   // using GenericType instances.
    var q = from p in e.Data.Element(AtomNamespace + "content")
                            .Element(AstoriaMetadataNamespace + "properties")
                            .Elements()
            select new GenericType()
            {
                PropertyName = p.Name.LocalName,
                IsNull = string.Equals("true",
                      p.Attribute(AstoriaMetadataNamespace + "null") == null ? null :
                             p.Attribute(AstoriaMetadataNamespace + "null").Value,
                             StringComparison.OrdinalIgnoreCase),
                      ValueType =
                             p.Attribute(AstoriaMetadataNamespace + "type") == null ?
                            null : p.Attribute(AstoriaMetadataNamespace + "type").Value,
                       Value = p.Value
            };

   // We have stored the type of the entity in a property called "EntityType".
   // We shall read it to create appropriate entity types and store it.
    string type = (from s in  q.AsQueryable()
                   where s.PropertyName == "EntityType"
                   select s.Value.ToString()).FirstOrDefault();

    if (string.Equals(type, "Blogs"))
    {
        Blogs blog = ReadBlog(e.Entity as BaseEntity, q);
        this.Blogs.Add(blog.RowKey, blog);
    }
    else
    {
        // Since the comment stores the blog id key in its own key, we shall parse
        // it out to recreate the relationship. The newly read comment is stored
        // along with the blog which should have been read since key of blog < key
        // of comment.
        Comments comment = ReadComment(e.Entity as BaseEntity, q);
        string[] tokens = comment.RowKey.Split('_');
        this.Blogs[tokens[0]].Comments.Add(comment);
    }
}

private Blogs ReadBlog(BaseEntity entity, IEnumerable<GenericType> blogProperties)
{
    Blogs blog = new Blogs();
    blog.PartitionKey = entity.PartitionKey;
    blog.RowKey = entity.RowKey;
    blog.Timestamp = entity.Timestamp;

    // Either go through each property or use Linq to extract the properties that
    // we are interested in.
    foreach (GenericType t in blogProperties)
    {
        if (string.Equals(t.PropertyName, "Message"))
        {
            blog.Message = GetTypedEdmValue(t.ValueType, t.Value, t.IsNull).ToString();
        }
    }

    return blog;
}
```

```
        private Comments ReadComment(BaseEntity entity, IEnumerable<GenericType> properties)
        {
            Comments comment = new Comments();
            comment.PartitionKey = entity.PartitionKey;
            comment.RowKey = entity.RowKey;
            comment.Timestamp = entity.Timestamp;

            // Either go through each property or use Linq to extract the properties that
            // we are interested in.
            foreach (GenericType t in properties)
            {
                if (string.Equals(t.PropertyName, "Comment"))
                {
                    comment.Comment = GetTypedEdmValue(t.ValueType, t.Value, t.IsNull).ToString();
                }
                else if (string.Equals(t.PropertyName, "Rating"))
                {
                    comment.Rating = (int)GetTypedEdmValue(t.ValueType, t.Value, t.IsNull);
                }
            }

            return comment;
        }

        private static object GetTypedEdmValue(string type, string value, bool isnull)
        {
            // Depending on the type, create the value
            if (isnull) return null;

            if (string.IsNullOrEmpty(type)) return value;

            switch (type)
            {
                case "Edm.String": return value;
                case "Edm.Byte": return Convert.ChangeType(value, typeof(byte));
                case "Edm.SByte": return Convert.ChangeType(value, typeof(sbyte));
                case "Edm.Int16": return Convert.ChangeType(value, typeof(short));
                case "Edm.Int32": return Convert.ChangeType(value, typeof(int));
                case "Edm.Int64": return Convert.ChangeType(value, typeof(long));
                case "Edm.Double": return Convert.ChangeType(value, typeof(double));
                case "Edm.Single": return Convert.ChangeType(value, typeof(float));
                case "Edm.Boolean": return Convert.ChangeType(value, typeof(bool));
                case "Edm.Decimal": return Convert.ChangeType(value, typeof(decimal));
                case "Edm.DateTime": return XmlConvert.ToDateTime(value,
XmlDateTimeSerializationMode.RoundtripKind);
                case "Edm.Binary": return Convert.FromBase64String(value);
                case "Edm.Guid": return new Guid(value);

                default: throw new NotSupportedException("Not supported type " + type);
            }
        }
    }
```

Given the above classes, we can then read the entities that belong to different types using the following code:

