Accessing Azure

Using Cloud databases from static programming environments

Rik van der Sanden

Accessing Azure

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Rik van der Sanden
born in Eindhoven, The Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Avanade
Versterkerstraat 6
Almere, the Netherlands
www.avanade.com/nl

Accessing Azure

Author: Rik van der Sanden
Student id: 1116738
Email: rik.van.der.sanden@gmail.com

Abstract

Cloud databases generally use a data model different from relational databases. Instead of a fixed structure, the data is organized as collections of key-value pairs. The structure of entities is not enforced by the system; entities are flexible with regard to the properties they have. Many programming languages use static data models, which makes communication between these systems non-trivial. This document describes LINQ-to-Azure, a data access layer between Windows Azure Tables and .NET which enables manipulation and querying (using LINQ) of the data and supports both the flexible model of Azure Tables and the static one of .NET.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Drs. Z. Hemel, Faculty EEMCS, TU Delft
Company supervisor:	J. Vardy, Avanade

Contents

List of Figures

Chapter 1

Introduction

In recent years, Cloud computing has gained momentum as a hosting option for web applications. By using Cloud computing, it is possible to outsource many hardware concerns, including up-front cost, maintenance and scalability. Instead of running an application on a fixed number of servers, it is hosted by a Cloud computing vendor. This vendor runs the application within its computing Cloud, which generally consists of many virtualized servers. The actual amount of hardware available for the application can be adjusted dynamically to satisfy demand. This can result in significantly lower cost to run the application.

Since scalability is an important feature in Cloud computing, it needs a data storage system which also scales well. In traditional relational databases, query performance often suffers as the volume of data grows, because normalized data needs to be recombined at runtime. Therefore, most Cloud databases use a different data model, in which joins are not supported and entities are represented as a collection of key-value pairs. Such a data model is more flexible than the fixed tables used in relational databases, and more scalable. Many applications which use these databases, however, are written in statically typed languages, which work with classes with fixed properties. This forms a mismatch between the data models of these databases and the applications using them. The ability to use the flexibility of the data model is essential. Because of the lack of joins, a normalized data structure will be inefficient when it comes to querying the data; multiple queries are necessary to retrieve fragmented data, which has to be recombined by the application itself. A better way to use the data model is to store the data in a denormalized way. Its when using denormalization that the flexibility of the data model is useful. Properties that would be in different tables in a relational database, can be included within a single entity without having to specify of know exactly which properties an entity has and withut being limited by a fixed schema.

In this project, we will focus on Windows Azure, Microsoft's platform for cloud computing. Azure is currently in a preview phase and will be commercially released late 2009, at Microsoft's annual professional developers conference. At Avanade, this development is being watched with a keen interest.

Windows Azure includes a Cloud database called Windows Azure Tables. It has a flexible data model as described above, but, being part of Windows Azure, it is most likely accessed by application written in .NET. This means it will also suffer from the mismatch between fixed and flexible data models.

Windows Azure Tables implements ADO.NET Data Services, also known by its Microsoft codename, Astoria. For brevity, we will use this codename through the rest of this document. Astoria exposes the entities in Azure Tables as URI-addressable resources that can be accessed using standard HTTP requests. Data can be manipulated by either using the client-side libraries of Astoria, or by using the REST interface directly. Using the libraries is dramatically easier, because it takes care of the issues involving communicating with a remote data source and allows the Table service to be queried using LINQ, .NET's native query facility. However, there is one important drawback: Astoria assumes the data has a fixed schema, and can therefore only manipulate data based on a pre-determined schema. Azure Tables does not have a fixed schema, but instead it has a flexible data model. When the Astoria client-side libraries are used, these will enforce a fixed schema. The flexibility will be lost, which means the data model cannot be used optimally.

Currently, if this flexibility is a requirement for an application, the only option remaining is to use the REST interface directly. This places the burden of having to deal with remote communication squarely on the programmers of the application in question, causing a lot of extra code to be designed, written, tested and maintained. A reliable data access layer that takes care of these issues would thus relieve these programmers of this work.

This project will describe the design and implementation of a data access layer for Windows Azure Tables, with which the flexibility provided by the Tables' data model can be leveraged from within a .NET application. This data access layer should, like Astoria, enable data modeling using .NET objects and querying using LINQ, without imposing a fixed schema on the data. This access layer will illustrate how the mismatch between fixed data models in programming languages and flexible data models in Cloud databases can be overcome.

1.1 Research Question(s)

The research questions investigated in this document are:

- How can the flexibility of Cloud databases be leveraged from a language with fixed classes?
- How can such a system be implemented for Windows Azure Tables, using C#?

Chapter 2

Cloud Computing

In order to build a web application, there are a lot of things to think about. For example, the application needs some servers to run on. These servers need an operating system that has to be kept up to date. Then the application requires a network, including load balancers, routers and DNS, and enough bandwidth. The application will also need some facility for data storage and management. And last but not least, scaling needs to be considered; when the application is successful, how is this infrastructure going to handle a growing number of users? All of these concerns apply universally to all web applications, but none of them have anything to do with the actual goal: building a web application.

What if it was possible to delegate the concernes mentioned in the previous paragraph to specialists, so the focus can be on building the application? This is what Cloud computing means to achieve.

2.1 What is Cloud computing?

What is generally referred to as Cloud computing is a combination of a computing grid, virtualization, and dynamic scaling, accessible over the Internet. Like many technology buzzwords, there are several definitions and descriptions going around for it. This chapter will describe what Cloud computing is, and how and why it is beneficial.

”Cloud Computing refers to both the applications delivered as services over the Internet and the hardware and systems software in the datacenters that provide those services. The services themselves have long been referred to as Software as a Service (SaaS) (...). The datacenter hardware and software is what we will call a Cloud.” [4] The phrase ”Software as a Service” refers to a paradigm shift in software over recent years, where instead of selling or licensing software to be installed on a client’s own hardware, the software is hosted in a datacenter and is accessible to the client over the Internet. This has several benefits for both the producers and the consumers of a service. For the producer, installation, maintenance and version control become much simpler. The consumer, in turn, has access to the service everywhere an Internet connection is available, while their data is stored safely.

Cloud computing makes it possible for a SaaS provider to focus on developing good software, without having to worry too much about the hardware to support it. The hard-

ware issues, including scalability, are out-sourced to a Cloud computing vendor. Service providers pay only for the storage and computing time actually used. This is especially advantageous for new services whose popularity is not yet established, or for services which experience large differences in the amount of traffic they receive over time, since it is no longer necessary to plan ahead for peaks in traffic, and no large up-front investments are necessary. Other hardware-related issues such as maintenance, network connectivity and disaster recovery are all taken care of by the Cloud vendor.

The typical Cloud computing vendor is a company which has an established name and experience managing large data centers, possibly for their own operations. "Adding Cloud computing to an already existing infrastructure provides a new revenue stream at (ideally) low incremental cost."[4] Cloud computing vendors can build their data centers in locations where conditions such as the cost for electricity, bandwidth, labor and taxes are favorable. Since resources like electricity and bandwidth can be bought in bulk, this can result in enormous savings.

2.2 Advantages

Cost Cloud computing is basically the out-sourcing of hardware. Instead of paying the up-front cost hardware and hiring maintenance personnel, one pays only for the actual usage of servers, based on computing time or the volume of data handled. Because Cloud computing vendors can purchase hardware, bandwidth and power in bulk, they can offer services at a premium that is unattainable for most companies. For a medium sized company, it has been estimated that they can save a factor of 5 for their infrastructure. [4]

Accessibility Having your applications and data hosted in the Cloud means that they are accessible wherever an Internet connection is available.

Scalability One of the most innovative features of Cloud computing is its ability to scale. Not only does a user have virtually unlimited computing power when the service requires it, the amount of computing power used can be adjusted dynamically, up or down, to follow the requirements of the user. This also makes using Cloud computing very cost effective.

Focus on the software Because all the infrastructure is handled by the Cloud computing vendor, software companies can focus on what they are (supposedly) good at: building a great application.

2.3 Disadvantages

Switching technologies Converting an existing application to one that is suitable for Cloud computing can be a big investment. Applications are often built with a specific stack of supporting software (like the operation system, database system and programming platform) in mind. Changing this stack will often require large changes in the software itself.

Data storage and the law Certain data is required by law to be stored in certain locations. For example, banks in the European Union are required to store their data within the European Union. Highly sensitive data is also unlikely to be stored in way in which its owners have less than total control of the location and security of the data, which is typically the case when using Cloud computing.

Vendor lock-in At this moment, there are no standards for exactly what a Cloud computing vendor offers or how one would interact with it. This means every Cloud computing vendor has their own facilities and programming platforms. Building an application using Cloud computing vendor A is likely to be different to building one using vendor B. This means moving applications between different vendors is a complicated and costly matter. Once a vendor has been chosen, that choice is tough to reconsider.

2.4 Other aspects

Data centers The actual computing involved in Cloud computing, along with data storage, is done in large data centers. This means that this data is probably safe from things like fires or theft. On the other hand, the sheer volume of possibly valuable information does make the a potential target for hackers or terrorists. Cloud vendors are very aware that their success depends on their ability to keep their data centers up and running and secure, so every possible measure is taken to ensure availability 24-7. However, when an outage does occur, a client can do little else than report the problem and wait for the Cloud computing vendor to solve it.

Environmental issues Data centers can be built near power plants, which result in higher efficiency. Power efficiency is a concern for both vendors and clients. For vendors, power consumption is a significant part of the cost to operate a data center. For clients, efficient use of the resources is critical, since less resources used translates directly in lower cost.

General versus custom software By providing SaaS applications, software companies can offer the same application to many clients. This has several benefits. The code will only have to written and tested once, and maintenance and updates can be administered easily for all clients. This eliminates the need to support multiple versions, as wel as the need to administer the software on clients' own hardware. On the other hand, it becomes more difficult to provide customizations for specific clients, which is an important source of income for some companies.[6]

Chapter 3

Cloud Database

By far, most databases used in the past 30 years are SQL databases. These are based on the relational model. Cloud databases, however, are generally not. The remainder of this chapter will discuss how and why cloud databases differ from SQL databases and the relational model.

3.1 The relational model

The relational model was developed in 1969 by Ted Codd and described in a now classic paper in 1970[7]. It was a departure from the then widely used hierarchical and network database models, providing a simple model with a strong mathematical foundation. The relational model takes its name from the mathematical relation, the cornerstone of the model. A relation is often visualized as a table of values. A row in such a table is a collection of related values representing a real world entity or relationship, called a tuple. Every tuple in a relation is of the same type and has the same set of attributes, represented as the columns of the table.

An important concept in the relational model is normalization. Normalization aims to ensure that data is stored in one and only one place, by removing dependencies. Duplicate and interdependent data is dangerous, because an update to one of the occurrences of a certain datum, but not to any other occurrences, can lead to discrepancies. Whenever two or more tables share information, normalization generally resolves this dependency by creating one or more new relations representing the dependencies. This new relation will contain a tuple for the shared data which can be referred to by the other tuples using its key. In this way, the data can be maintained in a single location. However, it also causes data concerning a single high-level entity to be fragmented over several tables. When the entity is later being retrieved, these fragments need to be recombined at runtime in an operation called a join. So, while normalization greatly simplifies insertion and updates in the database by preventing inconsistencies, it incurs extra work when data is queried.

