Jeffrey Richter's Guide to Working with Azure Storage Tables via C# and other .NET Languages

August 16, 2014

This document has been reviewed by Jai Haridas and Vamshi Kommineni of Microsoft's Azure Storage team.

About this Guide

I have been with working with Azure Storage Tables (hereinafter referred to as *Tables*) for many years now. You perform operations on Tables by calling various http(s) REST APIs. However, to make programmers' lives simpler, Microsoft provides numerous client libraries (for .NET, Java, C++, Node.js, PHP, and Python) that put a more programmer-friendly face on top of this REST API. Since I primarily program in C#, I learned how to work with Tables using Microsoft's .NET storage client library. This library is not just a simple wrapper around the REST APIs, it adds additional features and concepts over and above what the REST API offers.

At first, this might sound good as it can make developers more productive. However, over time, I found the .NET storage client library was getting in my way and was actually hurting me. You see, I formed my mental model of Tables and how they work based on the library's abstractions and additional concepts. Then, for the projects I was working on, I made architectural decisions based on this flawed mental model. Over time, I improved my mental model of Tables and how to best leverage the features they offer. And now, I know which parts of the library I should use and which parts of the library I should avoid. I then created my own library (built on top of the good parts of Microsoft's library) to simplify everyday Table tasks.

This guide has many purposes:

- Help developers improve their mental model with respect to Tables.
- Point out the good and bad parts of Microsoft's Azure storage library.
- Introduce a set of patterns and practices related to Tables that have served me very well over the years.
- Introduce my (free) Azure storage library and how it adheres to a better Table mental model while increasing programmer productivity. My library is called the *Wintellect Azure Storage Library* and it can be easily added to your own Visual Studio projects via the NuGet package manager. The library can be found here: https://www.nuget.org/packages/Wintellect.Azure.Storage/. Wintellect (http://Wintellect.com/) has been using this library for years to assist migrating consulting customers to Microsoft Azure. In addition, this library is used very heavily with Wintellect's own on-demand training service: http://WintellectNOW.com/.

This paper is not meant as a tutorial for working with Tables. You will get the most out of this paper if you already understand the basics of Tables and non-relational (NoSQL) databases.

Tables versus Blobs

A Table holds a collection of entities. Each entity is uniquely identified via its PartitionKey and RowKey. Associated with this entity, you can define up to 252 properties and values that must total less than 1MB. But, instead of using a Table, you could

create a blob container and store each entity in a blob. You'd combine the PartitionKey and RowKey values to create the blob's unique name and then you could store whatever you want (after converting it to a byte array) inside the blob. A block blob can hold up to 200GB of data (far more than the 1MB limit of each Table entity) and using a blob removes the 252 property limit as well.

Furthermore, blob data is cheaper to store and is also cheaper to transfer over the wire because blob data is sent over the wire as bytes while Table entities are sent over the wire as UTF-8 encoded strings. In addition, it is easier to move blobs from Azure to another storage service (like Amazon S3) or to a hard disk because blobs are very similar to files. It can be much harder moving Table data (and changing code that accesses it) from Azure to another storage service.

So, if blobs have much higher limits, are cheaper, and are less Azure-specific, why store entities in a Table instead of in blobs? There are three answers to this (listed from most useful to least useful [in my experience]):

1. **Filtering.** You can execute a query against a Table's entities' properties to filter out entities matching a query.[1]
2. **Operate on Portions of an Entity.** With Tables you can easily retrieve some properties related to an entity and you can also easily modify the properties you desire. You do not have to work with all of an entities' properties all the time.
3. **Entity Group Transactions (EGT).** With an Entity Group Transaction, you can atomically update multiple entities in the same Table partition.

If your data needs do not require filtering, acting on portions of data or manipulating multiple objects via a transaction, then I recommend you use blobs instead of Table entities.

Understanding Table Properties

If you need to perform filtered queries, then you need to really understand how an entity's properties work. Let's start with the pure and simple facts:

- An entity is a collection of up to 255 property tuples.
- Each property tuple consists of a name, data type, and value:
  - The name is ≤ 25 characters & is case sensitive.
  - The data type is one of the following scalar or array types[2]:
    - Scalars: **Boolean**, **Int32**, **Int64**, **Double**, **DateTimeOffset**, **Guid**
    - Arrays: **Byte[]** (≤ 65536 bytes) or a **String** (≤ 32767 UTF-16 characters)
  - The data value is a non-**null** value.
- Three properties are mandatory: **PartitionKey** (a **String**), **RowKey** (a **String**), and **Timstamp** (a **DateTimeOffset**). You can define up to an additional 252 properties.
- The total number of bytes occupied by all of an entity's tuples must be ≤1MB and is calculated like this[3]:

Property	# of Bytes
PartitionKey	2 * PartitionKey.Length

[1] Within each Table partition, the Table service can scan about 2,000 1KB entities per second.

[2] If you need to store a value that is not one of the Table-supported types, you can split the object's states across multiple properties (for example, store a **Point** using an **Int32** X property and an **Int32** Y property), serialize the **Point**'s X & Y values into a byte array which you'll have to deserialize later, for format the X & Y values into some string which you'll have to parse later. Of course, whichever you choose impacts your ability to filter on the value when performing a query.

[3] See the section at the end of this blog post: http://blogs.msdn.com/b/windowsazurestorage/archive/2010/07/09/understanding-windows-azure-storage-billing-bandwidth-transactions-and-capacity.aspx

RowKey	2 * RowKey.Length
Timestamp	4
Additional Properties	8 + (2 * PropertyName.Length) + sizeof(PropertyValue)[4]

- To reduce monthly storage costs and to reduce bytes going across the wire (thereby increasing performance and reducing bandwidth costs), you can use sensible short property names and store the property value as compactly as possible.

For the scalar property data types, the type indicates to the Table how many bytes you are billed for (a **Boolean** is 1 byte, an **Int32** is 4 bytes, an **Int64** is 8 bytes, and so on). In addition, Tables use the type to determine how to interpret the value when performing a filter query comparison. For example, if the Table knows that a "Birthday" property is a **DateTimeOffset**, then you can find all people born during 2014:

$filter=Birthday **ge** datetime'2014-01-01T00:00:00.0000000Z' and Birthday **lt** datetime'2015-01-01T00:00:00.0000000Z'

The table below shows all the supported property data types, the number of bytes each requires, and the comparison operators that make sense for each data type:

Property Data Type	# of Bytes	Comparison Operators
Boolean	1	eq, ne
Int32	4	eq,,ne, gt, ge, lt, le
Int64	8	eq,,ne, gt, ge, lt, le
Double	8	eq,,ne, gt, ge, lt, le
DateTimeOffset	8	eq,,ne, gt, ge, lt, le
Guid	16	eq,, ne
String	4 + (2 * # of UTF-16 code points)	eq,,ne, gt, ge, lt, le
Binary	4 + (# of bytes)	eq,,ne, gt, ge, lt, le

.NET developers are able to perform many operations on .NET strings including case-insensitive comparisons, culture-sensitive comparison, **StartsWith**, **EndsWith**, **Contains**, and more. However, Table string properties support a severe subset of these operations. **Specifically, String properties support only case-sensitive, ordinal UTF-16 code point comparisons.** However, you can accomplish the equivalent of **StartsWith** using the **ge** and **lt** comparison operators. Here's a filter that returns entities whose "Name" property starts with "Jeff" (remember the comparison is always case sensitive):

$filter=Name ge 'Jeff'' and Name lt 'Jefg'

Microsoft's .NET Azure Storage Library's Table Support

Microsoft's .NET Azure Storage library defines a **Microsoft.WindowsAzure.Storage.Table** namespace. Here are the main classes defined in this namespace:

- **CloudTableClient.** This class exposes operations that affect all Tables within a storage account, such as managing service properties, service statistics, and listing Tables. From an instance of this class you can call **GetTableReference** to get a **CloudTable**.
- **CloudTable.** This class exposes operations that affect a specific Table. With an instance of this class, you can create, delete, set/get permissions, and get a shared access signature for a Table. With an instance of this class, you can also perform operations that change entities in the Table, such as inserting/merging/replacing/deleting an entity or performing a batch of these operations (an EGT). And, of course, you can use a **CloudTable** object to querying a Table's entities.

