Windows Azure Table

Jai Haridas, Niranjan Nilakantan, Brad Calder

Май 2009

Содержание

[1 Введение 2](#_Toc238286281)

[2 Модель данных таблицы 3](#_Toc238286282)

[3 Секционирование таблиц 5](#_Toc238286283)

[3.1 Влияние секционирования 6](#_Toc238286284)

[3.1.1 Масштабируемость таблицы 6](#_Toc238286285)

[3.1.2 Транзакции над группами сущностей 7](#_Toc238286286)

[3.1.3 Расположение сущностей 7](#_Toc238286287)

[3.2 Выбор ключа секции 7](#_Toc238286288)

[3.2.1 Транзакции над группами сущностей 8](#_Toc238286289)

[3.2.2 Эффективные запросы 8](#_Toc238286290)

[3.2.3 Масштабируемость 8](#_Toc238286291)

[3.2.4 Гибкое секционирование 9](#_Toc238286292)

[4 Программирование таблиц 9](#_Toc238286293)

[4.1 Контроль версий 11](#_Toc238286294)

[4.2 Пример 12](#_Toc238286295)

[4.3 Описание класса сущности для таблицы 12](#_Toc238286296)

[4.4 Создание таблицы 13](#_Toc238286297)

[4.5 Вставка Blog 14](#_Toc238286298)

[4.6 Запрос Blogs 14](#_Toc238286299)

[4.7 Обновление Blog 15](#_Toc238286300)

[4.8 Удаление Blog 15](#_Toc238286301)

[4.9 Транзакции над группами сущностей 16](#_Toc238286302)

[4.9.1 Обработка ответа 18](#_Toc238286303)

[4.9.2 Ошибки 18](#_Toc238286304)

[4.10 Рекомендации по работе с DataServiceContext 20](#_Toc238286305)

[4.11 Использование API REST 22](#_Toc238286306)

[5 Параллельные обновления 22](#_Toc238286307)

[5.1 Безусловные обновления 24](#_Toc238286308)

[6 Разбиение на страницы результатов запроса 25](#_Toc238286309)

[6.1 Получение первых N сущностей 25](#_Toc238286310)

[6.2 Маркеры продолжения 25](#_Toc238286311)

[7 Модель согласованности 26](#_Toc238286312)

[7.1 Согласованность в рамках одной таблицы 26](#_Toc238286313)

[7.2 Согласованность таблиц 27](#_Toc238286314)

[8 Советы и рекомендации 28](#_Toc238286315)

[8.1 Извлечение элементов, добавленных последними (моделирование расположения элементов в порядке по убыванию) 28](#_Toc238286316)

[8.2 Извлечение с использованием префикса 29](#_Toc238286317)

[8.3 Пример секционирования данных 30](#_Toc238286318)

[8.3.1 Микроблоггинг 30](#_Toc238286319)

[8.3.2 Динамическая настройка PartitionKey 33](#_Toc238286320)

[8.3.3 Разные виды сущностей в одной таблице 34](#_Toc238286321)

[8.4 Обновление и управление версиями 39](#_Toc238286322)

[8.4.1 Добавление нового свойства 40](#_Toc238286323)

[8.4.2 Удаление типа свойства 41](#_Toc238286324)

[8.4.3 Изменение типа свойства 42](#_Toc238286325)

[9 Лучшие практики Windows Azure Table 42](#_Toc238286326)

[9.1 Создание таблицы 42](#_Toc238286327)

[9.2 Асинхронная версия ADO.NET Data Services API 42](#_Toc238286328)

[9.3 Настройки DataServiceContext 42](#_Toc238286329)

[9.4 Схема секционирования 43](#_Toc238286330)

[9.5 Безусловные обновления и удаления 43](#_Toc238286331)

[9.6 Обработка ошибок 43](#_Toc238286332)

[9.6.1 Успешное завершение операций на стороне сервера, несмотря на ошибки сети и истечение времени ожидания сети 43](#_Toc238286333)

[9.6.2 Повторные запросы и ошибки «Connection closed by Host» 44](#_Toc238286334)

[9.6.3 Конфликты при обновлениях 44](#_Toc238286335)

[9.6.4 Настройка приложения для обработки повторяющихся ошибок превышения времени ожидания 44](#_Toc238286336)

[9.6.5 Обработка ошибок и составление отчетов 44](#_Toc238286337)

[9.7 Настройка производительности .NET и ADO.NET 45](#_Toc238286338)

[9.7.1 Улучшение производительности десериализации при использовании сервисов данных ADO.NET (Data Service Deserialization). 45](#_Toc238286339)

[9.7.2 Изменения значения по умолчанию для HTTP-соединений .NET 45](#_Toc238286340)

[9.7.3 Отключение «100-continue» 46](#_Toc238286341)

[9.7.4 Отключение Nagle может ускорить операции вставки/обновления 47](#_Toc238286342)

[9.8 Повторное использование имени удаленной таблицы 47](#_Toc238286343)

[10 Заключение 47](#_Toc238286344)

# Введение

Windows Azure является основой платформы Cloud Platform Microsoft. Это «операционная система для облака», которая обеспечивает необходимые стандартные блоки для создания масштабируемых сервисов высокой надежности. Windows Azure предоставляет:

* Виртуализированные вычисления
* Масштабируемое хранилище
* Автоматизированное управление
* Насыщенный SDK

Хранилище Windows Azure Storage обеспечивает разработчикам возможность хранения данных в облаке. Приложение может выполнять доступ к своим данным в любой момент времени из любой точки планеты, хранить любой объем данных и как угодно долго. При этом данные гарантированно не будут повреждены и утеряны. Windows Azure Storage предлагает богатый набор абстракций данных:

* Windows Azure Blob – обеспечивает хранилище больших элементов данных.
* Windows Azure Table – обеспечивает структурированное хранилище.
* Windows Azure Queue – обеспечивает диспетчеризацию асинхронных заданий для реализации обмена данными между сервисами.

Этот документ посвящен Windows Azure Table - структурированному хранилищу. Оно поддерживает высокомасштабируемые таблицы в облаке, которые могут содержать миллиарды сущностей и терабайты данных. По мере увеличения трафика, система будет эффективно масштабироваться, автоматически подключая тысячи серверов.

Структурированное хранилище реализовано в виде таблиц (Tables), в которых располагаются сущности (Entities), содержащие ряд именованных свойств (Properties). Вот некоторые из основных характеристик Windows Azure Table:

* Поддержка LINQ, ADO .NET Data Services и REST.
* Контроль типов во время компиляции при использовании клиентской библиотеки ADO .NET Data Services.
* Богатый набор типов данных для значений свойств.
* Поддержка неограниченного количества таблиц и сущностей без ограничения размеров таблиц.
* Поддержка целостности для каждой сущности.
* Нежесткая блокировка при обновлениях и удалениях.
* Для запросов, возвращающих большое количество результатов, или запросов, прерванных по завершению времени ожидания, возвращаются частичные результаты и маркер продолжения, позволяющий возобновить запрос с момента, на котором он был прерван.

# Модель данных таблицы

Далее представлен обзор модели данных таблицы Windows Azure Table:

* **Учетная запись хранилища (Storage Account)** – для доступа к Windows Azure Storage приложение должно использовать действительную учетную запись. Новую учетную запись можно создать через веб-интерфейс портала Windows Azure. Как только учетная запись создана, пользователь получает 256-разрядный секретный ключ, который впоследствии используется для аутентификации запросов этого пользователя к системе хранения. В частности, с помощью этого секретного ключа создается подпись HMAC SHA256 для запроса. Эта подпись передается с каждым запросом данного пользователя для обеспечения аутентификации.  
  Имя учетной записи входит в состав имени хоста в URL. Для доступа к таблицам используется следующий формат имени хоста: <имяУчетнойЗаписи>.table.core.windows.net.
* **Таблица (Table)** – содержит набор сущностей. Область действия имен таблиц ограничена учетной записью. Приложение может создавать множество таблиц в рамках учетной записи хранилища.
* **Сущность (строка) (Entity (Row))** – Сущности (сущность является аналогом «строки») – это основные элементы данных, хранящиеся в таблице. Сущность включает набор свойств. В каждой таблице имеется два свойства, «PartitionKey» и «RowKey», которые образуют уникальный ключ для сущности.
* **Свойство (столбец) (Property (Column))** – Представляет отдельное значение сущности. Имена свойств чувствительны к регистру. Для значений свойств поддерживается богатый набор типов.
* **Ключ секции (PartitionKey)** – Первое свойство ключа каждой таблицы. Эта система использует данный ключ для автоматического распределения сущностей таблицы по множеству узлов хранения.
* **Ключ строки (RowKey)** – Второе свойство ключа таблицы. Это уникальный ID сущности в рамках секции. PartitionKey в сочетании с RowKey уникально идентифицирует сущность в таблице.
* **Временная метка (Timestamp)** – Каждая сущность имеет версию, сохраняемую системой.
* **Секция (Partition)** – Набор сущностей в таблице с одинаковым значением ключа секции.
* **Порядок сортировки (Sort Order)** – Для CTP-версии предоставляется всего один индекс, в котором все сущности сортированы по PartitionKey и затем по RowKey. Это означает, что запросы с указанием этих ключей будут более эффективными, и все возвращаемые результаты будут сортированы по PartitionKey и затем по RowKey.

Таблица имеет гибкую схему. Windows Azure Table отслеживает имя и типизированное значение каждого свойства каждой сущности. Приложение может моделировать фиксированную схему на стороне клиента, обеспечивая одинаковый набор свойств для всех создаваемых сущностей.