3.2 Considerations for the cloud

For a long time, relational databases have been the de facto choice for storing any sort of structured data. However, as the size of databases grew along with Moore's Law[1], some limitations became apparent. Storing massive amounts of data in a relational database is no problem. Querying, however, becomes a tougher problem as the haystack in which to search for the needle grows. Specifically, queries involving joins become more costly, since joining takes relatively more work for larger amounts of data. Query execution times can grow enormously due to joins if the involved tables are very large or if many tables are involved. In simple terms, joining two tables *A* and *B* on an attribute *x* involves searching for the relevant value of *x* in *B*, *for every tuple in A*. In modern SQL DBMSs it is possible to do these kinds of joins far more efficiently than the O(n*m) operation mentioned in the previous sentence. By using indexes, hash functions and the physical sort order of the tables in question, execution times can be reduced to almost O(n) in certain cases. However, these techniques require extra disk space and they have their own limitations: indexes can only be defined on a few attributes per table, and a table can only be physically sorted on a single attribute. The loss of query performance is particularly problematic when query operations are far more common that insert and update operations, as they are in most web applications.

As companies started running into this limitation of the relational model, they began to search for solutions. One such solution is to use databases which store much-requested data in a denormalized fashion, eliminating the need for joins in common requests. Instead of denormalizing specific tables, another approach is to provide an entire separate database where oft-queried tables are presented denormalized. The real database will still store all information in the regular, normalized way. This is also the database where any changes will be applied. These changes will then be synchronized to the published, denormalized database, which is itself read-only. This approach gives the best of both worlds (consistent data and fast querying) at the cost of the hardware necessary for the second database. In a time where hardware is quickly becoming the cheapest part of the IT equation, this becomes a very viable alternative.[11]

In a few cases, however, this was not enough. Most notably, Google and Amazon, both of which have to maintain massive volumes of data for their businesses, abandoned the relational model completely and decided to each develop their own database system, aiming at extreme scalability and availability.

3.3 Cloud Databases

Google and Amazon chose to radically depart from the classic relational model for data storage in their Cloud platforms[1, 2, 5]. Instead of using a schema of normalized tables, these databases store data as collections of key-value pairs in one or more heterogeneous tables. Every entity can have its own set of attributes, regardless of the table that contains them. Joins between tables are generally not supported, which means that retrieving re-

[1] Intel co-founder Gordon Moore stated in 1965 that number of transistors on a chip would double roughly every two years.[19]

lated data has to be done either by executing multiple queries, or by storing the data in an denormalized way (or a combination of both).

There are several reasons why these databases depart form the relational model. One of those is the poor performance of joins in databases with the massive amounts of data these systems are designed for. Secondly, cloud databases are generally distributed databases, spread over data centers around the world. Joining tables which are distributed over multiple servers (possibly in different geographical locations) is far more complex and costly than joining local tables. With the key-value pair approach taken in Cloud data models, it is easier to implement a scalable database. Partitioning and clustering data to be spread over many distributed servers is easier with flexible entities than with a fixed relational schema.

Because cloud databases are very different from other database systems, there are a number of considerations that need to be taken into account when using them. When using a non-normalized database, suddenly one has to consider issues like consistency, integrity, concurrency and performance in way that simply does not come up in a normalized relational database. The relational model was designed with consistency and integrity in mind and does a wonderful job at relieving users from having to worry about these issues. Concurrency and performance optimization are generally handled by the DBMS. When using a Cloud database, these issues are no longer trivial. Without joins, interrelated data has to be handled differently. Denormalizing the data, for example by inlining data, becomes necessary. However, there is no pre-defined way in which a normalized data model should be denormalized for storage in a Cloud database. There are many variables that play a role in determining the optimal approach. It all depends on how the data is used. It the data primarily read and seldom written? How many entries exist of a certain type? How many dependencies are there, and how strong are they? These and other aspect determine they way data handled in a Cloud database, whereas they are of little interest when a relational database would be used.

3.4 Denormalized data storage

In relational databases, denormalization is an optimization technique; query performance in increased at the cost of complicating inserts and updates. In a non-relational Cloud database, denormalization is unavoidable. Because joins are not supported, combining fragmented data becomes an even more expensive operation than in relational databases.

The data model of Cloud databases is also very different from that of relational databases. Where relational databases have a rigid schema every entity in the database must conform to, Cloud databases are typically flexible in the entities they can store. It is still possible to store data in a Cloud database in a normalized way, fragmenting data over several data types, each of which has its own table, provided the system allows multiple tables. (This is similar to what Astoria does when combined with Windows Azure Tables, actually. More on that in chapter 4.) Retrieving this data would be inefficient. Because of the lack of joins, fragmented data cannot be recombined by the system. Instead, it would be done by the application requesting the data. The application would have to send several queries to get a single high-level data entity. In the worst case scenario, the application will end up

retrieving (possibly huge amounts of) data that it ultimately does not need.

The flexibility provided by the data model of Cloud databases can be used to store the data in a denormalized or unnormalized way. This flexibility allows entities with different properties in the same table. Entities can therefore define properties which are applicable to some, but not all entities of a certain type, without negative side effects. (In a relational database, this would lead to a large number of NULL values and a waste of space.) Consequently, properties that would, in a relational database, be stored as a different entity in a different table, can now be easily combined into a single entity, without the need for specifying all possible properties in a schema.

Tables can even contain completely different entity types. And since tables define a scope for queries, the tables in a cloud database can play a role less like tables in a realtional database, and instead more like the database as a whole.

Chapter 4

Windows Azure

Windows Azure is Microsoft's platform for cloud computing. Announced at the Professional Developers Conference in October 2008, it's currently in CTP[1] and will be commercially available at the PDC 2009. Windows Azure provides a Cloud computing platform with an abstraction level which is in between the hardware virtual machines of Amazon EC2, and the application-specific frameworks like Google AppEngine (web applications) and Force.com (customer relationship management using the Salesforce.com platform). Applications for Azure are written using the libraries and any of the languages of the .NET framework. The remainder of this chapter will describe the fundamentals of Windows Azure Storage and the ADO.NET Data Services implementation of Azure Tables, both through the use of the client libraries and the REST interface.

4.1 Windows Azure Storage

There are two distinct data storage facilities for Windows Azure: Azure SQL, which offers relational storage in the cloud, and Azure Tables, which offers a simpler data model, focusing on high scalability.

Azure SQL, formerly SQL Data Services, is a subset of Microsoft SQL Server, which is what is uses as a backend. It uses T-SQL as the query language, and TDS[2] for transferring its data. It was the first relational database for the Cloud. Azure SQL databases are replicated multiple times in different locations to ensure performance and fail-over safety.

Windows Azure Storage Service is designed to be the lowest cost, most efficient solution for large scale data storage and retrieval in the cloud. It consists of three parts:

- BLOBs, used for storing large files
- Queues, used for messaging
- Tables, used for storing large amounts of structured data

[1] CTP, or Community Technical Preview, means that the software has not been officially released, but Microsoft partners are given a preview of the software so they can get acquainted with it.

[2] Tabular Data Stream, a network protocol developed specifically for transferring relational data between systems.

ADO.NET type	CLR type	Description
Edm.Binary	byte[]	An array of bytes op to 64 KB in size
Edm.Boolean	bool	A Boolean value
Edm.DateTime	DateTime	A 64-bit value expressed as Coordinated Universal Time (UTC)
Edm.Double	double	A 64-bit floating point value
Edm.Guid	Guid	A 128-bit globally unique identifier
Edm.Int32	Int32	A 32-bit integer
Edm.Int64	Int64	A 64-bit integer
Edm.String	String	A UTF-16 encoded value up to 64 KB in size

Table 4.1: Azure Tables data types

Azure Tables is not a relational database system, like most other Cloud databases. Instead, the data model used in Azure Tables is similar to that of SimpleDB, Amazon's Cloud database. Entities are organized into tables. An entity is a collection of key-value pairs. Unlike SimpleDB, where all values are strings, values in Azure Tables can be of one of a variety of types, described in table 4.1. Another difference between the two data models is that in SimpleDB, it is allowed to have multiple values for an attribute. The values are differentiated by a timestamp, allowing the storage of temporal (time-dependent) data. Azure Tables allows only a single value for an attribute.

There is no schema to which an entity must conform; entities with different properties can be stored in the same table. A table does not define the type of entity that can be stored in it, but merely define a scope for queries. Every entity must have the following three properties: `PartitionKey`, `RowKey`, and `Timestamp`. The partition key and row key, which are both strings, together uniquely identify an entity within a table. The partition key defines how entities in a table can be distributed over multiple storage nodes. Entities with the same partition key are guaranteed to be on the same node. Operations on entities which are in the same partition are therefore faster than operations that span several partitions. The timestamp value is used internally to provide optimistic concurrency.[15]

As in other Cloud databases, joins are not supported in Azure Tables. Structured data should not be stored in a normalized way in Azure Tables because of the trouble of recombining the normalized data when it is queried. When data is required to be stored in a structured and normalized fashion, Azure SQL should be used.

4.2 ADO.NET Data Services

Windows Azure Tables implements Microsoft's new data philosophy for the Web, called ADO.NET Data Services, also known as Astoria. The ADO.NET Data Services framework is Microsofts technology for creating and consuming data services. It is an effort from Microsoft to decouple the presentation and data of web applications. It allows applications to have an easily accessible data layer, independently of the presentation. This should make it easier to build web 2.0 applications with AJAX-style interfaces with a powerful back-end.

As one of its building blocks, ADO.NET Data Services provides a uniform URI format for addressing resources and uses standard HTTP requests, through the use of a REST interface,[3] and simple data formats to manipulate them.[10]

These data services use the Entity Data Model (EDM) also used in ADO.NET Entity Framework. EDM is based on the entity-relationship (ER) model which models data in terms of entities and relationships.

Accessing the data stored in Azure Storage can be done using either the client-side libraries for Astoria, or by using the REST interface directly. By using the client, entities can be defined as CLR objects, which define a schema for Astoria to work with. Entities stored using this pattern can be queried using LINQ[4], which will return a collection of CLR objects satisfying the query. These objects are then associated with a context, which maintains the link between an Azure Tables entity and an in-memory object. These objects can then be manipulated in the application just as any other object. To save any changes to these objects to the data store, the object in question need to be marked as changed, and then the context is asked to save the changes to the data store.

Using the client saves programmers a lot of work by taking care of the communication with the data service and managing the entities in-memory. However, there is an important drawback to using the client. Since Astoria works with fixed schemas in the form of Entity Data Models, the client cannot handle the flexible data model used in Azure Tables. The client enforces the use a fixed schema, defined as CLR object classes. This severely limits the possibilities of Azure Tables, particularly when several applications work with the same data. These applications are forced to use the exact same class library as all the other, lest there arise conflicts over the data types. This is even more of a burden when not all the applications are based on the .NET platform, since they will have top conform to a schema defined in a ”foreign” language. Being able to deal with entities in a more natural way would be beneficial in these circumstances.