[4] The size of each property type is shown in the next table.

- **DynamicTableEntity.** You construct an instance of this class to create and initialize an entity's properties in memory and then serialize it to JSON and send it over the wire to insert, merge, replace, or delete this entity on the Table. When you query the Table, each returned entity is deserialized into a **DynamicTableEntity** object so you can examine its properties in your code.
- **EntityProperty.** Each **DynamicTableEntity** object has a **Properties** property which is a dictionary where the keys are strings (the name of the property) and the value is an **EntityProperty** object. Each **EntityProperty** object indicates the type of the property and its value.

For the most part, I'm happy with the Table support offered by Microsoft's .NET Azure Storage library. However, there are also many issues I've run into and I describe them in the following subsections. In the next major section, I'll talk about Wintellect's Azure Storage client library and how it addresses many of the issues I describe below.

Methods that Perform I/O Operations

Over the years, .NET has defined three different programming models for methods that perform I/O operations. The classic synchronous programming, the **Begin*Xxx*/End*Xxx*** asynchronous programming model, and the relatively new ***Xxx*Async** asynchronous programming model. Due to this history (and backward compatibility), Microsoft's Azure Storage client library also supports these three programing models. However, the trend within (and outside of) Microsoft is to only support the new ***Xxx*Async** methods. For example, ***all*** Windows Runtime (WinRT) types offer the ***Xxx*Async** methods only. And many newer .NET types (like System.Net.Http.HttpClient) only expose the newer ***Xxx*Async** methods as well.[5]

There are two main reasons to use Azure Tables over many other database technologies: cost and scalability. **You should avoid the synchronous methods because they can hurt your app's scalability dramatically.** In addition, your Azure subscription is charged for each I/O operation made against a Table. When you call an ***Xxx*Async** method to access the Azure storage service, you know that you are being charged for the operation. Calling ***Xxx*Async** methods in your code make it clear to you when code is incurring costs and I/O operations (which have unpredictable performance).

In my own projects, I always use the ***Xxx*Async** programming model and I recommend you do the same. For more information about the ***Xxx*Async** methods, please see Chapter 28 of my book, CLR via C#, 4th Edition.

Converting Table Entities to .NET Classes

The `CloudTable` class offers methods that affect a Table's entities. There are methods that operate on `DynamicTableEntity` objects and, for convenience, there are generic methods that operate on .NET classes. I recommend avoiding all the generic methods that operate on .NET classes for two reasons: efficiency and it prevents you from taking advantage of the great features provided by a NoSQL database.

Let me address efficiency first. Internally, these generic methods perform runtime reflection and runtime code compilation which are slow and memory consuming technologies. With newer versions of .NET, there are more efficient ways of accomplishing the same goals. Specifically, I'm talking about .NET's [CallerMemberName] attribute which was introduced with .NET 4.5 but can be used with .NET 4.0 if you install the Microsoft BCL Portability Pack NuGet package. Later in this document, I'll show how I use this when working with Tables.

A truly great feature of NoSQL databases is that entities within a single Table can all have different sets of properties. This allows properties to be added or removed from specific entities as needed enabling versioning overtime without your

[5] Note that the **Begin*Xxx*/End*Xxx*** asynchronous programming model is actually more efficient (uses less memory and is a bit faster) than the ***Xxx*Async** programming model. But, writing code using the **Begin*Xxx*/End*Xxx*** model is substantially more difficult. So, the **Begin*Xxx*/End*Xxx*** model should only be used in scenarios where you are extremely performance sensitive.

customers ever experiencing any service downtime. I have also leveraged this to store different properties based on the kind of entity. For example, I might have a Table of users and how they pay for services. In C#, I'd define my classes like this:

```
abstract class User                    { /* Define common User related properties here */ }
sealed   class CreditCardUser    : User { /* Define credit card related properties here */ }
sealed   class DirectDebitUser   : User { /* Define direct debit related properties here */ }
sealed   class PurchaseOrderUser : User { /* Define purchase order related properties here */ }
```

Every user entity has some common information such as the user's name and address. But, for users that pay via a credit card, I need to store credit card information and for users that pay via direct debit, I need checking account information, and for users that pay via a purchase order I need purchase order information. To accomplish this each entity has the common properties and then the additional properties required for the specific type of payment associated with the user. Azure Tables are awesome at handling this kind of thing. But, not if you use the client library's generic methods because you have to know the .NET type ***before*** you know the kind of entity you retrieved from the table.

In essence, the problem is that, after compilation, a .NET class always has the same set of properties. If an entity has a property that is missing in the .NET class, then your code can't access this property and, if you replace this entity on the Table, you end up deleting this data. On the other hand, if an entity is missing a property that the .NET class has, then your code manipulates non-existent data potentially causing undesired behavior.

There are some other problems too. The generic methods work only with types having properties supported by Azure tables. For example, I frequently want to use enum values in my .NET code but Tables don't support enums. If I call a generic method passing a type having an enum property, the client library ignores the property entirely: it won't save it to or load it from the Table. You can "work around" this by having the property use an **Int32** data type instead but then the rest of your code must constantly cast between the **Int32** property and the desired enum type. I find this very tedious and error prone.

Here's another problem: I have occasionally needed to change a property's type. For example, I once defined a property to be a **Boolean** and then, a year later, I wished I had made the property be an **Int32**. One of the great things about Tables is that old entities could have the property as a **Boolean** and new entities could have a property with the same name as an **Int32**. Using the client library's generic methods forbids you from easily dealing with this because, if you change the type's property from **Boolean** to **Int32** in your code, the library will try to parse an entity whose property is **Boolean** (**true**/**false**) to an **Int32** value which will fail.

For all these reasons, I strongly prefer using **`CloudTable`**'s methods that use the **`DynamicTableEntity`** class and I avoid using the generic methods. The **`DynamicTableEntity`** class looks like this:

```
public sealed class DynamicTableEntity : ITableEntity {
   // In my code, I use only the members shown here; I avoid all other members
   public DynamicTableEntity();
   public DynamicTableEntity(String partitionKey, String rowKey);
   public String ETag { get; set; }

   public String PartitionKey { get; set; }
   public String RowKey { get; set; }
   public DateTimeOffset Timestamp { get; set; }

   public IDictionary<String, EntityProperty> Properties { get; set; }
   public EntityProperty this[String key] { get; set; }  // Shorter syntax to access Properties
}
```

As you can see, this class contains a **`Properties`** property referring to a dictionary containing the entity's actual properties. You can dynamically add and remove properties from this dictionary exactly how you can dynamically add and remove

properties from a Table entity. And the **EntityProperty** class defines a **PropertyType** property that can tell you the property's type as read from the Table. You can use this to figure out how to interpret the property's value.

Properties with null Values

As just discussed, a **DynamicTableEntity** object has a **Properties** property referring to a collection of the property tuples you want as part of an entity. Each property has a **String** name and an **EntityProperty** object. The **EntityProperty** object contains the property's data type and value. The **EntityProperty** class looks like this (some members not shown):

```
public sealed class EntityProperty {
   // Constructors to create scalars (shown first) and arrays (shown next):
   public EntityProperty(Boolean? input);
   public EntityProperty(Int32? input);
   public EntityProperty(Int64? input);
   public EntityProperty(Double? input);
   public EntityProperty(DateTimeOffset? input);
   public EntityProperty(Guid? input);
   public EntityProperty(Byte[] input);
   public EntityProperty(String input);

   // Property that returns a property's data type
   public EdmType PropertyType { get; }

   // Properties that return scalars (shown first) and arrays (shown next):
   public Boolean?        BooleanValue        { get; set; }
   public Int32?          Int32Value          { get; set; }
   public Int64?          Int64Value          { get; set; }
   public Double?         DoubleValue         { get; set; }
   public DateTimeOffset? DateTimeOffsetValue { get; set; }
   public Guid?           GuidValue           { get; set; }
   public String          StringValue         { get; set; }
   public Byte[]          BinaryValue         { get; set; }
}
```

You'll immediately notice something very odd when looking at **EntityProperty**'s members: the members that work on scalar types accept and return nullable types. However, Tables do not support properties having a **null** value so you will never get back **null**.