Рассмотрим некоторые дополнительные сведения о таблицах, сущностях и свойствах:

* Таблица
  + Имена таблиц могут включать только буквенно-цифровые символы.
  + Имя таблицы не может начинаться с цифрового символа.
  + Имена таблиц чувствительны к регистру.
  + Имена таблиц должны включать от 3 до 63 символов.
* Имя свойства
  + Допускаются только буквенно-цифровые символы и «\_».
* Сущность может иметь до 255 свойств, включая обязательные системные свойства: PartitionKey, RowKey и Timestamp. Имена всех остальных свойств сущностей определяются приложением.
* Свойства PartitionKey и RowKey строкового типа размером не более 1 кБ.
* Свойство Timestamp является доступным только для чтения обслуживаемым системой свойством, которое должно рассматриваться как непрозрачное свойство.
* Отсутствие фиксированной схемы – Windows Azure Table не сохраняет никакой схемы, поэтому все свойства хранятся как пары <имя, типизированное значение>. Это означает, что свойства сущностей одной таблицы могут сильно отличаться. В таблице даже может быть две сущности, свойства которых имеют одинаковые имена, но разные типы значений. Однако имена свойств в рамках одной сущности должны быть уникальными.
* Суммарный объем всех данных сущности не может превышать 1 МБ. Сюда входит размер имен свойств, а также размер значений свойств или их типов, включая и два обязательных свойства ключей (PartitionKey и RowKey).
* Поддерживаются типы Binary, Bool, DateTime, Double, GUID, Int, Int64, String. Ограничения представлены в таблице ниже.

Мы используем стандартные ограничения для Http.sys, согласно которым размер сегмента URI не может превышать 260 символов. Это приводит к ограничению размера секции и ключа строки, поскольку GetRow, Delete, Update, Merge требуют включения секции и ключа строки как части одного сегмента URI. Например, следующий URI определяет одну сущность с PartitionKey «pk» и RowKey «rk»:

http://myaccount.windows.core.net/Customers(PartitionKey="pk",RowKey="rk")

Из-за налагаемых Http.sys ограничений выделенная часть не может включать более 260 символов. Для решения этой проблемы операции, имеющие такое ограничение, могут выполняться с помощью Entity Group Transactions (Транзакции групп сущностей), поскольку в Entity Group Transactions URI, определяющий ресурс, является частью тела запроса (подробнее смотрите в разделе 4.9).

| **Тип свойства** | **Описание** |
| --- | --- |
| Binary | Массив байтов размером до 64 КБ. |
| Bool | Булево значение. |
| DateTime | 64-разрядное значение, представляющее время в формате UTC. Поддерживаемый диапазон значений: от 1/1/1601 до 12/31/9999. |
| Double | 64-разрядное значение с плавающей точкой. |
| GUID | 128-разрядный глобально уникальный идентификатор. |
| Int | 32-разрядное целое значение. |
| Int64 | 64-разрядное целое значение. |
| String | 16-разрядное UTF-кодированное значение. Размер строковых значений может быть до 64 КБ. |

# Секционирование таблиц

Windows Azure Table обеспечивает возможность масштабирования таблиц до тысяч узлов хранения через распределение сущностей в таблице. При распределении сущностей желательно обеспечить, чтобы сущности, входящие в одно множество, располагались в одном узле хранения. Приложение формирует эти множества соответственно значениям свойства PartitionKey сущностей.

Приложениям должна быть известна рабочая нагрузка каждой отдельно взятой секции. Для обеспечения желаемых результатов тестирование должно моделировать максимальную рабочую нагрузку.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Ключ секции | Ключ строки | Свойство 3 | … | Свойство N |  |
| Имя документа | Версия | Время изменения |  | Описание |  |
| Examples Doc | V1.0 | 8/2/2007 | … | Завершенная версия | Секция 1 |
| Examples Doc | V2.0.1 | 9/28/2007 |  | Рабочая версия Алисы |
| FAQ Doc | V1.0 | 5/2/2007 |  | Завершенная версия | Секция 2 |
| FAQ Doc | V1.0.1 | 7/6/2007 |  | Рабочая версия Алисы |
| FAQ Doc | V1.0.2 | 8/1/2007 |  | Рабочая версия Салли |

Рис. 1 Примеры секций

На рисунке выше представлена таблица, содержащая множество версий документов. Каждая сущность данной таблицы соответствует определенной версии определенного документа. В этом примере ключом секции таблицы является имя документа, и ключом строки – номер версии. Имя документа и версия уникально идентифицируют каждую сущность таблицы. В данном примере секцию образуют все версии одного документа.

## Влияние секционирования

Теперь поговорим о назначении секций таблицы и принципах выбора ключа секции.

### Масштабируемость таблицы

Хорошая масштабируемость системы хранения достигается за счет распределения секций по множеству узлов хранения.

Система отслеживает характер использования секций и автоматически равномерно распределяет эти секции по всем узлам хранения. Это позволяет системе и приложению масштабироваться соответственно количеству запросов к таблице. То есть если некоторые секции запрашиваются больше других, система автоматически разнесет их на несколько узлов хранения, таким образом, распределяя трафик между множеством серверов. Однако секция, т.е. все сущности, имеющие одинаковый ключ секции, будут обслуживаться как один узел. Но даже несмотря на это объем данных в рамках секции не ограничен емкостью хранилища одного узла хранения.

### Транзакции над группами сущностей

Приложение может автоматически выполнять транзакцию над сущностями, хранящимися в одной таблице и одной секции (т.е. имеющих одинаковое значение ключа секции). Таким образом, приложение может автоматически выполнять множество операций Create/Update/Delete для множества сущностей за один пакетный запрос к системе хранения, поскольку все сущности располагаются в одной таблице и имеют одинаковое значение ключа секции. При выполнении транзакции обеспечивается изоляция моментального снимка независимо от того, были операции над сущностями выполнены успешно либо все они дали сбой. Все остальные запросы, выполняющиеся параллельно в это же время, не будут видеть результат транзакции, поскольку работают с моментальным снимком, сделанным до ее завершения. Результат транзакции становится видимым запросам только после полного успешного ее завершения.

Для Entity Group Transaction указание заголовка версии является обязательным, должна быть указана версия "2009-04-14" или более поздняя. Подробнее об этом рассказывается в разделе 4.1.

### Расположение сущностей

Сущности одной секции хранятся вместе. Это обеспечивает наиболее эффективную обработку запросов к секции. Более того, в этом случае приложение может использовать все преимущества эффективного кэширования и других оптимизаций производительности, обеспечиваемых расположением данных в секции.

В примере выше секцию образуют все версии одного документа. Таким образом, для извлечения «всех версий данного документа» необходимо выполнить доступ всего к одной секции. С другой стороны, чтобы получить все версии документов, измененные до 5/30/2007, придется запрашивать несколько секций. Поскольку запросу придется проверять все секции, которые к томе же могут располагаться на разных узлах хранения, такой запрос будет менее эффективным и более ресурсоемким.

## Выбор ключа секции

Выбор ключа секции важен с точки зрения обеспечения эффективного масштабирования приложения. При этом необходимо найти компромисс между размещением сущностей в одной секции, что обеспечивает большую эффективность запросов, и масштабируемостью таблицы, поскольку чем больше секций в таблице, тем проще для Windows Azure Table распределить нагрузку между множеством серверов.

### Транзакции над группами сущностей

Если в приложении предполагается использовать транзакции над группами сущностей, PartitionKey необходимо выбирать так, чтобы он обеспечивал выбор всех сущностей, участвующих в элементарной транзакции. Основное правило –ParitionKey должен обеспечивать группировку сущностей для выполнения транзакций над группами сущностей. Таким образом, группируются сущности, которые должны обрабатываться совместно, и при этом создается множество секций, что потенциально позволяет Windows Azure Table распределять нагрузку между несколькими серверами.

### Эффективные запросы

Для наиболее частых и критичных по времени ожидания запросов PartitionKey должен быть включен как фильтр запроса. Запрос, в котором указан PartitionKey, ограничен лишь одной или несколькими секциями (в зависимости от используемого условия) и, таким образом, будет намного более эффективным. Если при выполнении запроса PartitionKey не указан, в поисках необходимых сущностей просматриваются все секции таблицы, что значительно снижает эффективность.

Далее представлены некоторые советы и рекомендации по выбору PartitionKey для таблицы:

1. Прежде всего, выявите важные свойства таблицы. Это свойства, часто используемые как фильтры запросов.
2. Из этих важных свойств выберите потенциальные ключи.
   1. Важно понять, какой запрос будет преобладающим для приложения. Затем из этого преобладающего запроса необходимо выбрать свойства, используемые в фильтрах запроса.
   2. Это будет исходный набор свойств ключей.
   3. Расставьте свойства ключей в порядке их значимости в запросе.
3. Проверьте, обеспечивают ли свойства ключей уникальную идентификацию сущности?  Если нет, включите в набор ключей уникальный идентификатор.
4. Если имеется только одно свойство ключа, используйте его в качестве PartitionKey.
5. Если имеется только два свойства ключей, первое используйте как ParitionKey и второе – как RowKey.
6. При наличии более двух свойств ключей можно попытаться распределить их в две группы: первая группа будет PartitionKey, и вторая – RowKey. При таком подходе приложение должно знать, что PartitionKey, например, состоит из двух ключей, разделенных «-».

### Масштабируемость

Теперь, когда приложение имеет набор потенциальных ключей, необходимо убедиться, что выбранная схема секционирования является масштабируемой:

1. Исходя из статистических данных интенсивности использования приложения, определите, не приведет ли секционирование по выбранному выше PartitionKey к созданию слишком загруженных секций, которые не смогут эффективно обслуживаться одним сервером?  Проверить это можно, применив нагрузочное тестирование секции таблицы. Для этого создается тестовая таблица с использованием выбранных ключей и одна из ее секций подвергается пиковой нагрузке, полученной на основании предполагаемых полезной нагрузки. Это позволяет проверить, может ли секция таблицы обеспечить необходимую производительность приложения.
2. Если секция таблицы проходит нагрузочное тестирование, ключи выбраны правильно.
3. Если секция таблицы не проходит нагрузочного тестирования, найдите PartitionKey, который обеспечил бы более узкое подразделение сущностей. Для этого можно взять другой PartitionKey или изменить имеющийся (например, объединив его со следующим свойством ключа). Целью этой операции является создание большего количества секций, чтобы не возникало одной слишком большой или слишком загруженной секции.
4. Система спроектирована так, что обеспечивает необходимое масштабирование и обработку большого количества запросов. Но при чрезвычайно высокой интенсивности запросов ей приходится выполнять балансировку нагрузки, в результате чего некоторые из запросов могут завершаться ошибкой превышения времени ожидания. Сократить или устранить ошибки такого рода может снижение интенсивности запросов. Вообще говоря, такие ошибки возникают редко; однако если вы столкнулись с частыми или неожиданными ошибками превышения времени ожидания, свяжитесь с нами через форум Windows Azure на сайте MSDN, мы обсудим, как оптимизировать использование Windows Azure Table и предотвратить возникновение таких ошибок в вашем приложении.