4.3 Under the covers

The ADO.NET Data Services client libraries communicate with the Table service using a REST interface. Entities can be addressed using a URI, and manipulated using HTTP methods. Table 4.2 shows the correspondence between HTTP methods and operations on the Table service. The RESTful interface can also be used to work with the Table service directly, without using the client libraries. This is necessary when Azure Tables is accessed from languages not part of (or built on) the .NET framework. It also allows for some added flexibility, since the use of an EDM schema is not required.

[3] REST stands for REpresentational State Transfer. It basically means a service has a client-server architecture and uses the HTTP protocol for communicating with its clients

[4] LINQ, or language integrated query, enables native data querying abilities to the .NET Framework. Using LINQ, programmers can query arrays, objects, databases and XML files using a uniform, SQL-like query syntax. More on that in chapter 5.

Operation	HTTP method	Resource	Description
Query	GET	Entity, Table	Retrieve selected resources
Insert	PUT	Entity, Table	Create a new resource
Full update	POST	Entity	Replace an entire entity
Partial update	MERGE	Entity	Update property values within an entity
Delete	DELETE	Entity, Table	Remove the resource

Table 4.2: Operations in Azure Tables

4.3.1 Addressing resources

The base URI for Table service resources is the storage account:

```
http://[account name].table.core.windows.net
```

Table names are also stored in a dedicated table. To list the tables in a given storage account, to create a new table, or to delete a table, operations can be performed on this table, conveniently called ”Tables”, using the URI:

```
http://[account name].table.core.windows.net/Tables
```

To retrieve a single table, it can be named within the collection of tables, as follows:

```
http://[account name].table.core.windows.net/Tables('[table name]')
```

To query entities in a table, or to insert, update, or delete an entity, the following syntax is used to refer to the set of all entities in the named table:

```
http://[account name].table.core.windows.net/[table name]()
```

Individual entities are identified by the combination of their partition key and row key. A specific entity can be addressed as follows:

```
http://[account name].table.core.windows.net/
    [table name](PartitionKey='[partition key]',RowKey='[row key]')
```

4.3.2 Querying Azure Tables

Tables can be queried by using the `$filter` or the `$top` query options defined in Astoria. The other query options defined in Astoria are not supported. The `$top` option limits the number of entities to be returned, for example:

```
http://[account name].table.core.windows.net/[table name]()?$top=10
```

This would return the first ten elements of the table, as sorted by the partition key and row key values. The value must be 1000 or lower, since a query to the Table service always returns a maximum of 1000 results at a time. (This behavior is not standard in Astoria, but specific to Azure Tables.) If there are more than 1000 results, the query response will

Operator	URI expression	Description
==	eq	equals
>	gt	greater than
=>	ge	greater that or equals
<	lt	less than
<=	le	less than or equals
!=	ne	not equal
And	and	Boolean And
Or	or	Boolean Or
Not	not	Boolean Not

Table 4.3: Query operators for ADO.NET Data Services

include continuation tokens as custom HTTP headers. The continuation tokens identify the next result that satisfies the query and can be used to construct a follow-up query, which will return the next 1000 results. The `$filter` query option is followed by a query expression to which the returned entities must conform. Table 4.3 describes the operators defined for the Table service. For example:

```
http://[account name].table.core.windows.net/[table name]()?$filter=
    IntValue ge 51 and
    StringValue eq 'text'
    and BoolValue eq true
```

4.3.3 Entity encoding

Entities sent to or received from the Table service are encoded as entries in the body of the HTTP messages in an Atom feed, a standard XML-based publishing protocol. The entities are thus represented using XML. Listing 4.2 shows an example of the Atom representation entity returned by a query, listing 4.1 shows that of an insert operation.

4.3.4 Concurrency

Every entity returned comes with an e-tag. This is a value used for providing optimistic concurrency. Whenever an entity in Azure Tables changes, its e-tag changes too. With every update or delete operation, the e-tag of the retrieved entity must be sent with the request. If the e-tag matches the e-tag in the Table service, everything is fine. If they differ, this means that the entity has changed in the Table service since it was retrieved, which might invalidate the update or delete operation. In this case, the operation will fail, and the entity in question will have to be retrieved again to complete the operation.

4.3.5 Authentication

All operation performed on the Table service must be authenticated. The Table service supports two authentication schemes: Shared Key, and the similar but simpler Shared Key

Listing 4.1: An example of an entity to be inserted

```
<entry xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices" xmlns:m="http://
     schemas.microsoft.com/ado/2007/08/dataservices/metadata" xmlns="http://www.w3.org
     /2005/Atom">
  <title />
  <updated>2009-11-04T17:13:11.7541410Z</updated>
  <author>
    <name />
  </author>
  <id />
  <content type="application/xml">
    <m:properties>
      <d:BodyShape>Dinky</d:BodyShape>
      <d:NoOfStrings m:type="Edm.Int32">6</d:NoOfStrings>
      <d:NoOfFrets m:type="Edm.Int32">24</d:NoOfFrets>
      <d:Color>Transparant Blue</d:Color>
      <d:Price m:type="Edm.Int32">55000</d:Price>
      <d:Description />
      <d:Category>Guitar, electric</d:Category>
      <d:PartitionKey>Jackson</d:PartitionKey>
      <d:RowKey>DK2</d:RowKey>
      <d:Timestamp m:type="Edm.DateTime">0001-01-01T00:00:00.0000000</d:Timestamp>
      <d:Bridge>Floyd Rose licenced</d:Bridge>
    </m:properties>
  </content>
</entry>
```

Listing 4.2: The same entity as returned by a query

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<feed xml:base="http://phaedrus.table.core.windows.net/" xmlns:d="http://schemas.microsoft
     .com/ado/2007/08/dataservices" xmlns:m="http://schemas.microsoft.com/ado/2007/08/
     dataservices/metadata" xmlns="http://www.w3.org/2005/Atom">
  <title type="text">Products</title>
  <id>http://phaedrus.table.core.windows.net/Products</id>
  <updated>2009-11-04T17:13:36Z</updated>
  <link rel="self" title="Products" href="Products" />
  <entry m:etag="W/&amp;quot;datetime'2009-11-04T17%3A13%3A36.1310819Z'&amp;quot;">
    <id>http://phaedrus.table.core.windows.net/Products(PartitionKey='Jackson',RowKey='DK2
         ')</id>
    <title type="text"></title>
    <updated>2009-11-04T17:13:36Z</updated>
    <author>
      <name />
    </author>
    <link rel="edit" title="Products" href="Products(PartitionKey='Jackson',RowKey='DK2')"
          />
    <category term="phaedrus.Products" scheme="http://schemas.microsoft.com/ado/2007/08/
         dataservices/scheme" />
    <content type="application/xml">
      <m:properties>
        <d:PartitionKey>Jackson</d:PartitionKey>
        <d:RowKey>DK2</d:RowKey>
        <d:Timestamp m:type="Edm.DateTime">2009-11-04T17:13:36.1310819Z</d:Timestamp>
        <d:BodyShape>Dinky</d:BodyShape>
        <d:NoOfStrings m:type="Edm.Int32">6</d:NoOfStrings>
        <d:NoOfFrets m:type="Edm.Int32">24</d:NoOfFrets>
        <d:Color>Transparant Blue</d:Color>
        <d:Price m:type="Edm.Int32">55000</d:Price>
        <d:Description></d:Description>
        <d:Category>Guitar, electric</d:Category>
        <d:Bridge>Floyd Rose licenced</d:Bridge>
      </m:properties>
    </content>
  </entry>
</feed>
```

Lite. The Shared Key scheme is supported across all parts of Windows Azure Storage: BLOBs, Queues and Tables. The ADO.NET Data Services client libraries use the simpler Shared Key Lite scheme. Both schemes are based on a hash created from a string containing several pieces of information about the request, including a key value associated with the storage account. The exact algorithm for obtaining the value to sign a request with can be found on MSDN[17].

4.4 The Storage Client

As described in section 4.2, there are a number of differences between the Astoria client libraries and the Table service. The Windows Azure SDK includes a sample application called ”StorageClient”, which helps users by tying up some of the loose ends. It provides a number of classes which deal specifically with the issues concerning Azure Tables. These classes represent concepts from Azure Storage and the Astoria libraries to implement their behavior. This resolves a number of differences between Astoria and Azure Storage and make it easier for users to work with Azure storage through Astoria. However, some differences (like the use of a fixed schema) remain.

4.5 Development Storage

The Windows Azure SDK development environment includes development storage, a utility that simulates the blob, queue, and table services available in the Cloud.[18]. The development storage helps users with building and testing of applications which use the storage facilities. The development storage for Azure Tables is simulating the Table service in a local SQL Server database. Unfortunately, the functionality of the development storage does exactly mirror that of Windows Azure Storage. Most notably, the development storage for Tables requires the use of a fixed schema. Consequently, any application which utilizes the flexible nature of Azure Tables cannot be simulated in the local development storage. This, in combination with the other limitations of the local development storage, makes that using the development storage for building and testing applications which use Azure Tables only works when this application also uses the Astoria client libraries and the Storage Client. If there is any reason for using another access method, this will most likely also rule out using the development storage. The blob and queue parts of the development storage more closely resemble their implementations in the Cloud, but for using the tables (and hence, for this project), the usefulness of the development storage is fairly limited.

Chapter 5

LINQ

Dealing with data is a large part of programming. Hardly an application exists that does not, at some point, need to store, retrieve or manipulate pieces of data. These pieces of data can take a multitude of forms, such as tables in a relational database, programming constructs, XML documents or file systems. All these different forms of data have their own applications, along with their own methods of accessing and manipulating the data that they represent. Obviously, this leads to a large number of data access methods, each with their own learning curve.

LINQ provides an easy and uniform way to handle data using the .NET framework. Languages that target the .NET framework can also implement a number of language extensions which simplify queries and make them a first class language construct, integrating them with the programming language (LINQ stands for Language INtergated Query). Almost all the features introduced with C# 3.0 were aimed at enabling LINQ.

With LINQ, one can query a variety of data sources using a uniform, SQL-like syntax. However, the usage of LINQ is not limited to actual databases. The .NET framework includes several so-called LINQ providers which, as the name suggests, provide LINQ support for other data sources. These include LINQ-to-Objects, for querying objects and collections in memory, LINQ-to-XML, LINQ-to-Datasets (for querying ADO.NET datasets), LINQ-to-Entities (for working with ADO.NET Entity Framework) and, of course, LINQ-to-SQL. Queries to any of these data sources can be expressed uniformly in the user's .NET language of choice (in case of this project: C#) where it has the benefit of type checking. The LINQ provider translates the query into something specific for the data source, where it is eventually executed. The results are returned as CLR objects, and as such, they can easily be inspected and manipulated. Any data source can be made accessible via LINQ by

SQL	LINQ
`select Name`	`from e in Employees`
`from Employees`	`where e.Age > 40`
`where Age > 40`	`select e.Name`

Table 5.1: Two equivalent queries in SQL and LINQ syntax

Query expression	Method call
`from e in Employees`	`Employees`
`where e.Age > 40`	`.Where(e => e.Age > 40)`
`select e.Name`	`.Select(e => e.Name);`

Table 5.2: Query expression syntax versus method call syntax in C#

implementing a LINQ provider for it.

LINQ is fundamentally about sequences of data, or enumerables, as they are called in .NET. An enumerable, in this context, is any object that implements the `IEnumerable` interface.[1] A LINQ query expression is basically a refinement of a sequence, getting more specific from top to bottom.[2] For example, the query in figure 5.1 starts with the sequence containing all the elements in the `Employees` collection. The `e` in the first line is the declaration of a so-called range variable, which represents one element of a particular sequence. This range variable is used in the query to specify the element being processed. The `where` clause refines the sequence to only those employees that are over 40. Finally, the `select` clause transform the sequence from one containing employees to one that contains only their names. An important aspect of LINQ is that when a query expression is created, no data is actually being processed yet. Instead, an expression tree is being formed in memory which represents the query. Only when the query result is being enumerated –in other words, asked for the first element of the sequence –the data processing starts. This concept is called deferred execution and lies at the heart of LINQ. The data is loaded and processed one element at a time, instead of a complete collection. This means that generally, little or no time is wasted processing data that ultimately will not be used. This dramatically improves LINQ's performance. [3]

The friendliest way of specifying LINQ queries is by using the query expression available in C# and VB.NET. These are the SQL-like statements as presented in figure 5.1. A query expression consists of a number of query operators, most notably `select` and `where`, and their arguments. These query operators are implemented as static extension methods[4] which work with the `IEnumerable` interface.

Before compilation, a query expression is translated into a very similar looking sequence of extension method calls (illustrated in table 5.2) on an object which represents a data source and which implements the `IQueryable` interface. This interface extends the `IEnumerable` interface, but an object implementing the `IQueryable` interface is more pow-

[1] The `IEnumerable` interface itself is rather basic; it mostly requires the object implement a method that returns an `Enumerator` object. This is the object that does the actual enumerating: returning the elements of the `IEnumerable` object one by one.

[2] The main reason the LINQ query syntax deviates from SQL is that by putting the `select` clause at the end, this top-to-bottom argument holds.

[3] Some query operations, such as sorting, cannot benefit from deferred execution, because they require the whole collection of data to be loaded in order to return the first element.

[4] Extension methods are one of the language extensions that support LINQ. They are static methods which can be called as if they were defined by the type they operate on. As such, they *extend* the behavior of the type in question, hence the name.

erful than a regular enumerator, because it can inspect and modify the query expression that is passed to it. This will be covered in more detail in section 5.1. As a consequence, LINQ provider can take a LINQ query and translate it to query for the specific data source.

5.1 Under the covers

Since this document is about building a LINQ provider, it is necessary to delve deeper into the inner workings of LINQ. A LINQ provider consists of the implementation of a number of interfaces which together provide the functionality necessary to execute the LINQ queries. The interfaces to implement are `IQueryable<T>` (or `IOrderedQueryable<T>` when ordering is supported), which represent a query to a data source, and `IQueryProvider`, which processes they query for a particular data source.

`IOrderedQueryable<T>` is a simple interface consisting of three properties:

Listing 5.1: The IOrderedQueryable interface