> ***If you send an HTTP payload with a property whose value is null, the Table service ignores that property. That is, inserting or replacing a null property, does not associate the property with the entity. And, more importantly, merging a null property with an existing entity does not change the property's value.***

For this reason, you should never set a property's value to **null** and you will never receive **null** when querying a property's value. Because of this, **EntityProperty**'s method signatures break the mental model you should have when working with Tables. They should not expose nullable scalar types. Furthermore, the methods that accept **String** and **Byte[]** accept **null** and try to send it over the wire (with no affect); these methods should throw an **ArgumentNullException** instead.

There is one caveat to this. If you query a Table using the $select query option specifying a property that the entity does not have, the Table service returns the property over the wire with a **null** value. In my opinion, this is a bad design choice: since the entity does not have the property, the property should not be returned. Instead, the service returns the property with a **null** value but properties cannot have **null** values. So, this design choice breaks the correct mental model you should have of Tables.

Here's what I do: When I perform a query using the $select query option, I take the returned **DynamicTableEntity** object and call a helper method (see my **DynamicTableEntityEntensions**' **RemoveNullProperties** method) that walks through its dictionary, removing any properties with a **null** value. This prevents my code from seeing and manipulating properties that don't actually exist.

Creating Table Query Strings

Tables have extremely limited query support performed by making an http(s) REST API request. The URI's query string indicates the filter you want applied against the Table's entities and the service returns the subset of entities matching the query. Here is an example of a URI with a fairly complex filter query string:

```
http://.../TableName()?$filter=
       (PartitionKey ge 'Jeff') and (PartitionKey lt 'Jefg')
   and (ABoolean eq true)
   and (AByteArray eq X'010203')
   and (ADate gt datetime'2000-01-01T00:00:00.0000000Z')
   and (ADouble lt 1.23)
   and (AGuid eq guid'3236abc4-fd12-4773-9859-bb4e46d4e00c')
   and (AnInt32 ge 123)
   and (AnInt64 ne 321)
```

Here are some things you should know about filter strings (some of this was discussed earlier in this document):

- They support two combining operators: **and**, **or**.
- They support six comparison operators: **eq**, **ne**, **ge**, **gt**, **le**, **lt**.
- Property names and constants (like **true**, **false**, **datetime**, **guid**) must be lowercase.
- When comparing a **Boolean** property with a constant (**true**/**false**), you can use **eq**, **ne**.
- When comparing a **Guid** property with a constant, you can use **eq**, **ne**.
- When comparing a **Byte[]** property with a constant, you typically use **eq**, **ne**. However, you can also use **gt**, **ge**, **lt**, **le** to perform prefix matching.
- When comparing an **Int32**, **Int64**, **Double**, **DateTimeOffset**, or **String** property with a constant, you can use **eq**, **ne**, **ge**, **gt**, **le**, **lt**.
- **String** comparisons are always by UTF-16 code points. There is no way to do a case-insensitive or culture-sensitive comparison. Strings do not support operations like **EndsWith** or **Contains**; however, you can do a case-sensitive prefix match using **ge** and **lt**.

To make a query against a Table, you must create a filter string to pass as part of the http(s) REST API request. You can create this string anyway you'd like. For example, you can simply call **String.Format**. But, many developers don't like writing code to build strings due to the lack of IntelliSense, compile-time type-safety, and refactoring support.

The **Microsoft.WindowsAzure.Storage.Table.TableQuery** class offers several static methods (such as **GenerateFilterConditionForInt**) that you can use to help you create strings. These methods provide no IntelliSense support to help you with property name, no compile-time type-safety, and no refactoring support for when you change a property name. These methods provide almost no value over just calling **String.Format** and, in fact, using them makes the code more difficult to read and so I never use any of these methods.

Microsoft's Azure Storage client library allows you to create LINQ queries in code that, at runtime, produce the $filter query string for you. LINQ is a technology allowing you to create a query enabling IntelliSense, compile-time type-safety, and refactoring. In addition, in theory, LINQ allows you to change the underlying data store without having to change the code that makes a query against that data store. However, LINQ has substantial overhead because it creates many objects on the

heap (which must ultimately be GC'd) and the performance of walking through the linked-list of state machines that it creates is not as fast as if you accessed the data store directly.

If you're OK with the overhead, then using LINQ with Tables is fine in the most simple of scenarios. But, I personally never use LINQ with Tables and I would strongly warn you to avoid it as well. The problem is that LINQ allows you to easily create very rich queries that are simply not supported by Azure Tables. For example, it is very easy to create a LINQ query that includes **`orderby`**, **`join`**, or **`groupby`** but the Table service itself doesn't support these operations. If you write a LINQ query that cannot be converted to a $filter query string, your code might compile and throw an exception. Or, to make matters worse, in some cases, executing the query will not throw and instead, the Table service will return incorrect results. The following code demonstrates creating a table with two entities that differ only by the case of their PartitionKey. Then, I show two LINQ queries that try to retrieve an entity and fail.

```
CloudTable ct = account.CreateCloudTableClient().GetTableReference("MyTable");
await ct.CreateIfNotExistsAsync();
DynamicTableEntity dte = new DynamicTableEntity("ABC", String.Empty);
await ct.ExecuteAsync(TableOperation.InsertOrReplace(dte));
dte = new DynamicTableEntity("abc", String.Empty);
await ct.ExecuteAsync(TableOperation.InsertOrReplace(dte));

// NOTE: This compiles and returns ZERO results:
var results = (from e in ct.CreateQuery<DynamicTableEntity>()
               where String.Compare(e.PartitionKey, "Abc", StringComparison.OrdinalIgnoreCase) == 0
               select e).ToArray();

// NOTE: This compiles and throws an ArgumentException:
results = (from e in ct.CreateQuery<DynamicTableEntity>()
           where String.Equals(e.PartitionKey, "Abc", StringComparison.OrdinalIgnoreCase)
           select e).ToArray();
```

I'll add one more thing here: I worked with a group that was using LINQ to access Azure Tables. They wrote the code, tested it, and got the correct results. However, after the code was in production, they noticed some severe performance problems. After much research, they discovered that they wrote the LINQ query in such a way that caused all the table entities to be downloaded and be processed on the virtual machine instead of having the Table service process the query returning just the results. For all these reasons, I just avoid using LINQ with Azure tables.

Processing a Query's Results

The most efficient way to retrieve an entity from a Table is to do what is called a *point query*. A point query is when you retrieve an entity via its PartitionKey and RowKey values which uniquely identify it. This allows the Table service to quickly find the specific entity. All other queries cause the Table service to scan multiple entities within the table with some performance cost. To reduce the amount of the Table to scan, you should always try to restrict a query to a single partition when possible.

When you issue a query against a Table, the Table service can potentially return billions of result entities. Since Azure Storage is a multi-tenant service, it cannot dedicate all its resources to one client's request. For this reason, a Table ***always*** returns entities in chunks. Your code should process a returned chunk and then read/process another chunk, continuing until all chunks have been processed or until you have found what you're looking for.

> ***I have seen many developers write code that reads all the chunks accumulating the results in some collection and then they process the collection. However, you should not do this (unless you are very sure that the final result set is very small) because keeping thousands or***

millions of entity objects in memory is very memory intensive causing your application's performance to suffer greatly.

When you first issue a query, a chunk is returned along with a *continuation token* (an opaque string value). Your code should process the chunk and then look at the continuation token. If the continuation token is not **`null`**, then your code should make another http(s) REST API request to the table to get the next chunk. When you make this next request, you pass in the previously-returned continuation token; this tells the table where in the table it should continue scanning for results. The Table service returns a **`null`** continuation token after it has finished scanning the partition (or Table). You then process the last chunk's entities and then go on with your code.