### Гибкое секционирование

Также можно проанализировать расширяемость выбранных ключей, особенно если на момент их выбора нет точных сведений о характеристиках пользовательского трафика. В этом случае важно выбирать ключи, которые можно легко расширять для обеспечения более тонкого секционирования. Далее в данном документе приводится подробный пример этого.

# Программирование таблиц

Для таблиц и сущностей поддерживаются следующие базовые операции:

* Создание таблицы или сущности.
* Извлечение таблицы или сущности с применением фильтров.
* Обновление сущности (но не таблицы).
* Удаление таблицы или сущности.
* Транзакции над группами сущностей, которые поддерживают транзакции с множеством сущностей одной таблицы и секции.

Для работы с таблицами в .NET-приложении можно просто использовать ADO.NET Data Services.

В следующей таблице приведен список предлагаемых API. Поскольку применение ADO.NET Data Services в итоге сводится к передаче REST-пакетов, приложения могут использовать REST напрямую. Кроме того, что REST обеспечивает возможность доступа к хранилищу посредством не-.NET языков, он также позволяет реализовывать более тонкое управление сериализацией/десериализацией сущностей, что пригодится при работе с такими сценариями, как наличие разных типов сущностей, или если необходимо, чтобы объект имел большее количество свойств, чем допускается для данной сущности.

| **Операция** | **ADO.NET Data Services** | **HTTP-команда** | **Ресурс** | **Описание** |
| --- | --- | --- | --- | --- |
| Запрос | LINQ Query | GET | Таблица | Возвращает список всех таблиц данной учетной записи хранилища. В случае наличия фильтра таблицы возвращаются соответственно фильтру. |
| Сущность | Возвращает все сущности заданной таблицы или подмножество сущностей, если задан фильтр. |
| Обновление всей сущности | UpdateObject & SaveChanges(SaveChangesOptions.ReplaceOnUpdate) | PUT | Сущность | Обновляет значения свойств сущности. Операция PUT замещает всю сущность и может использоваться для удаления свойств. |
| Частичное обновление сущности | UpdateObject & SaveChanges() | MERGE | Сущность | Обновляет значения свойств сущности. |
| Создание новой сущности | AddObject & SaveChanges() | POST | Таблица | Создает новую таблицу в это учетной записи хранилища. |
| Сущность | Вставляет новую сущность в названную таблицу. |
| Удаление сущности | DeleteObject & SaveChanges() | DELETE | Таблица | Удаляет таблицу в данной учетной записи хранилища. |
| Сущность | Удаляет сущность из названной таблицы. |
| Транзакция над группой сущностей | SaveChanges(SaveChangesOptions.Batch) | POST | $batch | Поддержка транзакции над группой сущностей обеспечивается посредством пакетной операции над сущностями одной таблицы с одинаковым ключом секции. В ADO.NET Data Services опция SaveChanges требует, чтобы запрос выполнялся как одна транзакция. |

К расширенным операциям с таблицами относятся:

* Разбиение на страницы.
* Обработка конфликтов, возникающих в результате параллельных обновлений.

Более подробно мы поговорим о них позже.

## Контроль версий

Для всех решений Windows Azure Storage был введен новый HTTP-заголовок «x-ms-version». Все изменения API хранилища регистрируются как версии с помощью этого заголовка. Это обеспечивает возможность выполнять предыдущие версии команд и при этом расширять возможности существующих команд и вводить новые.

Заголовок «x-ms-version» должен быть задан для всех запросов к Windows Azure Storage. При поступлении анонимного запросе без указания версии система хранения выполнит самую старую из поддерживаемых версию этой команды.

К PDC 2009 мы планируем сделать заголовок «x-ms-version» обязательным для всех неанонимных команд. До тех пор, если версия команды в запросе не задана, предполагается, что должна быть выполнена CTP-версия API Windows Azure Storage, вышедшая к PDC 2008. Запрос с недействительной версией будет отклонен.

Текущей версией является «x-ms-version: 2009-04-14». Данный заголовок может использоваться для всех команд и запросов к Windows Azure Storage. В этой версии для Windows Azure Tables была введена функциональность транзакций над группами сущностей, и чтобы использовать эту новую возможность, в запросе обязательно должен быть задан заголовок с этой версией.

// добавим заголовок версии, используя событие SendingRequest. Этот заголовок

// располагается там же, где находился бы заголовок даты, но заголовок версии

// не является частью канонической строки,

// используемой для создания подписи

context.SendingRequest +=

new EventHandler<SendingRequestEventArgs>(

delegate(object sender, SendingRequestEventArgs requestArgs)

{

HttpWebRequest request = requestArgs.Request as HttpWebRequest;

request.Headers.Add(

"x-ms-date",

DateTime.UtcNow.ToString("R", CultureInfo.InvariantCulture));

request.Headers.Add("x-ms-version", "2009-04-14");

// ... добавляем заголовок авторизации, используя ключ доступа

});

## Пример

В приведенных ниже примерах описываются операции с таблицей «Blogs». В этой таблице хранятся блоги для приложения MicroBlogging.

В приложении MicroBlogging есть две таблицы: Channels (Каналы) и Blogs (Блоги). Имеется список каналов, блоги публикуются в определенном канале. Пользователи подписываются на каналы и ежедневно получают новые блоги этих каналов.

В данном примере рассмотрим только таблицу Blogs и приведем примеры следующих операций с ней:

1. Описание схемы таблицы
2. Создание таблицы
3. Вставка блога в таблицу
4. Получение списка блогов из таблицы
5. Обновление блога в таблице
6. Удаление блога из таблицы
7. Вставка множества блогов в таблицу

## Описание класса сущности для таблицы

Схема таблицы описывается как C#-класс. Такую модель использует ADO.NET Data Services. Схема известна только клиентскому приложению и упрощает доступ к данным. Сервер схему не применяет.

Рассмотрим описание сущностей Blog, хранящихся в таблице Blogs. Каждая сущность блога содержит следующие данные:

1. Имя канала (ChannelName) – канал, в котором размещается блог.
2. Дата размещения.
3. Текст (Text) – содержимое тела блога.
4. Рейтинг (Rating) – популярность этого блога.

Для данной таблицы «Blogs» мы выбрали в качестве PartitionKey имя канала и в качестве RowKey - дату размещения блога. PartitionKey и RowKey – ключи таблицы Blogs, они объявляются посредством атрибута класса DataServiceKey (Ключ сервиса данных). Таблица «Blogs» секционирована по именам каналов (ChannelName). Это позволяет приложению эффективно извлекать самые недавние блоги канала, на который подписан пользователь. Кроме ключей, в качестве свойств объявлены характерные для пользователя атрибуты. Все свойства имеют открытые (public) методы считывания и присвоения значения и хранятся в таблице Windows Azure Table. Итак, в примере ниже:

* Text и Rating хранятся для экземпляра сущности в таблице Azure.
* RatingAsString нет, потому что для него не определен метод присвоения значения.
* Id не хранится, потому что методы доступа не public.

[DataServiceKey("PartitionKey", "RowKey")]

public class Blog

{

// ChannelName

public string PartitionKey { get; set; }

// PostedDate

public string RowKey { get; set; }

// Определяемые пользователем свойства

public string Text { get; set; }

public int Rating { get; set; }

public string RatingAsString { get; }

protected string Id { get; set; }

}

## Создание таблицы

Далее рассмотрим, как создать таблицу Blogs для учетной записи хранилища. Создание таблицы аналогично созданию сущности в основной таблице «Tables». Эта основная таблица определена для каждой учетной записи хранилища, и имя каждой таблицы, используемой учетной записью хранения, должно быть зарегистрировано в основной таблице. Описание класса основной таблицы приведено ниже, где свойство TableName (Имя таблицы) представляет имя создаваемой таблицы.

[DataServiceKey("TableName")]

public class TableStorageTable

{

public string TableName { get; set; }

}

Фактическое создание таблицы происходит следующим образом:

// Uri сервиса: “http://<Account>.table.core.windows.net/”

DataServiceContext context = new DataServiceContext(serviceUri);

TableStorageTable table = new TableStorageTable("Blogs");

// Создаем новую таблицу, добавляя новую сущность

// в основную таблицу "Tables"

context.AddObject("Tables", table);

// результатом вызова SaveChanges является отклик сервера

DataServiceResponse response = context.SaveChanges();

serviceUri – это uri сервиса таблицы, http://<Здесь указывается имя учетной записи>.table.core.windows.net/. DataServiceContext (Контекст сервиса данных) – один из основных классов сервиса данных ADO.NET, представляющий контекст времени выполнения для сервиса. Он обеспечивает API для вставки, обновления, удаления и запроса сущностей с помощью либо LINQ, либо RESTful URI и сохраняет состояние на стороне клиента.

## Вставка Blog

Чтобы вставить сущность, приложение должно выполнить следующее.

1. Создать новый C#-объект и задать все свойства.
2. Создать экземпляр DataServiceContext, который представляет подключение к серверу в сервисе данных ADO .NET для вашей учетной записи хранилища.
3. Добавить C#-объект в контекст.
4. Вызвать метод SaveChanges (Сохранить изменения) объекта DataServiceContext для отправки запроса серверу. Это обеспечивает отправку на сервер HTTP-запроса с сущностью в XML-формате ATOM.