```
        DataServiceQuery<BaseEntity> query = ((
               from a in context.CreateQuery<BaseEntity>(DataContext.TableName)
                where a.PartitionKey == user
                select a)) as DataServiceQuery<BaseEntity>;

       // force the execution of the query so that BlogReader's OnReadingEntity event is invoked
       BlogReader reader = new BlogReader(context);
        query.Execute().ToList();

       // access the blogs stored in BlogReader
        foreach (Blogs blog in reader.Blogs.Values)
        {
```

```
        Console.WriteLine("{0}", blog);
        foreach (Comments comment in blog.Comments)
        {
          Console.WriteLine("\t{0}", comment);
        }
}
```

8.4 Upgrade and Versioning

Every application goes through scenarios where its current schema needs an upgrade to handle new requirements.

It is important to note that Windows Azure Table does not store a schema; it only stores <name, typed-value> pairs for each property in an entity. Therefore, you can have two entities in the same table with different properties. You can also have two entities in the same table with the same property name, but with different types. For example, one could have an entity A with a property "Rating" of type Int, and another entity B with a property "Rating" of type Double. When performing a query, the server ignores entities if the property type in the query condition does not match the property type for that entity. Using the above example, if we queried for "Rating > 1.2", then this would evaluate against entity B, but entity A would be skipped, since there is a type mismatch.

Now when upgrading a property it is imperative for an application to handle both versions until all the clients and existing data in the store are upgraded. The rest of this section concentrates on the usefulness of maintaining a version property on each entity which allows applications to handle different kinds of upgrades and how to use ADO.NET Service to accomplish an upgrade.

In all of the upgrade scenarios defined below, there are two possibilities for applying the upgrade:

1. Upgrade on access – i.e. whenever an entity is retrieved for update, upgrade it to the newer version.
2. Upgrade in background – upgrade is done in the background using a worker role where the worker retrieves all entities that belong to the older version (using the version column) and updates it.

During read operations, as stated earlier, an application will need to handle both the versions. Before we get into the details of adding or deleting properties, we will go over an important DataServiceContext boolean property called IgnoreMissingProperties. IgnoreMissingProperties controls whether ADO.NET Data Service throws an exception when the client side entity class does not define a property that was returned by the server. For example: If the Server returns a property called "Rating" on a Blog entity, but "Rating" is not defined in the client's class definition for Blog, then ADO.NET Data Service throws an exception only if IgnoreMissingProperties is false.

```
// Set this to allow client definition of an entity to have certain properties  missing
because of either
// an addition/deletion of property on entities stored on the server.
context.IgnoreMissingProperties = true;
```

8.4.1 Adding a new property

When an application adds a new property, it will need to define the property in the entity definition too. Since older versions of an application may also be running, it is a good idea to always set IgnoreMissingProperties on the DataServiceContext to true as shown above. This allows older clients without the newer properties to still read those entities. Without setting it, ADO.NET Data Service will throw an exception when an entity does not define a property that the returning REST packet contains.

8.4.2 Deleting a property type

We will explain property deletion via an example where we delete the "Rating" property from blog entities.

1. We will create a new definition for the Blog entity in which we will not define "Rating".