Microsoft's Azure Storage library defines classes that are supposed to simplify the code you must write to deal with chunks (they call them segments) and continuation tokens. Here's what code looks like to process all the chunks returned by Table query using Microsoft's classes:

```
TableQuery query = new TableQuery();
for (TableQuerySegment<DynamicTableEntity> querySegment = null;
   querySegment == null || querySegment.ContinuationToken != null; ) {

   // Query the Table's first (or next) segment
   querySegment = await cloudTable.ExecuteQuerySegmentedAsync(query,
      querySegment == null ? null : querySegment.ContinuationToken);

   // Process the segment's entities
   foreach (DynamicTableEntity dte in querySegment.Results) {
      // TODO: Place code to process each entity
   }
}
```

This code pattern works but I find it cumbersome to use. Here's what I don't like about it:

- It's hard to remember this pattern as it is not like most other **`for`** loops.
- I don't like the complexity of the **`for`** test and the complexity of the second argument to **`ExecuteQuerySegmentedAsync`** because the developer must explicitly examine the continuation token.
- You must pass the same **`TableQuery`** object into **`ExecuteQuerySegmentedAsync`** every time.
- This pattern makes it easy for developers to create the **`TableQuery`** object within the loop which allocates a new one with each iteration. This negatively impacts performance due to all the garbage collections that will have to occur.

Wintellect's Azure Storage library defines a **`TableQueryChunk`** class (discussed later) that simplifies this code and adds some sorely needed features as well.

Wintellect's Azure Storage Library

I created Wintellect's Azure Storage library to compensate for many of the shortcomings I felt existed when using Microsoft's Azure Storage library. My library uses small portions of Microsoft's library and therefore requires Microsoft.WindowsAzure.Storage.dll at runtime. The Wintellect library simplifies the code necessary when working with Tables while increasing productivity, scalability, memory efficiency, performance, and reliability. This library has been evolving since 2011 and I continuously improve it as I work on projects leveraging Azure storage.

This section's subsections discuss how my library improves on the deficiencies mentioned in the previous section. But, before we get into these subsections, I want to show you code that uses the Wintellect library's types. This way, when you read the remaining subsections you'll have something to refer back to so you can see how the types are being used.

The example code maintains a table of gamers; each gamer has a gamer tag (a **`String`**), a score (an **`Int32`**), and a collection of levels completed (each completed level consists of the level name [Dungeon, Castle, Forrest, etc.] and the timestamp indicating when the gamer completed that level). The **`GamerEntity`** class is defined as follows:

```
public sealed class GamerEntity : TableEntityBase {
   /// <summary>
   /// Factory method that looks up a Gamer via their gamer tag.
   /// </summary>
   /// <param name="table">Table to look up gamer in.</param>
   /// <param name="tag">Gamer tag identifying the gamer.</param>
   /// <returns>The GamerEntity (if exists) or throws EntityNotFoundException.</returns>
   public static async Task<GamerEntity> FindAsync(AzureTable table, String tag) {
      // Try to find gamer via their gamer tag
      // NOTE: Throws EntityNotFoundException if tag not found
      DynamicTableEntity dte = await EntityBase.FindAsync(table, tag.ToLowerInvariant(), String.Empty);

      // Wrap DynamicTableEntity object with .NET class object & return wrapper
      return new GamerEntity(table, dte);
   }

   /// <summary>
   /// Wraps a GamerEntity object around a DynamicTableEntity and an Azure table.
   /// </summary>
   /// <param name="table">The table the GamerEntity should interact with.</param>
   /// <param name="dte">The DynamicTableEntity object this .NET class object wraps.</param>
   public GamerEntity(AzureTable table, DynamicTableEntity dte = null) : base(table, dte) { }

   /// <summary>
   /// Creates a new GamerEntity object around a new DynamicTableEntity and Azure table.
   /// </summary>
   /// <param name="table">The table the GamerEntity should interact with.</param>
   /// <param name="tag">The gamer tag uniquely identifying the gamer.</param>
   public GamerEntity(AzureTable table, String tag) : this(table) {
      // Gamer is identified with PK=Tag (lowercase) & RowKey=""
      PartitionKey = tag.ToLowerInvariant();
      RowKey = String.Empty;  // This example doesn't use the RowKey for anything

      Tag = tag;    // Save the case-preserved gamer tag value
   }

   /// <summary>
   /// The gamer's tag (case-preserved).
   /// </summary>
   // This property's .NET name is different from its entity name
   public String Tag {  // .NET property name
      get { return Get(String.Empty, "GamerTag"); } // Entity property name
      private set { Set(value, "GamerTag"); }       // Entity property name
   }

   /// <summary>
   /// The gamer's current score (defaults to 0).
   /// </summary>
   // This property's .NET name is the same as its entity name
   public Int32 Score {
      get { return Get(0); } // If "Score" property doesn't exist, return 0
      set { Set(value); }
   }

   // VERSIONING: You can easily add/remove .NET properties in the future.
   // Old entities without the property will return the default value.
   // If you remove a .NET property, the entity property will just be ignored.
   // If you remove a .NET property and Replace the entity, the property
   // will be removed from the entity.

   /// <summary>
   /// A string showing the gamer's info (for debugging).
   /// </summary>
```

```
   /// <returns>Gamer info.</returns>
   public override String ToString() {
      return String.Format("Tag={0}, Score={1}", Tag, Score);
   }

   #region Members supporting the Gamer's CompletedLevel collection
   private static readonly Byte[] s_empty = new Byte[0];
   private const Int32 c_maxCompletedLevelProperties = 1;
   public const String CompletedLevelsPropertyName = "CompletedLevels";

   // I prefer methods here instead of a property because these methods are
   // computationally expensive and I want callers to carefully consider
   // calling them. Callers tend to access properties without much thought
   // as to what they do internally.
   /// <summary>
   /// Returns completed levels associated with the Gamer.
   /// </summary>
   public IEnumerable<CompletedLevel> GetCompletedLevels() {
      // Read the byte array containing the serialized CompletedLevel collection
      Byte[] bytes = Get(s_empty, c_maxCompletedLevelProperties, CompletedLevelsPropertyName);

      // Deserialize the byte array to a collection
      return CompletedLevel.Deserialize(bytes);
   }

   /// <summary>
   /// Associates completed levels with the Gamer.
   /// </summary>
   /// <param name="completedLevels"></param>
   /// <returns></returns>
   public GamerEntity SetCompletedLevels(IEnumerable<CompletedLevel> completedLevels) {
      // Serialize the Chapter collection to a byte array
      Byte[] bytes = CompletedLevel.Serialize(completedLevels);

      // Save the byte array to a single table property
      Set(bytes, c_maxCompletedLevelProperties, CompletedLevelsPropertyName);
      return this;
   }
   #endregion
}
```

The **`GetCompletedLevels`** and **`SetCompletedLevels`** methods operate on collections of **`CompletedLevel`** objects. The **`CompletedLevel`** class looks like this:

```
public enum Level { Dungeon, Castle, Forrest, /* ... */ }

public sealed class CompletedLevel {
   public readonly Level Level;
   public readonly DateTimeOffset Completed;
   public CompletedLevel(Level level, DateTimeOffset completed) {
      Level = level; Completed = completed;
   }

   #region Serialization & Deserialization members
   private const UInt16 c_version = 0; // Increment this when you change (de)serialization format
   private const Int32 c_ApproxRecordSizeV0 = sizeof(Level) + sizeof(Int64);
   internal static Byte[] Serialize(IEnumerable<CompletedLevel> completedLevels) {
      // Serialize the collection to a byte array
      return PropertySerializer.ToProperty(
         maxRecordsPerEntity: AzureTable.MaxPropertyBytes / c_ApproxRecordSizeV0,
         version: c_version,
         records: completedLevels,
         approxRecordSize: c_ApproxRecordSizeV0,   // Improves performance
         recordToProperty: (chapter, version, bytes) => chapter.Serialize(version, bytes));
   }
   private void Serialize(UInt16 version, PropertySerializer propertyBytes) {
      switch (version) {
         case 0:
```

```
            propertyBytes.Add((Int32)Level).Add(Completed.UtcTicks);
            break;
         default:
            throw new InvalidOperationException("'version' identifies an unknown format.");
      }
   }
   internal static IEnumerable<CompletedLevel> Deserialize(Byte[] bytes) {
      // Deserialize the byte array to a collection
      return PropertyDeserializer.FromProperty<CompletedLevel>(bytes,
         (version, deserializer) => new CompletedLevel(version, deserializer));
   }
   private CompletedLevel(UInt16 version, PropertyDeserializer propertyBytes) {
      switch (version) {
         case 0:
            Level = (Level)propertyBytes.ToInt32();
            Completed = new DateTimeOffset(propertyBytes.ToInt64(), TimeSpan.Zero);
            break;
         default:
            throw new InvalidOperationException("'version' identifies an unknown format.");
      }
   }
   #endregion
}
```