```
Type ElementType;
Expression Expression;
IQueryProvider Provider;
```

The first one gives the element type (the `T` in `IOrderedQueryable<T>`), denoting the data type of the results that will be returned. The second gives the expression of the query. This is the actual query to be executed, in the form of an expression tree.

The `IQueryProvider` interface has the following methods:

Listing 5.2: The IQueryProvider interface

```
IQueryable CreateQuery(Expression expression);
IQueryable<TElement> CreateQuery<TElement>(Expression expression);
object Execute(Expression expression);
TResult Execute<TResult>(Expression expression);
```

`CreateQuery` and `Execute` (both of which exist in a generic and a non-generic flavor) do exactly what their names suggest. Specifically, `CreateQuery` creates an `IQueryable` for the expression provided to it. This is the method that is used for queries that return a collection of elements. Such queries use the concept of deferred execution described earlier: only when its results are enumerated, the gears start turning. The `Execute` method is used for queries that return a single result. Such queries do not use deferred execution; they are executed immediately. The `Execute` method is also used by the `IQueryable` object once it is being enumerated.

An `IQueryable` represents a query in LINQ. The `IQueryable` interface extends the `IEnumerable` interface and has two additional properties for an expression tree and a LINQ provider. The expression tree is an in-memory representation of the query expression, similar to an abstract syntax tree in a compiler. The `Provider` property indicates the LINQ provider associated with the query. When the `IQueryable` is asked for its results, this invokes the `Execute` method for the provider, which can analyze the expression tree for the query and translate it into a query specific for the data source. This translated query is then executed in the data source. The resulting CLR objects can be retrieved, one by one, by enumerating the `IQueryable`.

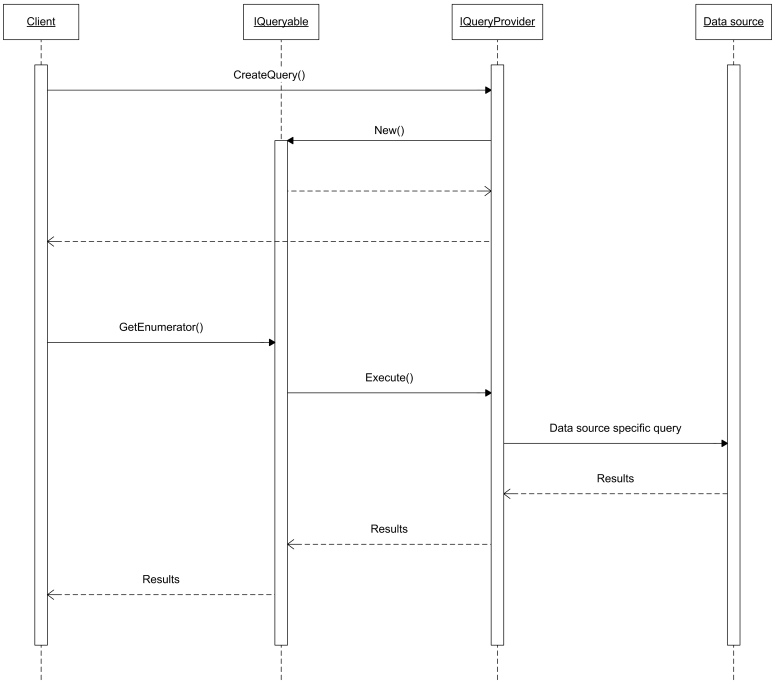

Figure 5.1: The sequence of operations involved in a typical LINQ query

Figure 5.1 illustrates the sequence of operations in a general LINQ query. An application first creates a new query and along with a new query provider. The query includes the query expression in the form of an expression tree. Once the query is asked for its result (by calling the `GetEnumerator` method) the query asks the associated query provider to `Execute` its query expression. The query provider will then translate the expression tree to a query for the specific data source and interpret the results. It returns an `IEnumerable` to the query, which will in turn return it to the application, which can then iterate through it.[16, 20]

Chapter 6

Flexible access to Azure Tables

As described in section 4.2, the currently available methods of working with Azure Storage Tables force the user to choose between the convenience of using the client-side libraries of Astoria and the flexibility provided by the Tables. Choosing for the flexibility means a lot of code will have to be designed, built and maintained for handling communication with the storage service. The goal of this project is to alleviate the user from having to worry about the details of dealing with the data service, by providing a data access layer similar to the Astoria client libraries, but capable of dealing with the flexibility of entities with different properties. Since the actual data access will be handled by a LINQ provider, this data access layer is called LINQ-to-Azure. This data access layer will consist of two parts: a LINQ provider for querying Azure Tables, and a runtime where entities can be manipulated as CLR objects, which can then be saved to Azure Tables. The LINQ provider takes care of translating LINQ queries into equivalent Table service queries. This is a crucial part of the software, but makes up only a modest part of the code. The rest of the infrastructure is concerned with everything else; creating and modifying entities in memory, translating between .NET objects and Atom representations, and creating and handling the HTTP traffic, including authentication. Both the public interface and the architecture of the system resemble those of the Astoria client libraries.

The remainder of this chapter will describe the design and implementation of LINQ-to-Azure, in two parts. The first part will describe the framework for inserting, updating and deleting entities in Azure Tables, and tracking them at runtime. The second part will focus on querying Azure Tables using LINQ and the implementation of the actual LINQ provider.

6.1 Managing entities in memory

The `AzureTablesContext` class is the central class of LINQ-to-Azure. It basically provides a .NET API for Azure Tables; all operations which are supported by the Table service can be performed by using methods in the `AzureTablesContext`. This is also the class that tracks the entities in memory. The object tracking resembles that of the `DataServiceContext` class in Astoria, except it is a bit simpler. Astoria defines a number of concepts that are not supported by the Table service, hence they are not included in LINQ-to-Azure. For

Category	Method name	Description
Tracking	AddObject	Add an object to be inserted in a table
	AttachTo	Track an object which has been returned by a query
	CreateTable	Create a new table
	DeleteObject	Mark an object for deletion
	DeleteTable	Mark a table for deletion
	Detach	Stop tracking the object
	SaveChanges	Save the tracked changes to the Table service
	UpdateObject	Mark an object as changed, to be updated
Querying	CreateQuery< >	Create an `IQueryable` for a table
	Execute< >	Execute a Data Services query
	GetEntity	Retrieve an entity based on its keys
	GetTableNames	Get the names of the tables in the account
	Query< >	Execute a Data Services query
	TryGetEntity< >	Retrieve an entity by its URI

Table 6.1: The public methods of the AzureTablesContext class

instance, Astoria's data model is based on the entity-relationship model, so it needs to maintain both the entities and the relationships (or links, as they are called in this context). Azure Tables has only entities and no links. Consequently, there is no support for links in LINQ-to-Azure. The `AzureTablesContext` class also provides some functionality present in the Storage Client sample, described in 4.4. Table 6.1 describes the public methods of the `AzureTablesContext` class. The methods in the tracking category are the same as those found in Astoria's `DataServiceContext` class, with the exception of the explicit methods for creating and deleting tables.[1] Internally, these methods operate on a collection which maintains the operations that have been performed on the entity objects. Once the `SaveChanges` method is called, these operations are transformed into the HTTP messages which are sent to the Table service in order to actually make the changes. The order of these operations is important. For example, a `CreateTable` operation should be executed before attempting to store any entities in it.

The part of the `AzureTablesContext` class that enables querying is a little different from Astoria. The `AzureTablesContext` class, like `DataaserviceContext`, defines the following methods which deal with queries: `CreateQuery< >`, `Execute< >` and `TryGetEntity`. The `CreateQuery< >` method returns an `IQueryable` for a table whose name is specified as a parameter, and whose generic type is indicated by the type parameter of the method. The `Execute< >` method takes a URI which denotes a Data Service query and executes it, returning an `IEnumerable` with the results. This is the method that is called by the LINQ provider when a query is enumerated, but it can also be called directly, as long as the URI is a valid Data Services query. The `TryGetEntity< >` method retrieves

[1] When Azure Tables are used through the Astoria client libraries, creating, deleting and querying tables can be done by manipulating the "Tables" table, or by using methods provided in the StorageClient application (see section 4.4).

an entity based on its Data Services URI. These three method reflect the behavior of the `DataServiceContext` in Astoria. The other three are Azure-specific wrappers for these methods, provided for convenience. `GetEntity` takes three string parameters: `tableName`, `partitionKey` and `rowKey`. These are combined with the base URI for the storage account into an entity URI. `GetTableNames` returns the names of all the tables in the storage account. Finally, `Query< >` is similar to `Execute< >`, except it takes just the table name and the query string as parameters. These are combined with the storage account URI to form a URI which can be passed on to `Execute< >`. The parts of the public interfaces of `AzureTablesContext` and `DataServiceContext` that are relevant to Azure Tables are thus the same. The implementation is very different, however, for two reasons. First, since LINQ-to-Azure is designed specifically for Azure Tables, it is not necessary to be as general as Astoria. Secondly, LINQ-to-Azure is designed to deal with the flexible nature of the Table service. Therefore, instead of the general data objects used in Astoria, LINQ-to-Azure uses objects specifically designed for dealing with this flexibility.

6.1.1 Flexible Properties

As mentioned earlier, entities in Azure Tables and other Cloud databases are flexible with regard to their properties. This flexibility in Cloud databases has an equivalent in programming languages. In most cases, objects are static when it comes to the properties they contain, just like most databases use a fixed schema. This approach has the advantage of being simple and robust. In some cases, however, flexible properties can these out-weigh advantages.

Most languages (exceptions are JavaScript and Lisp) do not natively support dynamic properties, which means the compiler or IDE is often unable to assist in using them. It also means that in order to use them, the functionality will have to be implemented first. There are several patterns which can be used to implement flexible properties.[12] In the most basic implementations, objects use a simple dictionary to maintain a list of named properties. This approach is easy to implement and understand. On the other end of the spectrum, it is possible to build an entire prototyping framework allowing all sorts of flexible modeling.[25]

It is obvious that an application that uses flexible properties in any capacity can benefit from a database that offers a similar construct. And vice versa, an application using a database with a flexible data model would do well if this flexibility can be recreated in the runtime part of the application.

6.1.2 Modeling flexible tables and entities in .NET

An entity in a Cloud database as described in chapter 3 is basically collection of key-value pairs, so the obvious way to model it in .NET is as a Dictionary, which is basically the same thing. The keys are strings and the values can be any object. This reflects the data models used in Cloud databases.

This makes it possible to model basically any entities with flexible properties, however, fixed properties also have a lot of advantages, one of which is that they simpler to use,

particularly when it comes to LINQ queries (see section 6.1.4). Consequently, it would be nice if we could use both flexible and fixed properties in the run-time representation of a Cloud database entity. Such a type of entity already exists in the Python API of Google App Engine, and it is called an Expando.[14] An entity can derive from the `Expando` class, and define some fixed properties specific for the entity type. This creates a hybrid entity type, supporting both fixed and flexible properties in the runtime representation of an entity.

Google App Engine also defines a class for models with polymorphic properties. It is often useful to be able to define data types in a hierarchic fashion, much like in an object oriented system. In such a setup, a query for a certain type will return entities of that type and any of its subtypes. This is especially helpful in data sources where data is not normalized, as in Cloud databases, or for storing inheritance hierarchies. Chapter 7 describes an example of an application illustrating this functionality.

The behavior of Expando objects is defined in the `Expando` class. Unlike in Google App Engine, where the expando and inheritance functionalities described above are mutually exlusive, the LINQ-to-Azure `Expando` also provides the inheritance funtionality. By inheriting from this class, it is possible to create entire hierarchies of related objects, which support both fixed and flexible properties.

When using objects flexible properties, one of the consequences is that it is unknown which properties a specific object might have. In order to ease the discovery of these properties, `Expando` implements the IEnumerable interface, which can be used to list all the properties, fixed and flexible, an entity has.

Mapping inheritance hierarchies

An obvious use for the Expando objects is storing hierarchies of objects, since it has the ability to deal with polymorphic types. Storing these hierarchies of objects in a relational database is problematic, since there exists no concept comparable to inheritance in the relational model. Several approaches exist to store inheritance hierarchies in relation databases, none of which are ideal. They either lead to a lot of tables, a lot of NULLs, or a lot of joins[22, 3]. Using the flexible nature of Cloud databases and the Expando functionality of LINQ-to-Azure, storing inheritance hierarchies becomes a little easier. This does not mean that every application which requires the persistence of inheritance hierarchies should start using a Cloud database and LINQ-to-Azure, however. There are many factors which contribute to the choice of particular method data storage, and the benefit of the Expando functionality is only one of them. But in an application suitable for Cloud databases, `Expando` makes storing inheritance hierarchies relatively easy.

6.1.3 Specializing for Azure Tables

Now that we have defined a general `Expando` type we can specialize it for use with Azure Tables specifically. Since every entity in Azure Tables has three mandatory properties, `PartitionKey`, `RowKey` and `Timestamp`, it makes sense to define an `AzureTablesEntity` type which inherits from `Expando` and defines these properties as fixed properties of the type. `AzureTablesEntity` also overrides the `Equals` and `GetHashCode` methods inherited

from `Object`. These methods are used to determine the identity of objects, and since entities in Azure Tables are identified by their partition key and row key, `Equals` and `GetHashCode` use these properties to reflect this aspect.

There are some restrictions on the attributes of Azure Tables entities: the values can be only of the types described in table 4.1, and not all strings are legal property names. Attributes that are not allowed in Azure tables should not be allowed in the model either, so any attributes assigned to the entity should be validated. The names and types of these properties are validated, hence ensuring they are legal in Azure Tables. This prevents the potentially frustrating debugging scenario where a property is not stored in Azure tables because its name or value is invalid.

Apart from these general aspects, the types used to model entities contain little other logic. Any of the code required to actually store them in or retrieve them from Azure Tables is in other specialized classes. More on that later.

6.1.4 Syntactic Sugar with C# 4.0

C# 4.0 supports dynamic objects in order to provide better integration with dynamic languages like Python and Ruby. The Dynamic Language Runtime, or DLR, defines a high-level helper class, called `DynamicObject`, which can be used to enable dynamic behavior for a class that inherits from it.[21] By implementing an entity as a dynamic object, properties can be assigned and modified by using