Далее представлены примеры кода для перечисленных выше операций:

Blog blog = new Blog {

PartitionKey = "Channel9", // ChannelName

RowKey = DateTime.UtcNow.ToString(), // PostedDate

Text = "Hello"**,**

Rating = 3

};

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

var context = new DataServiceContext(serviceUri);

context.AddObject("Blogs", blog);

DataServiceContext response = context.SaveChanges();

## Запрос Blogs

Запрос сущностей выполняется с помощью встроенного в C# языка запросов LINQ (Language Integrated Query). В данном примере извлечем все блоги, рейтинг которых равен 3.

При обработке запроса (например, с помощью выражение foreach), он передается на сервер. Сервер отправляет результаты в XML-формате ATOM. Клиентская библиотека ADO .NET Data Services десериализует результаты в C#-объекты, после чего они могут использоваться приложением.

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

DataServiceContext context = new DataServiceContext(serviceUri);

// LINQ-запрос с использованием DataServiceContext для выбора

// из таблицы Blogs всех сущностей блогов, для которых rating = 3

var blogs =

from blog in context.CreateQuery<Blog>("Blogs")

where blogs.Rating == 3

select blog;

// запрос отправляется на сервер и выполняется

foreach (Blog blog in blogs) { }

## Обновление Blog

Обновление сущности выполняется следующим образом.

1. Создается DataContext (Контекст данных), свойству MergeOption (Вариант объединения) которого задается значение OverwriteChanges (Перезапись изменений) или PreserveChanges (Сохранение изменений), как описывается в разделе 4.10. Это обеспечивает правильную обработку ETag для каждого извлекаемого объекта.
2. С помощью LINQ DataContext получает сущность, которая будет обновляться. Извлечение ее с сервера гарантирует обновление ETag в сущностях, отслеживаемых контекстом, и то, что при последующих обновлениях и удалениях в заголовке if-match будет использоваться обновленный ETag. Меняем C#-объект, представляющий сущность.
3. Возвращаем C#-объект в тот же DataContext для обновления. Использование того же DataContext гарантирует автоматическое повторное использование ETag, полученного ранее для этого объекта.
4. Вызываем метод SaveChanges для отправки запроса на сервер.

Blog blog =

(from blog in context.CreateQuery<Blog>("Blogs")

where blog.PartitionKey == "Channel9"

&& blog.RowKey == "Oct-29"

select blog).FirstOrDefault();

blog.Text = "Hi there";

context.UpdateObject(blog);

DataServiceResponse response = context.SaveChanges();

## Удаление Blog

Удаление сущности аналогично ее обновлению. Для этого извлекаем сущность с помощью DataServiceContext и вызываем для содержимого вместо метода UpdateObject метод DeleteObject (Удалить объект).

// Получаем объект Blog для ("Channel9", "Oct-29")

context.DeleteObject(blog);

DataServiceResponse response = context.SaveChanges();

## Транзакции над группами сущностей

Транзакции над группами сущностей поддерживают выполнение до 100 команд CUD в одном пакете. Эти команды выполняются неразрывно, т.е. либо успешно завершаются все, либо ни одна.

За исключением определенных ограничений, мы следуем протоколу, используемому сервисом данных ADO.NET для пакетных транзакций (подробнее по адресу http://msdn.microsoft.com/en-us/library/cc668802.aspx). Рассмотрим некоторые характерные для пакетов термины:

* Массив изменений – Группа из одной или более команд CUD, выполняемых неразрывно.
* Пакет – Пакет включает один или более массив изменений или запрос.

На применение пакетных команд для Windows Azure Table налагаются следующие ограничения:

* Массив изменений может включать не более 100 операций.
* Допускается всего один массив изменений или запрос на пакетную операцию. Запрос не может быть объединен с операциями CUD. Поддерживаются только запросы, детерминировано возвращающие одну строку, т.е. для которых PartitionKey и RowKey определены с помощью '='
* Максимальный размер пакета – 4MБ.
* Все команды массива изменений должны касаться одной секции одной таблицы. Это означает, что все сущности должны иметь одно и то же значение ключа секции и операции должны выполняться с одной таблицей.
* Каждая сущность может встречаться в массиве изменений только один раз. Массив изменений не может осуществлять множество операций с одной и той же сущностью. Массив изменений предназначен для выполнения множества операций над разными сущностями одной секции и таблицы в неразрывной транзакции.
* Отдельные команды массива изменений выполняются в порядке, в котором они указаны в массиве изменений.

При выполнении транзакции ETag передается не как заголовок всего запроса, а рассылается в каждый массив изменений, как описывается в документации MSDN. На сервере проверяем, соответствует ли ETag значению, хранящемуся на сервере. Если нет, транзакция дает сбой.

Следующий фрагмент кода продолжает пример blob, демонстрирующий применение транзакции над группой сущностей для атомарного создания, обновления и удаления множества сущностей секции за одну пакетную операцию.

// newBlogs – список новых блогов в секции Channel\_19

// deletedBlogs – список блогов, которые требуется удалить из секции Channel\_19

// updatedBlogs – список блогов секции Channel\_19, которые необходимо обновить

// Предполагаем, что deletedBlogs и updatedBlogs были выбраны

// в контекст в предыдущих запросах.

// Вставляем множество объектов blog в секцию "Channel\_19".

// newBlogs – список блогов, которые были выбраны для вставки

// вне этой функции

for (int index = 0; index < newBlogs.Length; index++)

{

context.AddObject(newBlogs[index]);

}

// Удаляем старые объекты blog из секции Channel\_19.

// deletedBlogs – список блогов, которые были выбраны для удаления

// вне этой функции

for (int index = 0; index < deletedBlogs.Length; index++)

{

context.DeleteObject(deletedBlogs[index]);

}

// Обновляем существующие объекты blog секции Channel\_19.

// updatedBlogs – список блогов, которые были выбраны для

// увеличения рейтинга (increase the rating)

for (int index = 0; index < updatedBlogs.Length; index++)

{

updatedBlogs[index].Rating++;

context.UpdateObject(updatedBlogs[index]);

}

// Все приведенные выше операции CUD выполняются как один пакетный запрос.

DataServiceResponse response =

context.SaveChanges(SaveChangesOptions.Batch);

Задание SaveChangeOptions.Batch указывает команде SaveChanges, что все ожидающие подтверждения изменения группируются в один массив изменений и передаются в систему хранения как единая Entity Group Transaction. Пакетная транзакция проверяется на сервере на соответствие всем перечисленным выше ограничениям.

Если SaveChanges описана без SaveChangesOptions.Batch, операции Create/Update/Delete будут передаваться и выполняться как отдельный запрос для каждой сущности.

Следующий фрагмент кода демонстрирует один запрос пакетной команды. Обратите внимание, что пакетная команда может быть либо Entity Group Transaction, как показано выше, либо отдельным запросом, как показано ниже. Это полезно, поскольку позволяет выполнять запрос (Get) к одной строке, используя формат URI, представляющий одну сущность, что может использоваться для обхода ограничения размера сегмента URI (не более 260 символов), налагаемого HTTP.sys (Раздел 2).

// Выполняем запрос в пакете – обратите внимание,

// должен присутствовать фильтр в форме:

// PartitionKey == 'Some Value' && RowKey == 'Some Value'

var q1 = from o in context.CreateQuery<Blogs>("Blogs")

where o.PartitionKey == "Channel\_19" &&

o.RowKey == "2"

select o;

DataServiceResponse response =

context.ExecuteBatch((DataServiceQuery<RetailStoreV1>)q1);

Примеры можно найти в документации MSDN.

ПРИМЕЧАНИЕ: Для обхода этой проблемы без применения пакетных запросов можно сформировать запрос, используя приведенный ниже синтаксис. В этом случае фильтры, ключ секции и ключ строки, определены не в сегменте URI, в как часть параметра запроса $filter.

from o in context.CreateQuery<Blogs>("Blogs")

where o.PartitionKey == "Channel\_19"

select o into d

where d.RowKey == "2"

select d;

### Обработка ответа

Если пакетный запрос принят и обработан сервером, ответом на него всегда будет код возврата 202 Accepted. Ответы на отдельные операции пакета возвращаются в элементе ответа как полезная нагрузка пакета. По сути, если ни одна операция массива изменений не дала сбой, ответ на пакетную операцию один к одному соответствует запросу. Если же одна из операций дает сбой, для массива изменений возвращается единый ответ, потому что операции массивов изменений являются неразрывными.

* Если все операции массива изменений выполнены успешно, возвращается ответ для каждой операции массива изменений. Если какая-либо операция массива дает сбой, возвращается всего один объект ответа, представляющий единый ответ для всех операций массива изменений.
* Передача HTTP-тегов eTag осуществляется теми же механизмами, которые описаны в дополнении к основному протоколу спецификации нежесткой блокировки, где элемент ответа играет роль HTTP-заголовков ответа для заданной операции.

### Ошибки

При сбое пакетного запроса не выполняется ни одна из его операций. Тем не менее, приложению важно знать, какая из команд привела к сбою всей транзакции. Для этого в сообщении об ошибке возвращается порядковый номер команды, как видно в выделенной строке приведенного ниже примера ответа. Но если ошибка возникает из-за проблем с авторизацией или по какой-то другой причине, не связанной с какой-то определенной командой, порядковый номер не предоставляется. Таким образом, полезная нагрузка в случае сбоя будет включать:

1. Ключ операции, давшей сбой.
2. Индекс (начиная от 0) операции пакета, обусловившей сбой.
3. Строку, описывающую сбой.