Finally, here is the **GamerManager** class that shows how to perform various operations with gamers:

```
public sealed class GamerManager {

   // This method demonstrates various gamer operations.
   public static async Task DemoAsync() {
      GamerManager gm = new GamerManager();

      // Create 2 gamers
      GamerEntity dragon = await gm.CreateGamerAsync("Dragon");
      GamerEntity creeper = await gm.CreateGamerAsync("Creeper");

      // Increase a Gamer's score & indicate what level they completed
      await gm.IncreaseScoreAsync("Dragon", 10);
      await gm.SetCompletedLevelAsync("Dragon", Level.Castle, DateTimeOffset.UtcNow);

      await gm.ShowUsersAsync(20, 40, true); // Shows 0 gamers

      // Increase a Gamer's score & indicate what level they completed
      await gm.IncreaseScoreAsync("Creeper", 20);
      await gm.SetCompletedLevelAsync("Creeper", Level.Forrest, DateTimeOffset.UtcNow);

      await gm.ShowUsersAsync(20, 40, false); // Shows 1 gamer

      // Increase a Gamer's score & indicate what level they completed
      await gm.IncreaseScoreAsync("Dragon", 20);
      await gm.SetCompletedLevelAsync("Dragon", Level.Forrest, DateTimeOffset.UtcNow);

      await gm.ShowUsersAsync(20, 40, true); // Shows 2 gamers
   }

   // Maintain a reference to the table (the table must be created externally)
   private readonly AzureTable m_table = CloudStorageAccount.DevelopmentStorageAccount
         .CreateCloudTableClient().GetTableReference("Gamers");

   /// <summary>
   /// Inserts a new Gamer (via their tag) into the Azure table.
   /// </summary>
   /// <param name="tag">The gamer tag uniquely identifying the gamer.</param>
   /// <returns>The GamerEntity just inserted into the table.</returns>
   /// <exception cref="StorageException">Thrown if tag already in use.</exception>
   public async Task<GamerEntity> CreateGamerAsync(String tag) {
      GamerEntity gamer = new GamerEntity(m_table, tag);
```

```
   // NOTE: Throws StorageException (HttpStatusCode=409 [Conflict]) if tag already in use
   await gamer.InsertAsync();
   return gamer;
}

/// <summary>
/// Increases a gamer's score.
/// </summary>
/// <param name="tag">The gamer tag uniquely identifying the gamer.</param>
/// <param name="increase">The amount to increase gamer's score by (can be negative).</param>
/// <returns>The gamer's score after the change.</returns>
public Task<Int32> IncreaseScoreAsync(String tag, Int32 increase) {
   return AzureTable.OptimisticRetryAsync(async () => {
      GamerEntity gamer = await GamerEntity.FindAsync(m_table, tag);
      gamer.Score += increase;
      await gamer.MergeAsync();
      return gamer.Score;  // Returns the gamer's new score
   });
}

/// <summary>
/// Record's that gamer completed a game level.
/// </summary>
/// <param name="tag">The gamer tag uniquely identifying the gamer.</param>
/// <param name="level">The level the gamer just completed.</param>
/// <param name="completed">The time when the gamer completed the level.</param>
public Task SetCompletedLevelAsync(String tag, Level level, DateTimeOffset completed) {
   return AzureTable.OptimisticRetryAsync(async () => {
      // Find the Gamer
      GamerEntity gamer = await GamerEntity.FindAsync(m_table, tag);

      // Get Gamer's CompletedLevels, append new CompletedLevel, set CompletedLevels
      IEnumerable<CompletedLevel> levels = gamer.GetCompletedLevels();
      if (levels.Any(l => l.Level == level)) return; // Level already completed; nothing to do

      levels = levels.Concat(new[] { new CompletedLevel(level, completed) });
      gamer.SetCompletedLevels(levels);
      await gamer.MergeAsync();  // Update the entity in the table
   });
}

/// <summary>
/// Displays all gamers with a score between two values.
/// </summary>
/// <param name="lowScore">The low score.</param>
/// <param name="highScore">THe high score.</param>
public async Task ShowUsersAsync(Int32 lowScore, Int32 highScore, Boolean showCompletedLevels) {

   Console.WriteLine("GAMERS WITH SCORES BETWEEN {0} AND {1}:", lowScore, highScore);

   // Create a query that scans the whole table looking for gamers
   // with a score between lowScore & highScore inclusive
   TableQuery query = new TableQuery {
      FilterString = new TableFilterBuilder<GamerEntity>()
         .And(ge => ge.Score, CompareOp.GE, lowScore)
         .And(ge => ge.Score, CompareOp.LE, highScore),

      // This shows to request a subset of properties (Tag & Score) be returned
      SelectColumns = new PropertyName<GamerEntity>()[ge => ge.Tag, ge => ge.Score].ToList()
   };
   // This conditionally adds another property to be returned
   if (showCompletedLevels) query.SelectColumns.Add(GamerEntity.CompletedLevelsPropertyName);

   // Prepare to execute the query in chunks (segments)
   for (TableQueryChunk chunker = m_table.CreateQueryChunker(query); chunker.HasMore; ) {

      // Query a result segment
      foreach (DynamicTableEntity dte in await chunker.TakeAsync()) {
```

```
            // Wrap the DynamicTableEntity in a type-safety, IntelliSense, Refactoring class (BookEntity)
            GamerEntity gamer = new GamerEntity(m_table, dte);

            // Now, manipulate the entity
            Console.WriteLine("Tag={0}, Score={1}", gamer.Tag, gamer.Score);
            foreach (CompletedLevel completedLevel in gamer.GetCompletedLevels()) {
               Console.WriteLine("   {0,-10}   {1}", completedLevel.Level, completedLevel.Completed);
            }
            Console.WriteLine();

            // We could modify the entity here and update the table by calling be.MergeAsync
            // Or, we could delete the entity by calling be.DeleteAsync

            // Or, we could add an entity operation to a TableBatchOperation by calling
            // Insert, InsertOrMerge, InsertOrReplace, Merge, Replace, or Delete.
         }
      }
      Console.WriteLine();
   }
}
```

As you read the following subsections, I will periodically refer you back to this code so you can see how the concepts and library types and methods are being used. All my work with Azure Tables follow the guidelines I present below and all my code for working with Tables very closely matches the patterns I show above.

Methods that Perform I/O Operations

The **`Wintellect.Azure.Storage.Table.AzureTable`** struct is a light-weight (value type) wrapper over Microsoft's **`Microsoft.WindowsAzure.Storage.Table.CloudTable`** class. **`AzureTable`** is effectively identical to **`CloudTable`** and has implicit conversion operators casting a **`CloudTable`** to an **`AzureTable`** and vice versa. However, **`AzureTable`** exposes only the methods I consider good methods to call. For example, **`AzureTable`** does not expose any methods that perform I/O operations synchronously or via the old .NET asynchronous programming model (**`BeginXxx/EndXxx`** methods). Hiding unwanted methods forces the use of good Table practices and reduces IntelliSense clutter. You can see the use of my **`AzureTable`** struct In **`GamerManager`**'s **`Initialize`** method and **`GamerEntity`**'s two constructors and its static **`FindAsync`** method.