```
myEntity.PropertyName
```

instead of

```
myEntity.Properties("PropertyName")
```

or

```
myEntity.GetProperty("PropertyName")
```

Since the handling of these dynamic properties is done by the object itself, it can perform checks to ensure the property and its value are legal for storage in Azure Tables. Using this dynamic behavior requires the object to be typed as dynamic.

Dynamic operation are not allowed in LINQ expression trees, however, so it is not allowed to use dynamic properties in LINQ queries. As a work-around, `AzureTablesEntity` implements an indexer property, which allows dynamic properties to be accessed through the following syntax, much like other collections:

```
myEntity["PropertyName"]
```

The type of the property is unknown at compile time, but it is necessary for the compiler know which operators are defined for it, in order to compile a LINQ query. So, the return type of the indexer should be a type for which all possible combinations of operators and types are defined. The `AzureTablesProperty` class does just that. It defines a total of 72

operators[2] in order to allow every comparison supported by the Table service. The indexer property is also the only place where `AzureTablesProperty` type is used, since it is merely a workaround for this particular issue.

Even though this issue is a limitation came up only during the implementation, it is actually not a disadvantage. By having a different syntax for fixed properties versus dynamic properties, it is immediately clear from looking at a query condition which kind of property is used.

6.1.5 Communication with the Table service

Now that we can represent Azure Tables entities in .NET, we need to be able to get them in and out of the Tables themselves. When an entity is inserted or updated, any public properties of a type in table 4.1 that is part of a class inheriting from `AzureTablesEntity` will be stored in Azure Tables, except when it is marked with a custom attribute, called `ExpandoIgnoreProperty`. In addition, all the flexible properties added to the entity need to be stored. Since `Expando` implements `IEnumerable`, it is easy to loop through all properties in an entity and select all those that are storable in Azure Tables in order to build an Atom representation of the entity, which can be sent to the Table service. The hard part starts when an Expando entity is retrieved from the Table service. It is necessary to determine what the type of the entity was when it was originally stored, since it is necessary to distinguish between the fixed properties of the type and the dynamic properties of the entity itself. An easy way to achieve this is to reserve a property in the entity to explicitly denote its type. This allows the system to quickly determine what the original runtime type of the entity in the table was. If a corresponding type exists in the context of the application, the system will try to deserialize the entity into that type, and cache the matching type for later use (since finding the correct type is an expensive operation, and it is likely that more entities of the same type will be encountered). If no match is found, or if the deserialization fails, the system will default to using the general `AzureTablesEntity` type. This might happen when the data is accessed by multiple applications, where one application may define entity types unknown to another. This way, the data is still accessible, but all but the mandatory properties are regarded as flexible ones.

In order to also be able to distinguish subtypes, the `Type` property should include the entity's entire lineage, up to and including `AzureTablesEntity`. Queries for specific types can then be implemented by doing a prefix match on the `type` property of the entity.[3]

When `SaveChanges` is called, the `AzureTablesContext` iterates through its list of operations on entities which were being tracked. For each of these, the operations needs to be translated into a HTTP request with the correct headers and body, which is sent to the Table service, which will return a HTTP response, which needs to be processed by the system.

[2] These 72 operator break down like this: 2 for Booleans, byte arrays and Guids, 6 for string, int32, int64, double and DateTime. To support the operators both ways, we need two for each of them. (2+2+2+6+6+6+6+6)*2 = 72

[3] Prefix matching string properties in the Table service can be implemented by using the >= and <operators. For example, querying for "string >= 'type' AND string <'typf'" will return all entities whose 'string'-property start with "type".

However, for every kind of operation, the exact details of this chain of events is slightly different. Some message require a body, others do not. Some require an e-tag, others do not. On the other hand, there is also a large amount of overlap in the handling of these operations. As a result, implementing the different kinds of operations is the perfect scenario for using the power of inheritance of objects. Every operation has a corresponding class which includes only the behavior specific to that operation. Any behavior that is shared between two or more classes is defined in a common superclass. This approach leads to less and clearer code. Figure 6.1 illustrates the relationships between the different types of operations and which classes override which methods to suit their particular needs. The abstract `AzureTablesOperation` class is highest in the hierarchy and defines the common methods. Processing any operation involves three basic steps: Generating the HTTP request, sending it out to the service and getting the response, and finally processing this response. These steps are defined, respectively, in the `GenerateRequest`, `Send` and `HandleResponse` methods. The `ProcessOperation` method executes these methods in this order. Because this whole process is time consuming, particularly since it needs to wait for a response from the Table service, operations are processed asynchronously. When the `SaveChanges` method is called, it invokes the ProcessOperation method asynchronously for every operation. The method returns immediately so the rest of the work can continue. The operations are then completed in the background. The `AzureTablesContext` implements the `IDisposable` interface, in order to prevent any updates from being improperly handled at the end of its lifetime. When an `AzureTablesContext` is disposed, it first checks to see if any operations are still being processed, and waits until they are finished. This makes sure that no unexpected behavior can occur when an application is done with the context quickly after a call to `SaveChanges`, which is, after all, not unlikely to be the last method called before it loses the interest of an application.

6.2 Providing LINQ

As illustrated in chapter 5, the main parts of a LINQ provider are the implementations of the `IQueryable` and `IQueryProvider` interfaces. A large part of these implementations are rather general, and not specific to the data source. The most interesting part is the `Execute` method in the `IQueryProvider` implementation. In this method, the query expression tree is analyzed and modified through the use of expression tree visitors. [4] These visitors traverse the expression tree of the LINQ query in much the same way a compiler processes an abstract syntax tree. Each node in the tree is visited and is processed according to its node type. There are a number of different visitors, each with their own function. The `Execute` method uses the following expression tree visitors (presented in the order of usage):

- `InnerMostWhereFinder` extracts the first `where` clause of the query
- `SubTreeEvaluator` and `Nominator` find and replace subtrees that can be evaluated locally, without querying the data source.

[4] The Visitor design pattern is described in [13] and is useful for traversing complicated data structures.

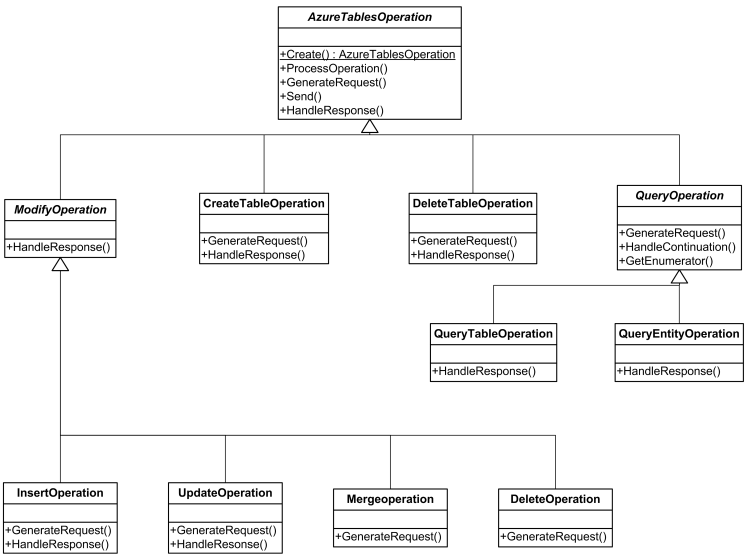

Figure 6.1: The relationships between the different Table service operations

- `QueryTranslator` constructs a Table service query based on the inner-most `where` clause.
- `TakeFinder` handles the `take` and `first` query operators
- `ExpressionTreeModifier` replaces the inner-most query operator with an actual list of entities returned by the query. This allows LINQ-to-Objects to handle query operators not implemented by the Table service.

`InnerMostWhereFinder`, `TakeFinder` and `ExpressionTreeModifier` are each only concerned with finding one specific node to process, and therefore they are rather simple. `SubTreeEvaluator` and `Nominator` work together to find, process and compile parts of the query that require no communication with the Table service. For example, if the `where` clause would include a condition where an attribute is compared to the result of a certain method, These two visitors make sure that this method is executed and replace the method call with its actual result. The `QueryTranslater` is by far the most complicated. It processes the lambda expression for the `where` operator and translates it to a Table service query, like a compiler translating one programming language to another, but simpler.

The Table service only supports a few of the query operators available in LINQ. `Where` can be implemented by the `$filter` option, `take` and `first` can be implemented by `$top`. But this does not mean the LINQ provider for Azure Tables can only support queries using just those operators. Operators like `select`, `orderby` and `groupby` may not be supported by the Table service, but they are frequently useful. Since a LINQ provider can modify the query's expression tree, it is possible to extract the query operators which are supported by the Table service. These are then translated for processing by the Table service, while