Пример ответа:

HTTP/1.1 202 Accepted

Cache-Control: no-cache

Transfer-Encoding: chunked

Content-Type: multipart/mixed; boundary=batchresponse\_7ab1553a-7dd6-44e7-8107-bf1ea1ab1876

Server: Table Service Version 1.0 Microsoft-HTTPAPI/2.0

x-ms-request-id: 45ac953e-a4a5-42ba-9b4d-97bf74a8a32e

Date: Thu, 30 Apr 2009 20:45:13 GMT

6E7

--batchresponse\_7ab1553a-7dd6-44e7-8107-bf1ea1ab1876

Content-Type: multipart/mixed; boundary=changesetresponse\_6cc856b4-8cb9-41eb-b8d2-bb73475c6cec

--changesetresponse\_6cc856b4-8cb9-41eb-b8d2-bb73475c6cec

Content-Type: application/http

Content-Transfer-Encoding: binary

HTTP/1.1 400 Bad Request

Content-ID: 4

Content-Type: application/xml

Cache-Control: no-cache

DataServiceVersion: 1.0;

<?xml version="1.0" encoding="utf-8" standalone="yes"?>

<error xmlns="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata">

<code>InvalidInput</code>

<message xml:lang="en-US">1:One of the request inputs is not valid.</message>

</error>

--changesetresponse\_6cc856b4-8cb9-41eb-b8d2-bb73475c6cec--

--batchresponse\_7ab1553a-7dd6-44e7-8107-bf1ea1ab1876--

0

При возникновении исключения получить порядковый номер (выделенная строка в примере выше) команды, ставшей причиной сбоя, можно следующим образом:

try

{

// ... сохраняем изменения

}

catch (InvalidOperationException e)

{

DataServiceClientException dsce = e.InnerException as DataServiceClientException;

int? commandIndex;

string errorMessage;

ParseErrorDetails(dsce, out commandIndex, out errorMessage);

}

void ParseErrorDetails(

DataServiceClientException e,

out string errorCode,

out int? commandIndex,

out string errorMessage)

{

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

commandIndex = null;

int indexOfSeparator = errorMessage.IndexOf(':');

if (indexOfSeparator > 0)

{

int temp;

if (Int32.TryParse(errorMessage.Substring(0, indexOfSeparator), out temp))

{

commandIndex = temp;

errorMessage = errorMessage.Substring(indexOfSeparator + 1);

}

}

}

void GetErrorInformation(

string xmlErrorMessage,

out string errorCode,

out string message)

{

message = null;

errorCode = null;

XName xnErrorCode = XName.Get(

"code",

"http://schemas.microsoft.com/ado/2007/08/dataservices/metadata");

XName xnMessage = XName.Get(

"message",

"http://schemas.microsoft.com/ado/2007/08/dataservices/metadata");

using (StringReader reader = new StringReader(xmlErrorMessage))

{

XDocument xDocument = null;

try

{

xDocument = XDocument.Load(reader);

}

catch (XmlException)

{

// Не удалось выполнить синтаксический разбор XML. Это могло произойти из-за

// невозможности установления подключения к серверу, или если ответ не содержал

// сведений об ошибке (например, если код состояния ответа не был ни кодом ошибки,

// ни кодом успешного завершения, а 3XX-кодом, таким как NotModified.

return;

}

XElement errorCodeElement =

xDocument.Descendants(xnErrorCode).FirstOrDefault();

if (errorCodeElement == null)

{

return;

}

errorCode = errorCodeElement.Value;

XElement messageElement =

xDocument.Descendants(xnMessage).FirstOrDefault();

if (messageElement != null)

{

message = messageElement.Value;

}

}

}

## Рекомендации по работе с DataServiceContext

Рассмотрим рекомендации по работе с DataServiceContext:

* Объект DataServiceContext не обеспечивает безопасность потоков, поэтому он не может использоваться совместно разными потоками. DataServiceContext не является объектом с длительным временем жизни. Вместо того чтобы использовать один DataServiceContext в течение всей жизни потока, рекомендуется создавать объект DataServiceContext каждый раз, когда возникает необходимость выполнить ряд транзакций с WindowsAzureTable, и затем удалять этот объект.
* Если для всех вставок/обновлений/удалений используется один экземпляр DataServiceContext и возникает сбой при выполнении SaveChanges, сведения об операции, давшей сбой, сохраняются в DataServiceContext. При последующем вызове SaveChanges попытка выполнить эту операцию повторяется.
* DataServiceContext имеет свойство MergeOption, которое используется для управления тем, как DataServiceContext обрабатывает отслеживаемые сущности. Возможные значения:
  + AppendOnly (Только добавление): Это значение по умолчанию, при использовании которого DataServiceContext не загружает экземпляр сущности с сервера, если он уже имеется в его кэше.
  + OverwriteChanges: DataServiceContext всегда загружает экземпляр сущности с сервера и перезаписывает предыдущий вариант сущности, т.е. обеспечивает соответствие экземпляра сущности ее текущему состоянию.
  + PreserveChanges: Если экземпляр сущности существует в DataServiceContext, он не загружается из постоянного хранилища. Все изменения, вносимые в свойства объектов в DataServiceContext, сохраняются, но ETag обновляется, поэтому данную опцию следует использовать при необходимости восстановления после ошибок совместного доступа с нежесткой блокировкой.
  + NoTracking (Без отслеживания): DataServiceContext не отслеживает экземпляры сущностей. Обновление сущности в контексте без отслеживания реализуется с помощью Etag, который обновляется посредством AttachTo. Этот вариант не рекомендуется к применению.

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

context.UpdateObject(blog);

context.SaveChanges();

Когда MergeOption контекста задано значение AppendOnly и объект DataServiceContext уже отслеживает сущность в результате предыдущей операции извлечения или добавления, повторное извлечение сущности с сервера не приведет к обновлению отслеживаемой сущности в контексте. Таким образом, если сущность на сервере была изменена, последующие обновления/удаления приведут к сбою необходимых условий (PreCondition). В примере кода раздела 5 MergeOption задано значение PreserveChanges, которое обеспечивает, что сущность будет загружаться с сервера всегда.

* Из-за известной проблемы с производительностью клиентской библиотеки ADO.NET Data Services при описании класса рекомендуется использовать имя таблицы или определять делегат ResolveType для DataServiceContext. Если этого не сделать и если имя класса не соответствует имени таблицы, производительность запроса деградирует с увеличением количества сущностей, возвращаемых в ответе. Рассмотрим пример использования делегата ResolveType:

public void Query(DataServiceContext context)

{

// задаем в качестве ResolveType метод, который будет возвращать тип,

// который должен быть создан

context.ResolveType = this.ResolveEntityType;

...

}

public Type ResolveEntityType(string name)

{

// если контекст включает всего один тип, он может быть

// возвращен без проверки значения свойства "name". В противном случае, проверяем

// имя и возвращаем соответствующий тип

// (возможно, сопоставление Dictionary<string, Type> будет полезным)

Type type = typeof(Customer);

return type;

}

## Использование API REST

Результатом всех рассматриваемых выше операций является передача HTTP-сообщений на и с сервера. Приложение может отказаться от использования клиентской библиотеки .NET и работать на уровне HTTP/REST.

# Параллельные обновления

Для обновления сущности необходимо выполнить следующие операции.

1. Получить сущность с сервера
2. Обновить объект локально и вернуть его на сервер.

Предположим, два процесса, выполняющихся параллельно, пытаются обновить одну и ту же сущность. Поскольку шаги 1 и 2 не являются неделимыми, на любом из них может возникнуть ситуация внесения изменений в уже устаревшую версию сущности. Для решения этой проблемы Windows Azure Table использует нежесткую блокировку.

1. Для каждой сущности система сохраняет версию, которая изменяется сервером при каждом обновлении.
2. При извлечении сущности, сервер отправляет эту версию клиенту в виде ETag HTTP.
3. Когда клиент передает запрос UPDATE (обновить) на сервер, он отправляет на него этот ETag в виде заголовка If-Match.
4. Если версия сущности, хранящаяся на сервере, аналогична ETag в заголовке If-Match, изменение принимается, и хранящаяся на сервере сущность получает новую версию. Эта новая версия возвращается клиенту как заголовок ETag.
5. Если версия сущности на сервере отличается от ETag в заголовке If-Match, изменение отклоняется, и клиенту возвращается HTTP-ошибка «precondition failed» (необходимое условие не выполнено).

При получении ошибки «precondition failed» типовым поведением клиентского приложения будет повторение всей операции, как показано в фрагменте кода ниже.

1. Приложение должно извлечь этот объект снова, т.е. получить его последнюю версию.
2. Обновить объект локально и вернуть его на сервер.

При использовании клиентской библиотеки .NET приложение получает HTTP-код ошибки в виде исключения DataServiceRequestException.

В примере ниже два разных клиента выполняют один и тот же код для изменения текста. Эти два клиента пытаются задать свойству «Text» разные значения. Рассмотрим возможную последовательность событий, иллюстрирующую обработку параллельных обновлений.

1. Оба клиента извлекают сущность. При этом для каждой сущности извлекается ETag, например, «v1». Оба клиента полагают, что предыдущая версия сущности – «v1».
2. Каждый клиент локально обновляет свойство Text.
3. Каждый клиент вызывает методы UpdateObject и SaveChanges.
4. Каждый клиент отправляет на сервер HTTP-запрос с заголовком «If-Match: v1».
5. Запрос одного из клиентов попадает на сервер первым.
   1. Сервер сравнивает заголовок «If-Match» с версией сущности. Они совпадают.
   2. Сервер применяет изменение.
   3. Версия сущности на сервере обновляется и становится «v2».
   4. В качестве ответа клиенту отправляется новый заголовок «ETag:v2».
6. Далее на сервер поступает запрос другого клиента. На этот момент изменения первого клиента уже применены.
   1. Сервер сравнивает заголовок «If-Match» с версией сущности. Они не совпадают, поскольку версия сущности уже изменена на «v2», тогда как в запросе указывается версия «v1».
   2. Сервер отклоняет запрос.

// Задаем такой вариант объединения, который обеспечивает

// сохранение обновлений, но позволяет обновление etag.