Converting Table Entities to .NET Classes

To maintain a proper mental model when working with Tables and to leverage its features (such as entity versioning), I believe it is a bad practice to convert entities to .NET class objects and vice versa. Instead, I ***always convert*** a table entity's JSON to a **`DynamicTableEntity`** object. And, I ***always convert*** a **`DynamicTableEntity`** object to JSON when sending it back to the Table. However, to get IntelliSense, compile-time type-safety, and refactoring support in my .NET code, I ***wrap a .NET class around*** a **`DynamicTableEntity`** object.

The **`Wintellect.Azure.Storage.Table.EntityBase`** class is an abstract base class that implements the **`Microsoft.WindowsAzure.Storage.Table.ITableEntity`** interface:

```
public abstract class EntityBase : ITableEntity {
   // Static factory method to find an entity in a table
   public static async Task<DynamicTableEntity> FindAsync(AzureTable at,
      String partitionKey, String rowKey) {
         TableResult tr = await at.ExecuteAsync(
            TableOperation.Retrieve(partitionKey.HtmlEncode(), rowKey.HtmlEncode()))
            .ConfigureAwait(false);

   if ((HttpStatusCode)tr.HttpStatusCode == HttpStatusCode.NotFound)
      throw new EntityNotFoundException(partitionKey, rowKey);
   return (DynamicTableEntity)tr.Result;
}
```

```
   private readonly DynamicTableEntity Entity;

   protected EntityBase(DynamicTableEntity dte = null) {
      Entity = dte ?? new DynamicTableEntity();
   }

   #region Common properties: ETag, PartitionKey, RowKey, Timestamp
   public String ETag {
      get { return Entity.ETag; }
      set { Entity.ETag = value; }
   }
   public String PartitionKey {
      get { return Entity.PartitionKey; }
      set { Entity.PartitionKey = value; }
   }
   public String RowKey {
      get { return Entity.RowKey; }
      set { Entity.RowKey = value; }
   }
   public DateTimeOffset Timestamp {
      get { return Entity.Timestamp; }
      set { Entity.Timestamp = value; }
   }
   #endregion

   #region Inherited entity property getters & setters
   // These protected Get and Set methods simply call the extension methods
   protected Boolean Get(Boolean defaultValue,
      [CallerMemberName] String tablePropertyName = null) {
      return Entity.Get(defaultValue, tablePropertyName);
   }
   protected void Set(Boolean value,
      [CallerMemberName] String tablePropertyName = null) {
      Entity.Set(value, tablePropertyName);
   }

   // Not shown: Get and Set methods for all remaining property types:
   // Int32, Int64, Double, Guid, DateTimeOffset, String, and Byte[]
   #endregion

   // Not shown: Methods to add an entity operation to a TableBatchOperation
}
```

This class allows it to be passed to any of Microsoft's Azure Storage library methods accepting this interface. It has a private, read-only **DynamicTableEntity** field and exposes its **PartitionKey**, **RowKey**, **Timestamp**, and **ETag** properties. It also has protected methods that call **DynamicTypeEntityExtension**'s extension methods (discussed in the next subsection), making them easy to call from a derived class. Finally, the class offers a fluent API, making it easy to add multiple **EntityBase**-derived objects to a **Microsoft.WindowsAzure.Storage.Table.TableBatchOperation** object.

You can define your own class, derive it from **EntityBase** and add any .NET properties you desire. The .NET properties provide IntelliSense, compile-time type-safety and refactoring support throughout the rest of your code. You can easily implement the properties by calling **EntityBase**'s **Get** and **Set** methods.

The **Wintellect.Azure.Storage.Table.TableEntityBase** class is derived from **EntityBase** and associates an **AzureTable** (or **CloudTable**) with the object. This creates an object that wraps an entity's properties *and* its Table. You can then manipulate the entity's properties and then tell the entity to update itself on the table via **TableEntityBase**'s **InsertAsync**, **InsertOrMergeAsync**, **InsertOrReplaceAsync**, **MergeAsync**, **ReplaceAsync**, or **DeleteAsync** methods. Here is what the **TableEntityBase** class looks like:

```
public class TableEntityBase : EntityBase {
   private readonly AzureTable m_azureTable;
```

```
   protected TableEntityBase(AzureTable at, DynamicTableEntity dte = null) : base(dte) {
      m_azureTable = at;
   }

   public Task<TableResult> InsertAsync() {
      return m_azureTable.ExecuteAsync(TableOperation.Insert(this));
   }

   // Not shown: InsertOrMergeAsync, InsertOrReplaceAsync, MergeAsync, ReplaceAsync, DeleteAsync
}
```

The **GamerEntity** class shown earlier is derived from **TableEntityBase**. In the **GamerManager** class, you'll see how simple it is to write methods (such as **CreateGamerAsync**, **IncreaseScoreAsync**, **SetCompletedLevelAsync**) that internally manipulate an entity on its Table.

Properties with null Values

My **Wintellect.Azure.Storage.Table.DynamicTableEntityExtensions** static class defines several extension methods to Microsoft's **Microsoft.WindowsAzure.Storage.Table.DynamicTableEntity** class. The extension methods looks like this:

```
public static class DynamicTableEntityExtensions {
   public static Boolean Get(this DynamicTableEntity dte, Boolean defaultValue,
      [CallerMemberName] String tablePropertyName = null);

   public static void Set(this DynamicTableEntity dte, Boolean value,
      [CallerMemberName] String tablePropertyName = null);

   // I have overloads of the same two methods above for the remaining types.
   // The Set overloads for String and Byte[] throw ArgumentNullException if you pass null.

   // For the Byte[] type, Get & Set support maxProperties.
   // See the "Storing Collections within an Entity" section towards the end of this paper.
   public static Byte[] Get(this DynamicTableEntity dte, Byte[] defaultValue, Int32 maxProperties = 1,
      [CallerMemberName] String tablePropertyName = null);
   public static void Set(this DynamicTableEntity dte, Byte[] value, Int32 maxProperties = 1,
      [CallerMemberName] String tablePropertyName = null);

   // Call this if you use $select passing a property that doesn't exist in the entity
   public static DynamicTableEntity RemoveNullProperties(this DynamicTableEntity dte);
}
```

As you can see, **Get** and **Set** methods exist only for the types supported by Tables: **Boolean**, **Int32**, **Int64**, **Double**, **Guid**, **DateTimeOffset**, **String**, and **Byte[]**. Since Table properties do not support **null** values, the **Set** methods that accept value types do not allow **null** to be passed in. The **Set** methods that accept **String** and **Byte[]** throw an **ArgumentNullException** if **null** is passed in. The **Get** methods accept default values which are returned if the entity does not have the property.

Also, note that the **Get** and **Set** methods take the **[CallerMemberName]** as their last argument. This allows the methods to create a property name that matches the C# property name. This simplifies code without compromising on memory or performance. You can also explicitly pass a property name if you want the C# property name and the entity property name to differ. **GamerEntity**'s **Score** property shows how to use the same property name for .NET as well as in the entity. **GamerEntity**'s **Tag** property shows how to have the .NET property name differ from the entity's property name. Also, look at **GamerEntity**'s **GetCompletedLevels** and **SetCompletedLevels** methods; these show how to create methods (not .NET properties) and use an entity's property internally.

As mentioned earlier, a property cannot be more than 65536 bytes. When you pass a **Byte[]** larger than this size, the **Set** method can automatically create multiple properties and split the **Byte[]** across these properties. The **Get** method can reassemble the properties back into a single **Byte[]**. You must specify how many properties you're willing to split the

Byte[] across. **GamerEntity**'s **GetCompletedLevels** and **SetCompletedLevels** methods take advantage of this feature. The **SetCompletedLevels** method takes a collection of **CompletedLevel** objects, serializes them all to a single **Byte[]** and then writes the **Byte[]** to a single property (although it could span multiple **Byte[]** properties). The **GetCompletedLevels** method deserializes the **Byte[]** property (or properties) back to a collection of **CompletedLevel** objects.