```
[DataServiceKey("PartitionKey", "RowKey")]
public class BlogV2
{
    // <Channel name>_<9999-YYYY>_<13-Month>
    public string PartitionKey { get; set; }

    // To sort RowKey from newest to oldest blog,
    // RowKey is DateTime.MaxValue.Ticks - DateTime.UtcNow.Ticks
    public string RowKey { get; set; }

    // User defined properties.
    public string Text { get; set; }
    public DateTime CreatedOn { get; set; }

     // NOTE: removing this property so that it can be deleted from the
    // storage too.
    // public int Rating { get; set; }
}
```

2. Now, to delete the property, we will retrieve the blog entities from the server and update it using the new entity definition:

```
// Ensure that an exception is not thrown since Rating is not defined in BlogV2 (as
// explained in section 8.4 above)
context.IgnoreMissingProperties = true;

// Retrieve blog entities into BlogV2.
// NOTE: Continuation tokens is not used here for brevity
var blogs =
    (from blog in context.CreateQuery<BlogV2>("Blogs")
    select blog);
foreach(BlogV2 blog in blogs)
{
   context.UpdateObject(blog);
}
```

3. We will invoke SaveChanges with SaveChangesOptions.ReplaceOnUpdate so that any missing properties will be deleted on the server.

```
// SaveChangesOptions.ReplaceOnUpdate will result in a "PUT" request being sent to the
// server as against "MERGE" request which will ensure that any missing property is deleted.
```

```
context.SaveChanges(SaveChangesOptions.ReplaceOnUpdate);
```

8.4.3 Modifying a property type

When the type of a property needs to be changed for a schema, one option is to create a new property, with a new name with the desired type, then copy the data from the old property to the new property and the finally delete the old property. To accomplish this employ the techniques described above in 8.4.1 and 8.4.2.

9 Windows Azure Table Best Practices

In this section we will cover various best practices that we recommend. These best practices aim at making your application robust, scalable and flexible.

9.1 Table Creation

Applications should separate table creation logic from the main business logic. For example, consider applications attempting to create a table before each entity insert, and relying on "Table already exists" exceptions being sent back if the table is already present. This results in a lot of unnecessary transactions to the store, which can accumulate in transaction costs for the application, once a billing model is in place.

We recommend that the table creation logic be executed, as a separate script when first setting up a service, when a role starts up, or when "TableNotFound" error is returned with HTTP Status code of 404. Table creation logic should not rely on "Table already exists" exception but instead it should retrieve the table before it tries to create it.

Note that, when a table is deleted, the same table name cannot be recreated for at least 30 seconds, while the table is being garbage collected.

9.2 Asynchronous version of ADO.NET Data Services API

Asynchronous I/O APIs allows an application to achieve better CPU utilization rather than being blocked waiting for a response from the server. ADO.NET Data Services provides an asynchronous version for CRUD operations. For improved performance, it is recommended that where ever possible, an application should use this asynchronous version rather than the synchronous version. For example, use the DataServiceContext's BeginExecute method for executing a query.

9.3 DataServiceContext settings

- When SaveChanges fails for update/delete/add, the change is still tracked and subsequent SaveChanges will resend the request. We recommend either using a new DataServiceContext or explicitly detaching the entity for which the operation failed.
- It is good to always set IgnoreMissingProperties to true on the data context to allow entities to be backwards compatible.

9.4 Partitioning scheme

It is recommended that a partitioning scheme is chosen for performance, scalability and future extensibility. See section 3 and 8.3 for more information.

9.5 Unconditional Updates and Deletes

ETags can be viewed as a version for entities. These can be used for concurrency checks using the If-Match header during updates/deletes. Astoria maintains this etag which is sent with every entity entry in the payload. To get into more details, Astoria tracks entities in the context via context.Entities which is a collection of EntityDescriptors. EntityDescriptor has an "Etag" property that Astoria maintains. On every update/delete the ETag is sent to the server. Astoria by default sends the mandatory "If-Match" header with this etag value. On the server side, Windows Azure table ensures that the etag sent in the If-Match header matches our Timestamp property in the data store. If it matches, the server goes ahead and performs the update/delete; otherwise the server returns a status code of 412 i.e. Precondition failed, indicating that someone else may have modified the entity being updated/deleted. If a client sends "*" in the "If-Match" header, it tells the server that an unconditional update/delete needs to be performed i.e. go ahead and perform the requested operation irrespective of whether someone has changed the entity in the store. A client can send unconditional updates/deletes using the following code:

```
context.AttachTo("TableName", entity, "*");
context.UpdateObject(entity);
```

However, if this entity is already being tracked, client will be required to detach the entity before attaching it:

```
context.Detach(entity);
```

9.6 Handling Errors

When designing an application for use with Windows Azure Table, it is important to handle errors appropriately. This section describes issues to consider when designing your application.

9.6.1 Network errors and timeouts on successful server side operations

When a network error, internal server error or timeout occurs on an operation, the operation may have still succeeded on the server. It is recommended to retry operations and expect that they may fail due to the prior request completing successfully. For example, if deleting an entity fails, try deleting the entity again and expect “not found” in case the previous call had succeeded. Another example is when adding an entity, if the first request times out the entity may still have been inserted into the table. In this case, the application needs to be able to deal with getting an “entity already exists” error back when retrying the operation.

9.6.2 Retry Timeouts and “Connection closed by Host” errors

Requests that receive a Timeout or “Connection closed by Host” response might not have been processed by Windows Azure Table. For example, if a PUT request returns a timeout, a subsequent GET

might retrieve the old value or the updated value. If you see repeated timeouts, retry the request again with exponential back-off to avoid further overloading the system.

9.6.3 Conflicts in Updates

As explained in section 5, application should handle "Precondition Failed" by retrieving the latest version of the entity and issuing a subsequent update.

9.6.4 Tune Application for Repeated Timeout errors

Timeout errors can occur if there are network issues between your application and the data center. Over the wide area network, it is recommended to break a single large operation into a series of smaller calls, and design your application to handle timeouts/failures in this case so that it is able to resume after an error and continue to make progress. Tune the size limit(s) of each request based on the network link you have.

We designed the system to scale and be able to handle a large amount of traffic. However, an extremely high rate of requests may lead to request timeouts, while the system load balances. In that case, reducing your request rate may decrease or eliminate errors of this type. In general, most users will not experience these errors regularly; however, if you are experiencing high or unexpected Timeout errors, contact us at the MSDN forums to discuss how to optimize your use of Windows Azure Table and prevent these types of errors.

9.6.5 Error handling and reporting

The REST API is designed to look like a standard HTTP server interacting with existing HTTP clients (e.g., browsers, HTTP client libraries, proxies, caches, and so on). To ensure the HTTP clients handle errors properly, we map each Windows Azure Table error to an HTTP status code.

HTTP status codes are less expressive than Windows Azure Table error codes and contain less information about the error. Although the HTTP status codes contain less information about the error, clients that understand HTTP will usually handle the error correctly.

Therefore, when handling errors or reporting Windows Azure Table errors to end users, use the Windows Azure Table error code along with the HTTP status code as it contains more information about the error. Additionally, when debugging your application, you should also consult the human readable <ExceptionDetails> element of the XML error response.

Each response from Windows Azure Table has HTTP headers of the form below.

```
HTTP/1.1 204 No Content
Content-Length: 0
ETag: W/"datetime'2008-10-01T15%3A27%3A34.4838174Z'"
x-ms-request-id: 7c1b5e22-831d-403c-b88a-caa4443e75cb
```

If you suspect a server issue related to your query, you can contact Windows Azure Storage at the MSDN forums with the x-ms-request-id above. This will help us diagnose the issue.

9.7 Tuning .NET and ADO.NET Performance

We have collected the common issues that users have come across while using Windows Azure Table and posted some solutions. Some of these are .NET related or ADO.NET Data Services (aka Astoria) related.

9.7.1 Improve Performance of ADO.NET Data Service Deserialization.

When you execute a query using ADO .Net data services, there are two important names – the name of the CLR class for the entity, and the name of the table in Windows Azure Table. We have noticed that when these names are different, there is a fixed overhead of approximately 8-15ms for deserializing each entity received in a query. See section 4.10 for more details.

9.7.2 Default .NET HTTP Connections set to 2

This is a notorious issue that has affected many developers. By default, the value for the number of .NET HTTP connections is 2. This implies that only 2 concurrent connections can be maintained. This manifests itself as "underlying connection was closed..." when the number of concurrent requests is greater than 2. The default can be increased by setting the following in the application configuration file OR in code.

Config File:

```
<system.net>
  <connectionManagement>
        <add address = "*" maxconnection = "48" />
  </connectionManagement>
 </system.net>
```

In Code:

```
ServicePointManager.DefaultConnectionLimit = 48;
```

The exact number depends on your application. http://support.microsoft.com/kb/821268 has good information on how to set this for server side applications. You can also set it for a particular URI by specifying the URI in place of "*". If you are setting it in code, you could use the ServicePoint class rather than the ServicePointManager class i.e.:

```
ServicePoint myServicePoint = ServicePointManager.FindServicePoint(myServiceUri);
myServicePoint.ConnectionLimit = 48;
```

9.7.3 Turn off 100-continue

What is 100-continue? When a client sends a POST/PUT request, it can delay sending the payload by sending an “Expect: 100-continue” header.

1. The server will use the URI plus headers to ensure that the call can be made.
2. The server would then send back a response with status code 100 (Continue) to the client.
3. The client would send the rest of the payload.

This allows the client to be notified of most errors without incurring the cost of sending that entire payload. However, once the entire payload is received on the server end, other errors may still occur. When using .NET library, HttpWebRequest by default sends "Expect: 100-Continue" for all PUT/POST requests (even though MSDN suggests that it does so only for POSTs).

In Windows Azure Tables/Blobs/Queue, authentication, unsupported verbs and missing headers failures can be tested just by receiving the headers and the URI. . If Windows Azure clients have tested the client well enough to ensure that it is not sending any bad requests, clients could turn off 100-continue so that the entire request is sent in one roundtrip. This is especially true when clients send small payloads as in the table or queue service. This setting can be turned off in code or via a configuration setting.

In Code:

```
// set it on service point if only a particular service needs to be disabled.
ServicePointManager.Expect100Continue = false;
```

Config file:

```
<system.net>
    <settings>
      <servicePointManager expect100Continue="false" />
    </settings>
</system.net>
```

Before turning 100-continue off, we recommend that you profile your application examining the effects with and without it.

9.7.4 Turning off Nagle may help Inserts/Updates

We have seen that turning nagle off has provided significant boost to latencies for inserts and updates in the table service. This is usually true if you have a large number of small entities. However, turning Nagle off is known to adversely affect throughput and hence your application should be tested to see if it makes a positive difference.

This can be turned off either in the configuration file or in code as below.

In Code:

```
ServicePointManager.UseNagleAlgorithm = false;
```

Config file:

```
<system.net>
    <settings>
      <servicePointManager expect100Continue="false" useNagleAlgorithm="false"/>
    </settings>
</system.net>
```

9.8 Deleting and then Recreating the Same Table Name

During development phase, it may be required to delete tables regularly. Currently, it takes at least 40 seconds before a deleted table can be recreated again. It may be beneficial to design the application

such that the table names are suffixed with an id or version each time. This would circumvent the need to wait before a table can be recreated after a delete. However, this requires that ResolveType be set to overcome the performance bug in ADO.NET Data Service client library which affects query scenarios where the table name is not the same as the entity class name.

10 Summary

Windows Azure Table provides a structured storage platform. It supports massively scalable tables in the cloud. The system efficiently scales your tables by automatically load balancing partitions to different servers as traffic grows. It supports a rich set of data types for properties and can be accessed via ADO.NET Data Services and REST. Compile time type checking is provided via ADO.NET Data Services which allows you to use Windows Azure Table just as you would use a structured table. This includes supporting features like pagination, optimistic concurrency and continuation for long running queries. The current limits on properties and query execution time allow us to provide a massively scalable table storage system while allowing users to build compelling applications.