// По умолчанию применяется значение AppendOnly, при котором

// уже отслеживаемая сущность не перезаписывается значениями,

// полученными с сервера, в результате чего в случае изменения

// сущности на сервере используется недействительный etag.

context.MergeOption = MergeOption.PreserveChanges;

Blog blog =

(from blog in context.CreateQuery<Blog>("Blogs")

where blog.PartitionKey == "Channel9"

&& blog.RowKey == "Oct-29"

select blog).FirstOrDefault();

blog.Text = "Hi there again";

try

{

context.UpdateObject(blog);

DataServiceResponse response = context.SaveChanges();

}

catch (DataServiceRequestException e)

{

OperationResponse response = e.Response.First();

if (response.StatusCode == (int)HttpStatusCode.PreconditionFailed)

{

// выполняем запрос объекта повторно, чтобы получить

// последний etag, и проводим обновление

}

}

## Безусловные обновления

Для безусловного обновления сущности приложение выполняет следующее:

1. Создает новый объект DataServiceContext или, в случае использования существующего контекста, отсоединяет объект, как демонстрирует пример ниже.
2. Присоединяем сущность к контексту и используем «\*» как новое значение ETag.
3. Обновляем сущность.
4. Вызываем SaveChanges.

// задаем опцию объединения, разрешающую перезапись,

// чтобы обеспечить возможность обновления отслеживаемой сущности

context.Detach(blog);

// Присоединяем сущность к контексту, используя имя таблицы, сущность,

// которая должна быть обновлена, и "\*" как значение etag.

context.AttachTo("Blogs", blog, "\*");

blog.Text = "Hi there again";

try

{

context.UpdateObject(blog);

DataServiceResponse response = context.SaveChanges();

}

catch (DataServiceRequestException e)

{

// Обработка ошибки, но в данном случае формировнаие ошибки PreCondition невозможно

}

# Разбиение на страницы результатов запроса

Для запросов, которые могут возвращать большое количество результатов, система обеспечивает два механизма:

1. Возможность получать первые N сущностей, используя LINQ-функцию Take(N).
2. Маркер продолжения, который обозначает место начала следующего множества результатов.

## Получение первых N сущностей

Система поддерживает функцию возвращения первых N соответствующих запросу сущностей. Например, в .NET, для извлечения первых N сущностей (в данном примере это первые 100 сущностей) можно использовать LINQ-функцию Take(N).

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

DataServiceContext svc = new DataServiceContext(serviceUri);

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

foreach (Blog blog in allBlogs.Take(100))

{

// выполняем некоторые операции с каждым блогом

}

Аналогичная функциональность поддерживается в интерфейсе REST через опцию строки запроса $top=N. Например, запрос «GET http://<UriСервиса>/Blogs?$top=100» обеспечил бы возвращение первых 100 сущностей, соответствующих запросу. Фильтрация выполняется на сервере, поэтому в ответе клиенту может быть передано максимум 100 сущностей.

## Маркеры продолжения

Количество возвращаемых запросом сущностей может быть ограничено по одной из следующих причин:

1. В запросе указывается максимальное число сущностей, которое должно быть возвращено.
2. Количество сущностей превышает максимально разрешенное сервером число сущностей в ответе (в настоящее время это 1000 сущностей).
3. Общий размер сущностей в ответе превышает максимально допустимый размер ответа (в настоящее время это 4МБ, включая имена свойств, но исключая xml-теги, используемые для REST).
4. На выполнение запроса требуется больше времени, чем заданный период ожидания сервера (в настоящее время это 60 секунд).

В любом из этих случаев ответ будет включать маркер продолжения в виде специального заголовка. Для запроса к вашим сущностям используются такие специальные заголовки, представляющие маркер продолжения:

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

Если клиент получил эти значения, он должен передать их со следующим запросом в виде опций HTTP-запроса; во всем остальном запрос остается неизменным. Это обеспечит возвращение следующего набора сущностей, начинающегося с места, обозначенного маркером продолжения.

Последующий запрос выглядит следующим образом:

http://<UriСервиса>/Blogs?<исходныйЗапрос>&NextPartitonKey=<некотороеЗначение>&NextRowKey=<другоеЗначение>

Это повторяется до тех пор, пока клиентом не будет получен ответ без маркера продолжения, что свидетельствует об извлечении всех соответствующих запросу результатов.

Маркер продолжения должен рассматриваться как непрозрачное значение. Оно указывает на точку начала следующего запроса и может не соответствовать фактической сущности в таблице. Если в таблицу добавляется новая сущность, так что Key(новая сущность) > Key(последняя сущность, извлеченная запросом), но Key(новая сущность) < «Маркера продолжения», тогда эта новая сущность не будет возвращена повторным запросом, использующим маркер продолжения. Но новые сущности, добавленные так, что Key(новая сущность) > «Маркера продолжения», войдут в результаты, возвращаемые последующими использующими маркер продолжения запросами.

# Модель согласованности

Теперь рассмотрим модель согласованности, обеспечиваемую Windows Azure Table.

## Согласованность в рамках одной таблицы

В рамках одной таблицы система обеспечивает гарантии транзакции ACID для всех операций вставки/обновления/удаления для одной сущности.

Для запросов в рамках отдельной секции выполняется изоляция моментального снимка. Запрос обеспечивается согласованным представлением секции с момента его начала и в течение всей транзакции. Моментальный снимок обеспечивает следующее:

1. Отсутствие «грязного считывания». Транзакция не будет видеть незафиксированные изменения, вносимые другими транзакциями, которые выполняются параллельно. Будут представлены только те изменения, которые были завершены до начала выполнения запроса на сервере.
2. Механизм изоляции моментального снимка позволяет производить чтение параллельно с обновлением секции без блокирования этого обновления.

Изоляция моментального снимка поддерживается только в рамках секции и в рамках одного запроса. Система не поддерживает изоляцию моментального снимка для нескольких секций таблицы или других фаз запроса.

## Согласованность таблиц

Приложения отвечают за сохранение согласованности между множеством таблиц.

В примере MicroBlogging использовалось две таблицы: Channels и Blogs. Приложение выполняет согласование таблиц Channels и Blogs. Например, когда канал удаляется из таблицы Channels, приложение должно удалить соответствующие блоги из таблицы Blogs.

Сбои могут возникать в моменты синхронизации состояния множества таблиц. Приложение должно уметь обрабатывать такие сбои и иметь возможность возобновлять работу с момента, на котором она была прервана.

В предыдущем примере, когда канал удаляется из таблицы каналов, приложение должно также удалить все блоги этого канала из таблицы Blogs. В ходе этого процесса могут возникать сбои приложения. Для обработки таких сбоев приложение может сохранять транзакцию в Windows Azure Queues, что позволяет пользователю возобновить операцию удаления канала и всех его блогов даже в случае сбоя.

Вернемся к примеру с таблицами «Channels» и «Blogs». Сущность таблицы «Channels» имеет следующие свойства: Name (Имя) как PartitionKey, пустая строка как RowKey, Owner (Владелец), CreatedOn (Дата создания). И сущность таблицы «Blogs» имеет свойства Channel Name (Имя канала) как PartitionKey, CreatedOn как RowKey, Title (Название), Blog, UserId. Теперь, когда канал удален, необходимо обеспечить, чтобы все ассоциированные с ним блоги также были удалены. Для этого выполняем следующие шаги:

1. Создаем очередь для обеспечения согласованности таблиц, назовем ее «DeleteChannelAndBlogs» (Удаление каналов и блогов).
2. При поступлении запроса на удаление канала от роли веб-интерфейса, ставим в созданную выше очередь элемент, определяющий имя канала.
3. Создаем рабочие роль, которые будут ожидать событие добавления элемента в очередь «DeleteChannelAndBlogs».
4. Рабочая роль изымает элемент из очереди DeleteChannelAndBlogs, задавая для извлеченного элемента очереди время невидимости в течение N секунд. При этом элемент, определяющий имя канала, который должен быть удален, изымается. Если роль работника удаляет элемент очереди в течение этих N секунд, данный элемент будет удален из очереди. Если нет, элемент станет вновь видимым и доступным для использования рабочей ролью. При извлечении элемента рабочая роль делает следующее:
   1. В таблице Channels помечает канал как недействительный, чтобы с этого момента никто не мог выполнять чтение из него.
   2. Удаляет из таблицы Blogs все записи, для которых PartitionKey = “имени канала”, указанному в элементе очереди.
   3. Удаляет канал из таблицы Channels.
   4. Удаляет элемент из очереди.
   5. Возвращается.

Предположим, в ходе выполнения шага 4 возникает какой-либо сбой, например, аварийное завершение рабочего процесса, и элемент не удаляется из очереди. Как только соответствующий элемент очереди станет снова видимым (т.е. когда истечет время, заданное как время ожидания видимости), это сообщение будет вновь извлечено из очереди рабочим процессом, и процесс удаления возобновится с шага 4. Более подробно обработка очередей рассматривается в документации Windows Azure Queue.

# Советы и рекомендации

В данном разделе будут рассмотрены некоторые обычные сценарии, которые могут возникать при работе приложения, и возможные их решения.

## Извлечение элементов, добавленных последними (моделирование расположения элементов в порядке по убыванию)

В приложениях для работы с электронными сообщениями, новостями, блогами и т.д. элемент, добавленный самым последним, обычно необходимо отображать первым. Учитывая, что все строки сортируются по PartitionKey и RowKey, очень маловероятно, что первым в списке будет стоять элемент, созданный позже других.

Для выполнения этого требования в качестве RowKey можно задать строку фиксированной длины, равную DateTime.MaxValue.Ticks - DateTime.UtcNow.Ticks (максимальное значение времени – текущее значение времени). Это позволяет RowKey сортировать элементы соотвественно смещению по времени, начиная с элементов, опубликованных последними.