Creating Filter Strings to Query a Table

The Wintellect Azure Storage library provides a **Wintellect.Azure.Storage.Table.TableFilterBuilder** struct (value type) and some supporting enums that gives you an easy way to create a filter string to query a table. **TableFilterBuilder** allows you to build a filter string using IntelliSense, compile-time type-safety, and refactoring support. So, it is better than performing raw string manipulations. And, it constrains you to operations actually supported Tables as opposed to the leaky abstraction and overhead associated with using LINQ to build Table filter strings.

Here is what **TableFilterBuilder** and its supporting enums looks like:

```
public enum EqualOp { EQ, NE }
public enum CompareOp { EQ = EqualOp.EQ, NE = EqualOp.NE, GT, GE, LT, LE }
public enum RangeOp {
   IncludeLowerExcludeUpper = (CompareOp.GE << 8) | CompareOp.LT,
   StartsWith = IncludeLowerExcludeUpper,
   IncludeLowerIncludeUpper = (CompareOp.GE << 8) | CompareOp.LE,
   ExcludeLowerExcludeUpper = (CompareOp.GT << 8) | CompareOp.LT,
   ExcludeLowerIncludeUpper = (CompareOp.GT << 8) | CompareOp.LE
}


public struct TableFilterBuilder<TEntity> {
   public static implicit operator String(TableFilterBuilder<TEntity> fe);

   public TableFilterBuilder<TEntity> And(TableFilterBuilder<TEntity> other);

   public TableFilterBuilder<TEntity> And(Expression<Func<TEntity, Boolean>> property,
      EqualOp equalOp, Boolean value);

   // Overloads for And exist operating on the other Table-supported data types

   public TableFilterBuilder<TEntity> And(Expression<Func<TEntity, String>> property,
      String lowerValue, RangeOp rangeOp = RangeOp.StartsWith, String upperValue = null);

   // There are also 'Or' methods that have identical signatures to the And methods
}
```

Here is an example that creates a complex filter string using **TableFilterBuilder**:

```
TableQuery query = new TableQuery {
   FilterString = new TableFilterBuilder<MyTableEntity>()
      .And(te => te.PartitionKey, "Jeff", RangeOp.StartsWith)  // PartitionKey starts with "Jeff"
      .And(te => te.ABoolean, EqualOp.EQ, true)
      .And(te => te.AByteArray, EqualOp.EQ, new Byte[] { 1, 2, 3 })
      .And(te => te.ADate, CompareOp.GT, new DateTimeOffset(2000, 1, 1, 0, 0, 0, TimeSpan.Zero))
      .And(te => te.ADouble, CompareOp.LT, 1.23)
      .And(te => te.AGuid, EqualOp.EQ, Guid.NewGuid())
      .And(te => te.AnInt32, CompareOp.GE, 123)
      .And(te => te.AnInt64, CompareOp.NE, 321)
};
```

When typing in the line above, I get IntelliSense support to autocomplete the property names. If I rename a property by refactoring, then the property names change in the code automatically. Also, I get compile-time type-safety. For example, the code will not compile if the property type is a **DateTimeOffset** and I pass a value other than a **DateTimeOffset** to the

last argument of an **And** or **Or** method. In addition, some methods take a **CompareOp**, some take an **EqualOp**, and some take a **RangeOp** forcing the code to perform only legal comparisons for the property's data type.

When the line above executes, it produces the following filter string:

```
(((((((((PartitionKey ge 'Jeff') and (PartitionKey lt 'Jefg'))
and (ABoolean eq true))
and (AByteArray eq X'010203'))
and (ADate gt datetime'2000-01-01T00:00:00.0000000Z'))
and (ADouble lt 1.23))
and (AGuid eq guid'b5856e12-2919-4350-acfe-47ceca4d510a'))
and (AnInt32 ge 123))
and (AnInt64 ne 321))
```

To execute a query, we just need to create and initialize a **TableQuery** object and then call **AzureTable**'s (or **CloudTable**'s) **ExecuteQuerySegmentedAsync** method.

```
TableQuerySegment<DynamicTableEntity> segment =
   await ct.ExecuteQuerySegmentedAsync(query, null, null, null, CancellationToken.None);
```

By the way, this method is the only method I ever use to execute a table query. I recommend avoiding all other methods as they are either synchronous (limiting responsiveness and scalability) or *convert* entities to .NET objects (as opposed to *wrapping* **DynamicTableEntity** objects with .NET objects).

In addition to **FilterString**, Microsoft's **TableQuery** class defines two other properties: **TakeCount** (a **Int32?**) and **SelectColumns** (an **IList<String>**). **TakeCount** lets you specify the maximum number of entities you want the Table to return; leave it **null** to have it return all entities (in chunks) matching the filter string.

The **SelectColumns** property is poorly named. Tables do not have columns; they have properties. So, **SelectColumns** should have been called **SelectProperties** instead. The **SelectColumns** property refers to an **IList<String>** containing the names of the properties you want returned from the Table. If you leave it **null**, then all properties are returned. If you set it to a list of **String**s, you'll get the properties you indicate.[6] Specifying only the properties you absolutely need reduces latency and decreases transfer costs (if data leaves an Azure datacenter). Again, instead of specifying strings, I prefer to have IntelliSense and refactoring support when creating this list of **String**s. To assist with this, I have created another type called **Wintellect.PropertyName<T>**. This lightweight (struct) type offers three indexer properties:

```
public struct PropertyName<T> {
   public String this[LambdaExpression propertyExpression] { get; }
   public String this[Expression<Func<T, Object>> propertyExpression] { get; }
   public String[] this[params Expression<Func<T, Object>>[] propertyExpressions] { get; }
}
```

GamerManager's **ShowGamersAsync** method shows how to use my library's **TableFilterBuilder** and **PropertyName** types.

Processing a Query's Results

The **Wintellect.Azure.Storage.Table.TableQueryChunker** class simplifies the code necessary to process chunks of entities returned from a Table query. **GamerManager**'s **ShowGamersAsync** method shows the nested loops necessary to process each chunk of the Table's results.

---

[6] Note: You always get back the PartitionKey, RowKey, and Timestamp properties whether you specify them or not. You'll also get back the entity's ETag as well (which matches the Timestamp).

Each call to **TakeAsync** causes Azure's Table service to return a chunk of results. With each call, the Table service may return anywhere from 0 to 1,000 results. **GamerManager**'s **ShowGamersAsync** method doesn't care about this; each chunk returned (no matter how big it is) is just processed until all entities have been processed.

However, sometimes, you want to process some fixed number of entities at a time. For example, a webpage might want to show the user 25 results per page. The code you'd have to write to query the table and ensure that you have 25 and no more) and then to get to the next set of 25 (continuing from where the last chunk left off) is quite substantial and error prone. However, the code to manage all of this for you is part of the **TableQueryChunker** class. Its **TakeAsync** method has an overload allowing you to pass in a **chunkSize** (**Int32**) argument. This overload is not as memory efficient as the overload that does not take a **chunkSize**, so you should only use this overload when you absolutely need to process entities in fixed-sized chunks.

Safely Updating Table Entities

It is possible that two or more computers might attempt to update the same Table entity at the same time. To ensure no loss of data or data corruption, code must use an optimistic-concurrency pattern. For example, let's say there is an entity that looks like this:

```
public sealed class Counter : TableEntityBase {
   public Counter(AzureTable table, DynamicTableEntity dte = null) : base(table, dte) { }
   public String Name {
      get { return Get("Unknown"); }
      set { PartitionKey = value.ToLowerInvariant().HtmlEncode(); Set(value); }
   }
   public Int32 Count {
      get { return Get(0); }
      set { Set(value); }
   }
}
```

And let's say that on a Table, an entity exists whose **Count** property's value is **0**. If two computers are trying to increment this value: the end result should be **2**. But, here's a possible scenario: both computers first read the entity whose **Count** property is **0**, both computers increment their copy of **Count**'s value to **1**, both merge the new entity back on to the Table. If this happens, the end result is that **Count** contains **1**; not the desired **2** – data has been corrupted.