the remaining operations are delegated to LINQ-to-objects. For example, a query which includes an `orderby` operator will query the Table service with only the `where` and `take` operators. The results returned by the Table service will be ordered by partition key and then by row key, since this is the specified behavior for the service. The ordering will consequently be handled by LINQ-to-Objects, so that the results returned by the LINQ query are ordered as requested.[5]

6.2.1 Handling tables with different types

So LINQ-to-Azure is only concerned with handling the `where` and `take` query operators, which means it is not the most complicated LINQ provider in the world. Most of the implementation of the `IQueryable` and `IQueryProvider` interfaces are rather general, and the complicated work is processing the expression tree, which is handled by different classes. There are some aspects, though, in which LINQ-to-Azure differs from providers for classic data sources. A data source, in this context, is the part that would implement the `IQueryable` interface and which appears in the `from` part of a query. One of these differences comes from the fact that most data sources, like arrays or SQL tables, are usually typed; they contain only a single data type. When it comes to Azure Tables, the data source would be, obviously, a table. In the simple case, every table in Azure Tables contains the same data type: the `AzureTablesEntity` type defined earlier, with three mandatory properties and a bunch of optional ones. But with the introduction of types (in the form of Expando objects), things become more complicated. Now it is necessary to deal with different types. But Azure Tables can still contain whatever type is put into them. There is no relationship between the name of the table and the type (or types) it contains. So LINQ-to-Azure has a unique problem to solve: dealing with data sources that can contain a multitude of types. Being able to query a table for a specific type is a desirable feature, because in many cases the user will only be interested in a specific type of entity. However, the table containing these entities may also include other types, so specifying a table is not enough.

Recall that the `CreateQuery< >` method in the `AzureTablesContext` class returns an `IQueryable` which can be used in LINQ queries. This is a generic method which takes the table name as a parameter and the type of the results as a type parameter. So far, decoupling the table name and the result type is supported. The `IQueryable` contains an `IQueryProvider` which handles the execution of the query. As such, the `IQueryProvider` implementation also needs to know the generic type of the `IQueryable` in order return only results of this generic type. Consequently, the `IQueryProvider` implementation for LINQ-to-Azure, `AzureTablesQueryProvider`, should also be generic, so it can be instantiated with the same type parameter. If the type specified is the general `AzureTablesEntity` type, things are easy. All entities in Azure Tables are of this type, so no filtering is necessary. If,

[5]There is one caveat to this behavior. Since tables in Azure can be truly massive, operations that are executed immediately (as opposed to deferred), like ordering, can end up retrieving enormous amount of entities, to a point where the results might not fit in the available local memory. This is one of those cases in which using LINQ does not absolve a user of knowing about the characteristics of the data source. In order to use LINQ-to-Azure effectively, a user should know which part of the query is executed in Azure, and which operations are done locally, and what the implications are of this distinction.

however, a subtype of `AzureTablesEntity` is specified, the provider should add a filter option to the Table service query in order to get only the entities of the specified type.

An important thing to keep in mind is that whether or not the query is typed, results returned from the data source will always be analyzed for type information and deserialized into an appropriate CLR type if one is available. The generic type of the query only affects the generic type of the returned `IQueryable`.

6.2.2 Query continuation

As mentioned in section 4.2, a query to the Table Service returns a maximum of 1000 entities at a time. When the number of results exceeds 1000, the HTTP response will include the partition key and row key values of the next entity in the result set as continuation tokens. The remaining results can be retrieved by repeating the query and including these values, indicating that the next result in the set should be the one defined by the continuation tokens.

This is an area in which the deferred execution aspect of the `IEnumerable` interface is helpful. When a query operation is performed, the first 1000 results will be returned by the request. When the results are enumerated, everything proceeds as normal. Only when the 1001st result is requested, the operation object will check to see if more results are available (in other words, did the query response include continuation tokens?) and will send the follow-up query. This is to make sure that follow-up queries are only executed when their result are actually requested, improving performance and limiting data traffic.

Chapter 7

Evaluation

This chapter describes an example application to illustrate the features and functions of LINQ-to-Azure and to evaluate it.

7.1 Example: Guitar store

Consider the product catalogue of a music store. It would contain a number of different kinds of products, like musical instruments, amplifiers, sound processors and speakers. Some properties apply to all of them, for example the price, the brand name and the description. Others properties apply only to certain kinds of products; the number and type of strings (the kind that is found on instruments, not the kind found in programming languages) are only useful in stringed instruments, of course.

7.2 The data model

How can such a product catalogue be stored in Azure Tables? First we need to decide some general issues concerning how it fits into the overall architecture of Azure Tables. This is a process that is not as clearly determined as it would be in the context of a relational database or an object oriented application. There are a lot of things to consider when designing a data model for Azure Tables, which inevitably leads to some compromises. The product catalogue as a whole constitutes a reasonably well defined domain, so it makes sense to put the catalogue in a single table. Note that, although this is easily achieved with LINQ-to-Azure, it would be difficult to achieve it using the Astoria libraries: it would require a single class containing all possible properties for any kind of product. Now that we have decided what the role of a table is, the next step is determining what the partition key and the row key should be. Recall that the combination of these two uniquely determine the entity. There are a couple of candidates for the partition key. One alternative is to partition the catalogue by the product category, as it would most likely be in a paper catalogue. Another one would be a product's manufacturer or the brand name. A product's type number seems a reasonable choice for the row key. To ensure uniqueness for the combination of the partition key and the row key, we will use the brand name as the partition key.

Listing 7.1 shows the code for the general `Product` class, which inherits from `AzureExpando`. The `Brand` and `TypeNumber` properties are wrappers for the partition key and row key, respectively. Although it is rarely an actual number, we need to distinguish this property from the CLR class `Type`, hence the ”number” suffix. They simply return these values when references. As such, these properties need not be stored in Azure Tables, so they are adorned with the `AzureIgnoreProperty` attribute. The `Price` property represents the price in Euro cents, `Descripion` speaks for itself. The `Category` property denoted the product category more accurately and human-friendly than the `ExpandoType` property, and is useful for certain queries, as we will see. Finally, the class provides two constructors: a parameterless one, necessary for deserialization, and one that which allows the key properties to be specified.

Listing 7.1: The Product class for the guitar store example

```
public class Product : AzureTablesEntity
{
    [ExpandoIgnoreProperty]
    public string Brand
    {
        get  { return this.PartitionKey; }
    }
    [ExpandoIgnoreProperty]
    public string TypeNumber
    {
        get { return this.RowKey; }
    }

    public int Price { get; set; } // in cents
    public string Description { get; set; }
    public string Category { get; set; }

    public Product() : base()
    {
    }
    public Product(string brandName, string typeNumber)
        : base(brandName, typeNumber)
    {
    }
}
```

For brevity and simplicity, we will now focus on the guitar department of our music store. There are a number of types of products in this department. Obviously there is the ”guitar” category. Within this category there are different guitar types: bass guitars, and acoustic and electric guitars, for example. Next, there are amplifiers, also of different types: transistor and valve amps, guitar amps and bass amps, combo’s that have integrate speakers, and tops that do not. Still other product types include effects, strings (again, the type found on guitars) and miscellaneous guitar parts.

Listing 7.2 shows the code for the `Guitar` class, which inherits from `Product`. It is a simple class, defining a few properties which are applicable for any guitar.

Listing 7.2: The Guitar class for the guitar store example