Например:

Blog blog= new Blog();

// Заметьте, что используется фиксированная длина 19 потому, что размер максимального значения тика – 19 разрядов.

string rowKeyToUse = string.Format("{0:D19}",

DateTime.MaxValue.Ticks - DateTime.UtcNow.Ticks);

blog.RowKey = rowKeyToUse;

Итак, RowKey блога b1, датированного 10/1/2008 10:00:00 AM, будет равен 2521794455999999999, и RowKey блога b2, датированного 10/2/2008 10:00:00 AM, будет равен 2521793591999999999. Следовательно, b2 будет располагаться выше b1.

Для извлечения всех блогов, датированных после 10/1/2008 10:00:00 AM, будет использоваться следующий запрос:

string rowKeyToUse = string.Format("{0:D19}",

DateTime.MaxValue.Ticks - DateTime.UtcNow.Ticks);

var blogs =

from blog in context.CreateQuery<Blog>("Blogs")

where blog.PartitionKey == "Football"

&& blog.RowKey.CompareTo(rowKeyToUse) > 0

select blog;

ПРИМЕЧАНИЕ: Фиксированная длина строки должна быть достаточной для обработки большого диапазона значений. Например, максимальная длина DateTime.MaxValue.Ticks - DateTime.UtcNow.Ticks составляет 19 разрядов. Однако для дат, начиная с 2/15/6831 2:13:20 PM, эта длина 18 разрядов. Поскольку сортировка лексикографическая, если не добавить '0' в начале строки, сортировка в этом случае обеспечит нежелательные результаты, т.е. 92 > 10000, но если дополнить это значение нулями соответствующим образом, получим 10000 > 00092.

Для CTP-версии допускается только один RowKey. Но, фактически, посредство конкатенации строк RowKey может представлять несколько ключей. Например, предположим, блог может быть рейтингован по шкале от 0 до 5, и требуется, чтобы блоги были сортированы по рейтингу и по времени создания в убывающем порядке, а именно (Rating in desc order, Created time in desc order). Это означает, что сначала блоги сортируются по рейтингу и затем в рамках каждого рейтинга по времени создания. Здесь ключ строки может быть задан как <Рейтинг>+<Разряды, обеспечивающие сортировку элементов в порядке от более свежих до более старых записей>. Итак, возьмем тот же пример, что и ранее, с двумя блогами, и предположим, что blog1 получил рейтинг 5, и blog2 – рейтинг 1. Поскольку в фильтре сортировки рейтинг идет раньше разрядов, blog1 будет располагаться в списке перед blog2, несмотря на то что blog2 был размещен позднее blog1.

PartitionKey и RowKey строкового типа, но соблюдение следующих правил позволит применять в этих свойствах данные и других типов. Также это гарантирует сортировку значений в ожидаемом порядке. Например:

* Integer – сохраняйте как значение фиксированного размера, в случае необходимости дополняя его нулями спереди.
* DateTime – сохраняйте в формате yyyy-mm-dd или yyyy-mm-dd-hh-ss, если требуется более точное указание даты и времени. Если более свежие сущности должны располагаться в списке выше более старых, используйте разряды, как описывалось ранее.

## Извлечение с использованием префикса

Сопоставление префиксов не поддерживается, но оно может быть сымитировано применением к свойству фильтра >= & <. Например, если у каждого PDC есть собственный канал, и требуется извлечь все блоги этого PDC, запрос должен обеспечивать извлечение всех блогов всех каналов, в которых PartitionKey начинается с «PDC». Это можно осуществить следующим запросом:

var blogs =

from blog in context.CreateQuery<Blog>("Blogs")

where blog.PartitionKey.CompareTo("PDC") >= 0

&& blog.PartitionKey.CompareTo("PDD") < 0

select blog;

## Пример секционирования данных

Секционирование данных заключается в группировке сущностей таблицы в секции, так что секция становится наименьшим элементом, используемым для балансировки нагрузки. Группировку сущностей в секции Windows Azure Table осуществляет посредством PartitionKey. В данном разделе мы рассмотрим, как секционировать данные для обеспечения лучшей производительности.

### Микроблоггинг

Для примера возьмем приложение микроблоггинга, в котором пользователи могут создавать каналы (или категории) и блоги в них. При выборе канала пользователем на экран выводится страница блогов.

Предположим, что блог в таблице Blogs уникально идентифицирован именем канала и датой создания. Будем оптимизировать таблицу для преобладающего запроса «Get me the 10 most recent blogs for a channel» (Найти 10 последних блогов канала).

В данном примере рассматриваются различные варианты выбора PartitionKey, обеспечивающие формирование секций различного размера, от очень больших до очень маленьких, вплоть до секции, включающей всего одну сущность. Таким образом, размеры и число секций в таблице может быть различным.

#### Несколько больших секций

Очевидный выбор для использования в качестве PartitionKey – имя канала. Также применим метод, описанный в разделе 8.1, что обеспечит сортировку блогов по RowKey в порядке публикации, начиная от опубликованного последним. Ниже представлено описание класса сущности для таблицы Blogs, в котором реализован данный подход к секционированию:

[DataServiceKey("PartitionKey", "RowKey")]

public class Blog

{

// Имя канала

public string PartitionKey { get; set; }

// Для сортировки по RowKey, начиная от самого недавнего, заканчивая

// опубликованным раньше всех блогом, RowKey определяем как

// DateTime.MaxValue.Ticks - DateTime.UtcNow.Ticks

public string RowKey { get; set; }

// Определяемые пользователем свойства

public string Text { get; set; }

public DateTime CreatedOn { get; set; }

public int Rating { get; set; }

}

Это позволяет эффективно запрашивать и выводить на экран все блоги канала «Football», сортируя их по дате публикации, начиная с самого недавнего, на основании заданного выше RowKey.

var blogs =

from blog in context.CreateQuery<Blog>("Blogs")

where blog.PartitionKey == "Football"

select blog;

foreach (Blog blog in blogs) { }

При такой схеме секционирования элементом для балансировки нагрузки является канал. Таким образом, блоги разных каналов располагаются в разных секциях, эти секции могут быть распределены по разным серверам для обеспечения необходимой пропускной способности к таблице Blogs.

#### Множество маленьких секций

Предположим, канал потенциально может содержать 10 миллионов блогов с множеством активных обсуждений. В этом случае использование в качестве ключа для секционирования только имени канала может быть недостаточным, поскольку это обеспечивает создание одной очень большой секции с 10 миллионами блогов, для которой нельзя обеспечить балансировку нагрузки, обусловленной активными обсуждениями, путем распределения данных по разным серверам.

Одно из возможных решений, позволяющее уйти от таких монолитных секций, – использовать в качестве PartitionKey комбинацию имени канала и даты и времени создания блога. В этом случае PartitionKey будет уникально идентифицировать каждую сущность, и каждая секция таблицы будет содержать лишь одну сущность. Это замечательно для балансировки нагрузки, поскольку обеспечивает системе хранения абсолютную свободу в распределении сущностей по серверам соответственно нуждам приложения. В зависимости от распределения секций по серверам запросы типа «Get me all the blogs in the past day or past week» (Выбрать все блоги за последний день или неделю) могут потенциально стать довольно ресурсоемкими из-за необходимости выполнения доступа к нескольким узлам хранения.

#### Секция среднего размера

Альтернативным походом было бы сгруппировать блоги на основании имени канала и некоторого периода времени (недели/месяцы/дни), опираясь на популярность каналов. Если особенная активность в канале наблюдается в течение определенных месяцев, недель или дней, целесообразно секционировать блоги по времени создания. В этом случае PartitionKey должен включать и имя канала, и данные промежутка времени. Это обеспечивает более тонкое секционирование и большую гибкость системе для распределения нагрузки между множеством серверов для обеспечения нужд трафика. При этом запросы типа «Get me all the blogs in channel X in the past day» (Найти все блоги канала Х за прошедший день) будут более эффективными, поскольку направлены на небольшое целевое множество, определяемое промежутком времени, включенным в PartitionKey.

Возьмем для примера канал «Football». Периоды особенной активности в нем наблюдаются в течение сезона, т.е. с сентября по февраль. Для приложения будет лучше секционировать таблицу не только по имени канала, но также по месяцам или неделям. Использование «week» (неделя) в PartitionKey позволило бы сбалансировать нагрузку на такие популярные секции, как Суперкубок, и обеспечить тем самым высокую надежность и эффективность запросов.

Поскольку Windows Azure Table поддерживает только один PartitionKey, приложение микроблоггинга может использовать в качестве PartitionKey для данной таблицы <Имя\_Канала>\_<9999 - Год>\_<5 - Неделя#> или <Имя\_Канала>\_<9999 - Год>\_<13 - Месяц>. Выбор между <9999 - Год>\_<13 - Месяц> или <9999 - Год>\_<5 - Неделя#> зависит от того, что обеспечивает более сбалансированную нагрузку: группировка блогов по неделям или по месяцам. В этом примере компонент даты вычитается из верхней границы для обеспечения сортировки сущностей по дате публикации, начиная от самой недавней. Далее представлен пример класса сущности для данного подхода к секционированию:

[DataServiceKey("PartitionKey", "RowKey")]

public class Blog

{

// <Имя канала>\_<9999-YYYY>\_<13-Месяц>

public string PartitionKey { get; set; }

// Для сортировки по RowKey, начиная от самого недавнего, заканчивая

// опубликованным раньше всех блогом, RowKey определяем как

// DateTime.MaxValue.Ticks - DateTime.UtcNow.Ticks

public string RowKey { get; set; }

// Определяемые пользователем свойства

public string Text { get; set; }

public DateTime CreatedOn { get; set; }

public int Rating { get; set; }

}

Это позволяет выполнять запросы, как представленный ниже, и выводить на экран блоги, опубликованные в заданном году и месяце.