We solve this race condition using an optimistic concurrency technique:

- Both computers read the entity whose property is **0**. Both computers also get the entity's **ETag** which is really a timestamp indicating when the entity was last modified on the Table.
- Both computers increment their copy of **Count**'s value to **1**.
- Both computers attempt to merge the updated entity back on to the Table. Both requests include the original entity's **ETag** (timestamp).
- The Table service always processes write requests sequentially (not in parallel). The first request's **ETag** matches the timestamp of the entity on the table, so the service accepts the first request changing the **Count**'s value to **1**. This also changes the entity's timestamp.
- The Table service processes the second request but the **ETag** passed in the request doesn't match the entity's new timestamp and so the service fails the second request and does not update the entity. The second computer is notified of the failure (via an HTTP 412 error "Precondition Failed").
- In response to the failure, the second computer re-reads the entity from the table getting the current **ETag** (timestamp) and **Count**'s current value of **1**. Then this computer increments **Count**'s value to **2** and attempts to merge the updated entity back on to the table. The request includes the latest **ETag** which matches the

timestamp of the entity on the Table and therefore the Table service accepts the request changing **Count**'s value to **2** (data is not corrupt). This also updates the entity's timestamp as well.

To avoid data corruption, the optimistic concurrency pattern must be used whenever code attempts to update an entity's properties; the code must continuously retry (loop around) until its update is successful. Whenever you have code that reads an entity, updates a property, and merges (or replaces) the entity back onto the Table, you should be performing these operations using this pattern. To simplify the code required to do this, **AzureTable** offers a static method, called **OptimisticRetryAsync,** implemented as follows:

```
public static async Task<TResult> OptimisticRetryAsync<TResult>(
   Func<Task<TResult>> operation, Int32 maxRetries = -1) {

   for (Int32 retry = 0; ; retry++) {
      try {
         return await operation().ConfigureAwait(false);
      }
      catch (StorageException se) {
         // If optimistic concurrency failed, it's OK, try again
         if (!se.Matches(HttpStatusCode.PreconditionFailed)) throw;
         if (maxRetries >= 0 && retry >= maxRetries) throw;
      }
   }
}
```

Using this method makes implementing the optimistic concurrency pattern trivial. Here is code using this method to properly update a **Counter** entity:

```
public static Task<Int32> IncrementCount(AzureTable table, String counterName) {
   return AzureTable.OptimisticRetryAsync(async () => {
      // Find the entity in the table we wish to update
      DynamicTableEntity dte = await TableEntityBase.FindAsync(table, counterName, String.Empty);

      // Wrap the entity with an IntelliSense, type-safety, refactoring wrapper
      Counter counter = new Counter(table, dte);

      counter.Count++;            // Increment the count

      await counter.MergeAsync(); // Attempt update (throws if optimistic concurency fails)
      return counter.Count;       // Return what we successfully set the Count to
   });
}
```

See **GamerManager**'s **IncreaseScoreAsync** and **SetCompletedLevelAsync** methods for more examples using **AzureTable**'s **OptimisticRetryAsync** method for optimistic concurrency.

> Note: Microsoft's Azure Storage library methods automatically include an entity's ETag when you issue a Merge, Replace or Delete operation. The library does not include an entity's ETag when performing an Insert, InsertOrMerge or InsertOrReplace operation. Also, if you don't care about potential loss of data, you can explicitly set an entity's **ETag** property to **"*"** (asterisk) before performing any operation; this forces the Table service to perform the operation regardless of the entity's current ETag value.

Storing Collections within an Entity

When working with Tables, I sometimes want to store a collection of records as a property within the entity itself. For example, with each gamer, I need to store the game levels that the gamer has completed. You should be careful making entities too large as this will slow down the performance of the Table service when you scan your Table's entities. Also note

that adding, updating, or removing a record requires downloading the property's value, deserializing the records, making the desired change, serializing the records, and uploading the new property value; this can negatively impact performance. But, for small collections that are frequently accessed with the rest of the entity's properties, this can be a useful and convenient thing to do (as opposed to storing the data in another Table or blob).

To save the collection of completed levels, I define a **`CompletedLevel`** class with fields for the data I need to persist (**`Level`** and **`Completed`** date). Then I take the collection of these, serialize them into a byte array and then save this data in a byte array property inside the gamer's entity. If a **`Level`** is 4 bytes and a **`Completed`** date is an Int64 (8 bytes), then each record is 12 bytes. A single byte array property can hold 65536 bytes, so I can store up to 5,461 (65536 / 12) completed levels per gamer. Most games I imagine have far fewer levels than this but you can extend this example to address storing a larger collection with an entity.

By the way, when I serialize a collection of records to a byte array, I insert a **`UInt16`** at the beginning of the byte array. This **`UInt16`** contains a version number: 0, 1, 2, 3, and so on. When I first put my app into production, I start serializing version 0. Then, in the future, I might make a change to the record's schema (like adding a field) and I'd serialize an array of these records using version 1. Later, when I deserialize the byte array, I look at the **`UInt16`** to know which version of the record was serialized so I can deserialize it appropriately. Versioning byte arrays this way allows me to add fields to records as time goes on. And, it allows different entities in the table to have different versions of the same property at the same time. **`CompletedLevel`** objects inserted in an entity long ago have records following an old schema and recently-inserted **`CompletedLevel`** objects have records with the newer schema. The service can easily handle both and more versions as time goes on.

The **`CompletedLevel`** class shows all the code necessary to serialize and deserialize a collection of **`CompletedLevel`** objects. The code also shows how to handle versioning. Also, look at **`GamerEntity`**'s **`SetCompleteLevels`** and **`GetCompletedLevels`** methods for how to associate the collection with an entity and how to get the collection back. Finally, examine **`GamerManger`**'s **`SetCompletedLevelsAsync`** method to see how to update elements in the collection. Internally, the serialization and deserialize code uses the high performance and low memory consuming **`Wintellect.Azure.Storage.Table.PropertySerializer`** and **`Wintellect.Azure.Storage.Table.PropertyDeserializer`** classes. You can see these classes being used in **`CompletedLevel`**'s **`Serialize`** and **`Deserialize`** methods.

Additional Features

Wintellect's Azure Storage library provides other useful Table functionality not mentioned in this document. I'll briefly describe it here.

The **`Wintellect.Azure.Storage.Table.AzureTable`** type offers **`CopyToBlobAsync`** and **`CopyToTableAsync`** methods which copies a Table's entities to a blob or another Table. Note that other machines can change the source Table while the copy takes place and the copy is not guaranteed to be consistent. You can use **`AzureTable`**'s **`CopyFromBlobAsync`** method to restore a blob backup into an existing Table. We have found these methods to be incredibly useful. First, these methods allow us to backup a table to maintain a historical snapshot. Second, when we want to explore our table's data, we copy the table to a blob, download the blob to a PC and then restore the table into the Azure Storage Emulator. This allows us to experiment with the data free and fast. It also ensures that we do not accidentally corrupt the live data.

The **`Wintellect.Azure.Storage.Table.PropertyChanger`** class contains **`KeepOnlyAsync`** and **`RemoveAsync`** methods that walk an entire table updating each entity's properties. **`KeepOnlyAsync`** takes a set of property names that you wish to keep (and optionally rename). It walks through all entities and keeps only the properties you've specified; it can also rename the properties being kept. **`RemoveAsync`** walks through all entities and just removes any properties you indicate removed (any properties not specified are kept). I use this class to perform maintenance on a table by changing the schema of the

table's entities. NOTE: Most-likely you'll want no other machines accessing the table while these maintenance operations are running.

In Conclusion

I love the Azure Storage service and, in particular, Azure Tables. Over the many years of writing code for this service, I feel that I have learned a lot about the right mental model to have when working with Tables. This has enabled me to know what technologies I should use and which technologies I should avoid. I hope this document imparted this knowledge on to you. I hope this document made you aware of some shortcomings in the Azure storage .NET client library so you can avoid them enabling yourself to be more productive. Finally, this document introduced Wintellect's Azure Storage library and how it's designed to match the mental model for Tables enabling you to easily build scalable and efficient apps.