```
public class Guitar : Product
{
    public string BodyShape { get; set; }
    public int NoOfStrings { get; set; }
    public int NoOfFrets { get; set; }

    public string Color { get; set; }

    public Guitar() :base()
    {
    }

    public Guitar(string brandName, string typeNumber)
        : base(brandName, typeNumber)
    {
    }
}
```

Whether a certain (sub-)category should be a distinct class is not clearly defined. It depends on the number of common properties, but it is also a matter of preference. Effect pedals have virtually no properties in common, but they clearly conform to the concept of a category. So, should it be a separate class or not? This is up to the designer to decide, there are no hard and fast rules. Since there is no multiple inheritance in .NET, we are limited in when it comes to overlapping categories. So in such cases, it is wise not to commit to any particular subclass, but use flexible properties, even though they may be shared between several types.

Figure 7.1 shows the class diagram of the data model for the guitar store example. The other classes inherit from `Product` and define some properties that all entities of these classes have in common. Individual products can still have a wide range of properties not specified by this data model. In the case of guitars, flexible properties could describe the materials of certain parts, the type of neck joint, information on the hardware, etc.

7.3 The data

Listing 7.3 shows the creation of a small sample of data for the product catalogue. It assumes a `AzureTablesContext` object is already initialized for use with a storage account. First, a table called ”Products” is created. Next, the a number of different entities are defined, which are added to the context using the `AddObject` method. Finally, the entities are submitted for insertion to the Table service by calling `SaveChanges`.

The fixed properties defined in the classes can be used as any strongly typed property. By typing an entity as `dynamic`, any flexible property can also be added as if it where already defined.

Listing 7.3: Inserting data into a table using LINQ-to-Azure

```
context.CreateTable("Products");

Guitar guitar1 = new Guitar("Gibson", "Gothic V");
guitar1.BodyShape = "Flying V";
guitar1.Category = "Guitar, electric";
guitar1.Color = "Black";
guitar1.NoOfFrets = 22;
guitar1.NoOfStrings = 6;
guitar1.Price = 160000; //remember, these are cents

dynamic dyn1 = guitar1;

dyn1.Fretboard = "Ebony";
dyn1.Bridge = "Tune-o-matic";
dyn1.BridgePickup = "Gibson 500T";
dyn1.NeckPickup = "Gibson 496R";
context.AddObject("Products", dyn1);

dynamic bass1 = new Guitar("Fender", "Precision American Standard
    Black");
bass1.BodyShape = "Precision";
bass1.Category = "Bass, electric";
bass1.Color = "Black";
bass1.NoOfFrets = 20;
bass1.NoOfStrings = 4;
bass1.Price = 120000; //remember, these are cents
bass1.BridgePickup = "American standard precision bass alnico 5 split
    single-coil";
bass1.Bridge = "High mass vintage";
context.AddObject("Products", bass1);

dynamic effect1 = new Product("Boss", "MT2");
effect1.Price = 9000;
effect1.Category = "Guitar Effect";
effect1.EffectType = "Distortion";
context.AddObject("Products", effect1);

context.SaveChanges();
```

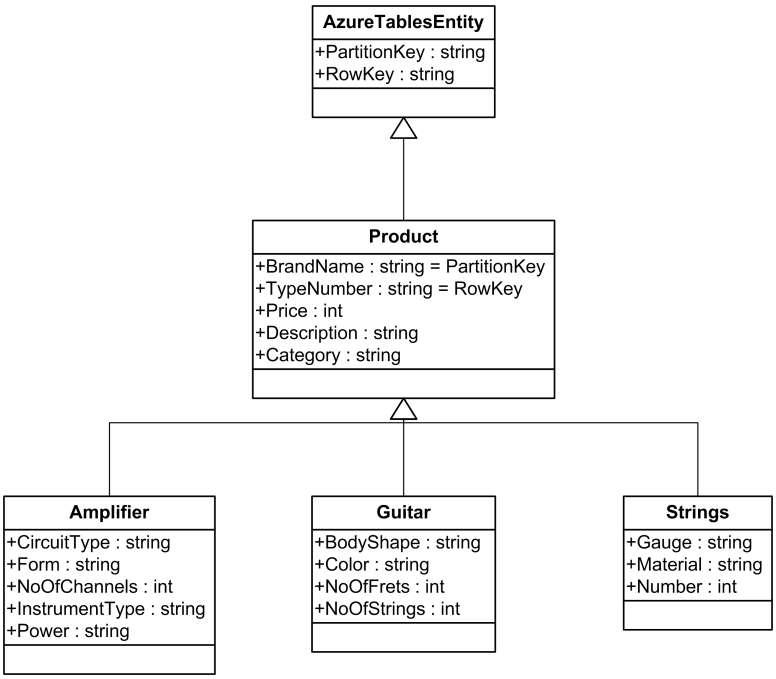

Figure 7.1: The data model for the guitar store example

The entities are identifiable and addressable by the combination of the brand name and the type number, which is quite a sensible way to distinguish the products.

7.4 Simple queries

The most basic query, simply retrieving all products in the table, would look like this:

Listing 7.4: Querying all entities in the table

```
var results = from p in context.CreateQuery("Products")
              select p;
```

Entities of a certain type can be queries using the generic `CreateQuery` method to specify the type:

Listing 7.5: Querying all entities of the Guitar type

```
var results = from g in context.CreateQuery<Guitar>("Products")
              select g;
```

Fixed properties are easy to use in queries, since they are part of the CLR type, they can be used as such in LINQ queries.

Listing 7.6: Querying entities cheaper than 100 euros

```
var results = from p in context.CreateQuery("Products")
              where p.Price < 10000
              select p;
```

Flexible properties require the use of the indexer property and the `AzureTablesProperty` workaround. For example when looking for an amplifier featuring two speakers the following query can be used:

Listing 7.7: Querying entities amplifiers with 2 speakers

```
var results = from a in context.CreateQuery<Amplifier>("Products")
              where a["NoOfSpeakers"] == 2
              select a;
```

Note that due to the nature of `AzureTablesProperty`, the `NoOfSpeakers` property could also be compared to non-integer values, like 2.0 (a floating point value) or ”2” (a string value). These will not provide any results, though, so care has to be taken when querying flexible properties.

7.5 More advanced queries

By always specifying the instrument as the first part of the `Category` property, it becomes easy to query all the items regarding to a certain instrument, by doing a prefix match on this property. For example:

Listing 7.8: Querying all entities regarding basses

```
var results = from p in context.CreateQuery<Product>("Products")
              where p.Category >= "Bass"
              && p.Category < "Bast"
              select p;
```

These sort of tricks are very useful in dealing with denormalized data. Lacking the querying powers of a relational system, sometimes some creativity is necessary to achieve the desired result. Using redundant data in ways like this is a possible trick, however it should be remembered that the user is responsible for maintaining consistency is such scenarios.

Recall that the Table service itself only supports the `where` and `take` query operators. Other query operators can be used, but they are handled by LINQ-to-Objects. For example, the following query will result in a list of the brands in the catalogue:

Listing 7.9: Retrieving the list of brand names

```
var results = context.CreateQuery<Product>("Products").Select(p => p.
    PartitionKey).Distinct();
```

The `Distinct` query operator is not part of the query expression defined for C#, so the above query is specified using the method call syntax.

The following query result is a list of all products ordered by price:

Listing 7.10: Ordering the products by price

```
var results = from p in context.CreateQuery("Products")
              select p
              orderby p.Price;
```

Keep in mind that although these queries return the expected results, the amount of data returned by the Table service can greatly exceed the data actually needed. When dealing with massive amounts of data (which is, after all, what the Table service was designed to do) the amount of data returned could cripple the performance of the application, or even exceed the available memory. Consequently, it is important to be careful in these scenarios.

7.6 Reflection

As described in this chapter, accessing and querying Windows Azure Tables using LINQ-to-Azure is made easy by leveraging the power of LINQ and the dynamic features of C# 4.0. It allows users to work with the REST interface without ever having to wonder what a HTTP request looks like. LINQ-to-Azure supports all the power and flexibility of the Table service with a minimum of overhead for the user and without the limitations of the Astoria libraries. Users can query entities in Azure Tables using LINQ and manipulate them through the use of CLR objects, and LINQ-to-Azure takes care of the whole HTTP communication, including paging and authentication. This allows them to make optimal use of Azure Tables, including the flexibility od the data model.

Chapter 8

Related Work

The amount of scientific literature on Cloud computing is rather limited. Cloud computing is a relatively young technology, developed mostly in corporate environments, not in academia. Consequently, most of the information on the topic, both on its theory and practice, is provided by the companies developing Cloud platforms, and blogging programmers (or programming bloggers) working with these platforms. Neither of these generally publish their material in peer-reviewed scientific journals. One of the few things to come out of academia is a paper written by a number of Berkeley researchers,[4] which gives a high level overview of Cloud technology, but offers little original material; it's mostly a summary of publicly available material.

In relation to Cloud databases, the story is not much different. Although the question how to deal with data in a flexible but non-relational data model is an interesting problem worth investigating, virtually all of the material on it is provided by the vendors of these systems. A possible reason for this is that there is no clearly defined Cloud database or data model; each vendor uses its own system. These systems may be similar on the surface, but there are subtle differences between them that make generalizations rather tricky.

8.1 ADO.NET Data Services

Since Azure Tables implements ADO.NET Data Services, the obvious method of communicating with this service, at least from a .NET environment, is using the client libraries supplied with the .NET Framework. The use of these libraries relieves the user from having to delve to much into the particulars of either the data source, or the Data Services architecture itself. However, as was illustrated in section 4.2, there exists a mismatch between the ADO.NET client libraries and the architecture of Azure Tables.

The main goal for ADO.NET Data Services is to provide a uniform approach for communicating with data sources over the Internet. So when a communication protocol was necessary for accessing Azure Tables, using ADO.NET Data Services made sense. Although certain operations defined in ADO.NET Data Services are not applicable to Azure Tables, it would be foolish to use a different mechanism for the operations that are applicable.

The mismatch is only introduced when the data service is accessed through the client libraries. For the client libraries to deal with many different data sources, some generalizations are necessary. One of the generalizations made in the ADO.NET Data Services client libraries is the use of a fixed schema. While this is certainly true for the vast majority of data sources, it is not for Azure Tables. The use of the client libraries for accessing Azure Tables is therefore a compromise between simplicity and generality on one side, and flexibility and control on the other.

The advantage that LINQ-to-Azure has over the client libraries is that LINQ-to-Azure is specific to Azure Tables, and therefore is not concerned with providing a uniform experience for any other data sources. Because of this singular focus, LINQ-to-Azure does not have to make any compromises, which are necessary for the ADO.NET client libraries in order to provide a general solution.

8.2 LINQ to SimpleDB

As mentioned in chapter 4, the data model used in Azure Tables resembles that of Amazon SimpleDB. The main difference between the two is that properties in SimpleDB are always string values, whereas Azure Tables supports several other data types. A common aspect is the concept of flexible entities with dynamic properties. A LINQ provider for SimpleDB would therefore share a significant part with one designed for Azure Tables.

Justin Etheredge developed such a LINQ provider.[8, 9] Although the local entity management part shows some difference with LINQ-to-Azure (largely due to the difference between Azure entities and SimpleDB entities), the LINQ part is quite similar. Of course, there are some aspects of SimpleDB that have an impact on LINQ-to-SimpleDB. Entities in SimpleDB can have multiple values, so LINQ-to-SimpleDB needs support for that. Another useful feature helps users deal with the fact that all attributes in SimpleDB are strings. LINQ-to-SimpleDB automatically converts other useful data types to a suitable string representation.

8.3 Google App Engine

For their App Engine, Google also uses a data storage facility that has a flexible data model. The App Enginge data store is built on BigTable, which is itself similar to SimpleDB.[5] Applications in AppEngine can be built either using Python or Java. When using the Python API, entities can be stored in the data store by defining special Python classes called models. There are three types of models in the Python API: regular `Model`s, which simply define a number of fixed properties all `Model`s have, `Expando` models, which can also have flexible properties, and `PolyModel`s, with which inheritance hierarchies with polymorphic data can be modeled. The `Expando` class described in chapter 6 is basically a combination of App Engine's `Expando` and `PolyModel`, supporting both flexible properties and inheritance hierarchies. In the App Engine, however, `Expando` and `PolyModel` are mutually exclusive: `Expando` has no polymorphic features, and `PolyModel` cannot have flexible entities. Although the App Engine data store shows a lot of similarities with Azure Tables, there is one

important difference: the App Engine data store can only be used by applications running on the App Engine, whereas Azure Tables can be used by any application, in the Cloud or local.

Chapter 9

Conclusions and Future Work

Cloud databases generally use a flexible data model, in contrast to relational databases or object models. Many programming languages are statically typed and do not support classes with flexible properties natively. This represents a mismatch between Cloud databases and the application that use them.

For example, at present, anyone who would like to use Windows Azure Tables in any .NET application has two options: either they use it via the Astoria .NET libraries and the storage client, or they work with the REST interface instead. The first choice gives you convenience, but limits what you can do, since Astoria needs a fixed EDM schema to work with. This prevents the user from taking advantage of the flexibility required to make optimal use of the data model. The second choice enables flexibility, but requires the user to do all the work Astoria would otherwise do for him, which is quite a lot.

9.1 Contributions

This document describes the design and implementation of LINQ-to-Azure, a data access layer for Azure Tables, as an alternative to the ADO.NET Data Services libraries in .NET. LINQ-to-Azure provides an abstraction layer to Azure Tables, allowing entities in the tables to be queried using LINQ and manipulated as CLR objects, just like when using ADO.NET Data Services client libraries. Unlike these client libraries, however, LINQ-to-Azure does not require the use of a fixed schema. The Table service itself is a schemaless database, consisting of tables containing key/typed-value pairs. The database or table does not define which properties an entity should have. Entities are flexible and entities with different properties can exist in the same table. LINQ-to-Azure supports this feature of Azure Tables, allowing a user to leverage the flexible nature of the entities to optimally use the storage. It achieves this by plugging into the REST interface available for the Table service. At the same time, the service can be queried using LINQ, and entities will be returned as typed CLR objects which can have both fixed (pre-defined) and flexible properties. These objects can be manipulated and changes can be saved back to the Table service in way similar to the functionality of the ADO.NET Data Services client. This gives user all the benefits the REST interface, without having to design, build an maintain any of the communication

code. This can dramatically simplify applications which require the use of the flexibility provided by Azure Tables by delegating all the repetitive work to LINQ-to-Azure.

The CLR representation of Azure Tables entities can also be organized in an inheritance hierarchy, a feature which is fully supported by the LINQ provider. Even when an entity has an unknown type (possibly introduced by a LINQ-to-Azure application with a different data model) these can still be manipulated as a general entity, represented by an object with flexible properties.

9.2 Conclusions

LINQ-to-Azure illustrates how a database with a flexible data model can be manipulated through a programming language with a fixed data model. While LINQ-to-Azure, as described in this document, is specific to .NET and Windows Azure Tables, a similar approach can be taken to intergrate other Cloud databases or other languages. On one hand, LINQ-to-Azure could be modified to target Amazon SimpleDB, for example, by changing the query translator and the HTTP message handlers. On the other hand, the principles illustrated by the communication portion of LINQ-to-Azure could be used as a basis for implementing a similar data access layer to Azure Tables in another language. Of course, non-.NET languages will not get the benefits of using LINQ for querying the Table service, and the dynamic handling of flexible properties is not supported in every language. Nevertheless, the concepts outlined by LINQ-to-Azure, including Expando objects and using lists to store properties, are applicable to many other environments.

9.3 Reflection

Building a LINQ provider is not a easy thing, even a relatively simple one like LINQ-to-Azure. A LINQ provider is, in effect, a miniature compiler translating one query language to another, which is never a trivial problem. It took quite some time to get familiar with all the ins and outs of the different parts that make up a LINQ provider, making for a slow start of the implementation.

An interesting part of the project for me was working with C# 4.0. In this project, the only new feature I used was the dynamic language capabilities, but experiment with other features of the new version was certainly instructive. On the other hand, using C# 4.0 implies using Visual Studio 2010, which was, like C# 4.0 itself, still in beta during the implementation phase of this project. This meant my IDE was often slow and prone to crash, especially during debugging. The most interesting part of working with a beta language and IDE came when I ran into a problem concerning calling constructors in inner classes with arguments that are typed `dynamic`, resulting in a runtime exception. After spending a lot of time testing, debugging and trying to track down the problem (with the assistance of the people at StackOverflow[23]), it appeared that calling an internal constructor with an argument that is typed `dynamic` was the source of the problem. It seemed the runtime was not considering internal constructors when it was trying to resolve the dynamic call. Conclusion: this is a bug in the runtime, and I spent a day finding a bug that in C# 4.0

instead of my own code. This bug was consequently reported on Connect (Microsoft's bug report system), and will be fixed in the next release.[24]

9.4 Future work

9.4.1 Complex objects

In LINQ-to-Azure, entities can only have properties of primitive types, specifically types that are (similar to those) supported by the Table Service. A useful addition to LINQ-to-Azure could be the option to allow entities with properties that are of the `AzureTablesEntity` type. This would make it easier to work with more complex objects. These could be implemented in Azure Tables either by storing the properties of the nested `AzureTablesEntity` in-line, as if they were directly defined in the containing entity, or by including a URI to the nested `AzureTablesEntity` as a string property. This is not a trivial feature, however. It introduces some complicated issues, such as dealing with very deep or circular references and how and when nested entities should be retrieved. Another possible addition would be to allow properties to be collections. These collections could contain primitive types, or possibly, `AzureTablesEntity`s. Both these features would increase the expressiveness of LINQ-to-Azure, but also its complexity. Whether they are worth the effort would have to be carefully evaluated.

9.4.2 Batch operations

In May 2009, an update to the CTP of Windows Azure added some features to the Table service. On of those features was the ability to perform batch operations. Operations on entities within the same table and having the same partition key could now be combined into a single batch request. A batch request can contain up to 100 insert, update and delete operations, but entities can only occur once within a single batch. Batches are atomic, meaning that the operations in the batch will either all fail, or all succeed. If any operation in the batch fails, all previous operation within that batch are rolled back. This enables transactions to be performed by using batch requests.

The main advantages of batch transactions are their atomicity and the reduction in the amount of HTTP traffic. Another use for batch transaction in to enable entities to be retrieved whose URI path is longer than the maximum 260 characters allowed by HTTP 1.1. To circumvent this limitation, a batch request can contain a single query retrieving a single entity.

LINQ-to-Azure currently does not support batches. Future versions could support this feature, allowing LINQ-to-Azure to benefit from its atomic transactions and reduction in traffic.

9.4.3 Other target systems

It is possible to use LINQ-to-Azure as a basis for implementing a similar data access layer targeting other, similar systems, either at the front end (other programming environments)

or the back end (other Cloud databases). The development of different back ends would be particularly interesting, since it would provide a uniform way of targeting different Cloud databases from within .NET. Programmers could use the same interface regardless of whether they are talking to Azure Tables or SimpleDB, for example.

Bibliography

[1] Amazon.com. Amazon simpledb developer guide. http://docs.amazonwebservices.com/AmazonSimpleDB/2007-11-07/DeveloperGuide/, 2007.

[2] Amazon.com. Amazon simpledb home page. http://aws.amazon.com/simpledb, 2008.

[3] Scott W. Ambler. Mapping objects to relational databases: O-r mapping in detail. http://www.agiledata.org/essays/mappingObjects.html, 2006.

[4] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical report, UC Berkeley, 2009.

[5] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):1–26, 2006.

[6] David Chappelle. Windows azure and isvs: A guide for decision makers. http://www.davidchappell.com/blog/2009/07/windows-azure-and-isvs-guide-for.html, 2009.

[7] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[8] Justin Etheredge. Linq to simpledb. http://www.codeplex.com/LinqToSimpleDB, 2008.

[9] Justin Etheredge. Linqtosimpledb preview. http://www.codethinked.com/post/2008/01/LinqToSimpleDB-Preview.aspx, 2008.

[10] Mike Flasko. Overview: Ado.net data services. http://msdn.microsoft.com/en-us/library/cc956153.aspx, 2009.

[11] Stephen Forte. Database design patterns. http://www.iasahome.org/pdf/S.Forte.pdf, 2008.

[12] Martin Fowler. Dealing with properties. http://martinfowler.com/apsupp/properties.pdf, 1997.

[13] Erich Gamme, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns, elements of reusable object-oriented software*. Addison Wesley, 1995.

[14] Google. Google app engine: Python service apis. http://code.google.com/intl/nl-NL/appengine/docs/python/apis.html, 2009.

[15] Jai Haridas, Niranjan Nilakantan, and Brad Calder. Windows azure table programming table storage. http://go.microsoft.com/fwlink/?LinkId=153401, 2009.

[16] Fabrice Marguerie, Steve Eichert, and Jim Wooley. *Linq in action*. Manning Publications Co., Greenwich, CT, USA, 2008.

[17] Microsoft. Authentication schemes. http://msdn.microsoft.com/en-us/library/dd179428.aspx, 2009.

[18] Microsoft. Differences between development storage and windows azure storage services. http://msdn.microsoft.com/en-us/library/dd320275.aspx, 2009.

[19] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[20] Jon Skeet. *C# in Depth: What you need to master C# 2 and 3*. Manning Publications Co., Greenwich, CT, USA, 2008.

[21] Alex Turner and Bill Chiles. Getting started with the dlr as a library author. http://dlr.codeplex.com/Project/Download/FileDownload.aspx?DownloadId=68818, 2009.

[22] Rik van der Sanden. Objects, relations and clouds: Investigations into interoperability. Master's thesis, Delft University of Technology, 2009.

[23] Rik van der Sanden and Jon Skeet. Passing a dynamic attribute to an internal constructor. http://stackoverflow.com/questions/1087389/passing-a-dynamic-attribute-to-an-internal-constructor, 2009.

[24] Rik van der Sanden and Alex Turner. internal constructor not found when it's called with dynamic arguments. https://connect.microsoft.com/VisualStudio/feedback/ViewFeedback.aspx?FeedbackID=472924, 2009.

[25] Steve Yegge. The universal design pattern. http://steve-yegge.blogspot.com/2008/10/universal-design-pattern.html, 2008.