// для 10/2008 будем использовать <Имя канала>\_7991\_03. 7991 = 9999 – 2008; 03 = 13 - 10

var blogs =

from blog in context.CreateQuery<Blog>("Blogs")

where blog.PartitionKey == "Football\_7991\_03"

select blog;

foreach (Blog blog in blogs) { }

Если требуется выбрать все блоги года, может использоваться следующий запрос:

var blogs =

(from blog in context.CreateQuery<Blog>("Blogs")

where blog.PartitionKey.CompareTo("Football\_7991") <= 0

&& blog.PartitionKey.CompareTo("Football\_7990") > 0

select blog).Take(1000);

foreach (Blog blog in blogs) { }

ПРИМЕЧАНИЕ: Во всех фрагментах кода примера для краткости опущены маркеры продолжения. В реальном приложении для получения желаемых результатов может понадобиться выполнить несколько запросов с использованием маркеров продолжения.

### Динамическая настройка PartitionKey

В рассмотренном выше примере большую эффективность может обеспечить использование динамического секционирования. Можно было бы начать с секционирования по <Имя\_канала>\_<9999 - Год>\_<13 - Месяц>, но на время предполагаемых «горячих» месяцев сократить размеры секций за счет использования ключа <Имя\_канала>\_<9999 - Год>\_<13 - Месяц>\_<5 - Неделя>. Приложение должно знать о динамической схеме секционирования. Рассмотрим пример. Пусть секционирование блогов выполняется по следующим правилам:

1. В период с марта по июль, т.е. в «мертвый» сезон, используется ключ <Имя\_канала>\_<9999 - Год>\_<13 - Месяц>.
2. В остальные месяцы (т.е. во время сезона NFL) используется <Имя\_канала>\_<9999 - Год>\_<13 - Месяц>\_<5 - Неделя>

Теперь для извлечения всех блогов, опубликованных 7/3/2008 соответственно всемирному времени, приложение может использовать следующий запрос:

// long rowKeyStart = DateTime.MaxValue.Ticks - d1.Ticks where d1 = 7/3/2008 00:00:00

// long rowKeyEnd = DateTime.MaxValue.Ticks - d2.Ticks where d2 = 7/4/2008 00:00:00

var blogs =

(from blog in context.CreateQuery<Blog>("Blogs")

where blog.PartitionKey == "Football\_7991\_06"

&& blog.RowKey.CompareTo(rowKeyStart) >= 0 &&

&& blog.RowKey.CompareTo(rowKeyEnd) < 0

select blog).Take(1000);

foreach (Blog blog in blogs) { }

Однако для получения блогов, созданных 10/3/2008 соответственно всемирному времени, приложение может использовать следующий запрос:

// long rowKeyStart = DateTime.MaxValue.Ticks - d1.Ticks where d1 = 10/3/2008 00:00:00

// long rowKeyEnd = DateTime.MaxValue.Ticks - d2.Ticks where d2 = 10/4/2008 00:00:00

var blogs =

(from blog in context.CreateQuery<Blog>("Blogs")

where blog.PartitionKey == "Football\_7991\_03\_04"

&& blog.RowKey.CompareTo(rowKeyStart) >= 0 &&

&& blog.RowKey.CompareTo(rowKeyEnd) < 0

select blog).Take(1000);

foreach (Blog blog in blogs) { }

Данный запрос более эффективен при использовании «недели» в схеме секционирования, поскольку в этом случае область поиска сводится к блогам, созданным только в первую неделю месяца, в котором ожидается повышенная активность публикации блогов.

### Разные виды сущностей в одной таблице

Часто в приложениях используются данные, которые представляют разные сущности и должны извлекаться и обрабатываться вместе. Возьмем, к примеру, сайт социальной сети, на котором пользователь может вести блог, размещать фотографии и тому подобное. Обычно пользовательский интерфейс такой социальной сети группирует всю эту информацию для конкретного пользователя, так что при переходе на страницу Джо на ней можно будет увидеть последние изменения, сделанные Джо (т.е. самые недавние сообщения блога, фотографии, видео и т.д.). Также было установлено, что поскольку домашняя страница представляет самые недавние изменения, требуется оптимизировать запросы для обеспечения наиболее эффективного извлечения этих данных. Учитывая эти требования, можно обеспечить кластеризацию данных для эффективного извлечения, создав единственную таблицу, в которой будут храниться все сущности с указанием их вида («Kind»), сортированные по дате публикации, начиная от самых недавних. Данные могут быть секционированы по id пользователя, и дифференциация сущностей может выполняться по RowKey (т.е. <Время публикации (от самых недавних до самых старых)>\_<Вид сущности, например, фотография, сообщение блога, ролик и т.д.>).

Это также позволяет использовать транзакции над группами сущностей для создания/обновления/удаления всех связанных сущностей, принадлежащих одному пользователю, за одну транзакцию (раздел 4.9).

ПРИМЕЧАНИЕ: Как альтернатива, если основным сценарием является извлечение сущностей соответственно их виду, в качестве RowKey может использоваться <Вид сущности>\_<Время публикации (от самых недавних до самых старых)>. Но поскольку мы хотим оптимизировать доступ домашней страницы к последним изменениям, Время создания будет предшествовать Виду сущности в данном примере.

// Это объединение всех сущностей будет использоваться только

// для запросов. При вставке и обновлении сущностей используются

// отдельные классы для каждого вида сущностей.

[DataServiceKey("PartitionKey", "RowKey")]

public class SocialNetworkEntity

{

// Id пользователя

public string PartitionKey { get; set; }

// Для сортировки сущностей от самых недавних

// до самых старых определяем RowKey как

// <DateTime.MaxValue.Ticks - DateTime.UtcNow.Ticks>\_EntityType

// Длина может быть не более 19 разрядов

public string RowKey { get; set; }

// Определяемые пользователем свойства будут объединением

// всех свойств сущности.

// Тип сущности повторяется для облегчения поиска

public string EntityType { get; set; }

// Свойства блога

public string Text { get; set; }

public string Category { get; set; }

// Свойства фотографии

public string Uri { get; set; }

public string ThumbnailUri { get; set; }

// Общее для блога и фотографии

public string Caption { get; set; }

// Общее для блога и фотографии

public DateTime CreatedOn { get; set; }

public int Rating { get; set; }

// Данный класс может отвечать за возвращение соответствующего вида

// сущности, что избавляет от необходимости обработки объединения в

// бизнес-логике. Метод присвоения закрытый, что предотвращает его

// сохранение в хранилище.

public BlogEntity Blog { get; private set }

public PhotoEntity Photo { get; private set }

}

ПРИМЕЧАНИЕ: Класс с объединением всех свойств всех сущностей, сгруппированных в одной таблице, необходим для выполнения запросов только в ADO.Net Services. При использовании REST он не нужен, поскольку в этом случае мы имеем общий контроль над сериализацией и десериализацией. Это означает, что при наличии Kind в результате REST мы можем заполнить соответствующий тип полями данных, возвращенных запросом REST.

Теперь для извлечения сущностей, отображаемых на домашней странице, может использоваться следующий запрос:

var entities =

(from entity in

context.CreateQuery<SocialNetworkEntity>("SocialNetworkTable")

where entity.PartitionKey == userId

select entity).Take(100);

И следующий запрос обеспечит выбор только фотографий для отображения в фотоальбоме:

string startPhotoKey = string.Format("{0:D19}\_Photo", DateTime.MinValue.Ticks);

string endPhotoKey = string.Format("{0:D19}\_Photo", DateTime.MaxValue.Ticks);

var entities =

(from entity in

context.CreateQuery<SocialNetworkEntity>("SocialNetworkTable")

where entity.PartitionKey == userId

and RowKey.CompareTo(startPhotoKey) >= 0

and RowKey.CompareTo(endPhotoKey) <= 0

select entity).Take(100);

ПРИМЕЧАНИЕ: При создании RowKey обеспечиваем фиксированную длину префикса. Применяем 19 разрядов, что позволяет использовать весь диапазон длинных целых чисел.

Если целью приложения является извлечение сущностей по их видам (Kind), вид должен идти перед временем создания в RowKey. Но в данном примере основной задачей является извлечение и отображение на веб-странице последних N изменений, так что выбранный RowKey оптимизирован для этого.

#### Десериализация сущностей различных типов

Также существует возможность десериализации сущностей различных типов с помощью события Astoria ReadingEntity. В данном примере предположим, что в одной таблице хранятся блоги и комментарии. PartitionKey таблицы – «имя пользователя», и RowKey – <Ключ строки блока>\_<CommentId>

/// <summary>

/// Вспомогательный класс, используемый для хранения данных,

/// полученных в результате синтаксического разбора полезной нагрузки ответа.

/// </summary>

class GenericType

{

public string ValueType { get; set; }

public string Value { get; set; }

public string PropertyName { get; set; }

public bool IsNull { get; set; }

}

/// <summary>

/// Реализуем обработчик чтения сущности так, чтобы он читал

/// сущности и сохранял все блоги в словаре. Каждый блог включает все связанные комментарии.

/// Этот класс будет воссоздавать сущности из полезной нагрузки и сохранять их

/// для последующей работы с ними.

/// </summary>

class BlogReader

{

private static readonly XNamespace AtomNamespace = "http://www.w3.org/2005/Atom";

private static readonly XNamespace AstoriaDataNamespace =

"http://schemas.microsoft.com/ado/2007/08/dataservices";

private static readonly XNamespace AstoriaMetadataNamespace =

"http://schemas.microsoft.com/ado/2007/08/dataservices/metadata";

internal Dictionary<string, Blogs> Blogs = new Dictionary<string, Blogs>();

internal BlogReader(DataServiceContext context)

{

context.ReadingEntity +=

new EventHandler<ReadingWritingEntityEventArgs>(OnReadingEntity);

context.ResolveType = this.ResolveType;

}

public Type ResolveType(string name)

{

return typeof(BaseEntity);

}

public void OnReadingEntity(object sender, ReadingWritingEntityEventArgs e)

{

// Читаем все свойства из полузной нагрузки для этой сущности и сохраняем их,

// используя экземпляры GenericType.

var q = from p in e.Data.Element(AtomNamespace + "content")

.Element(AstoriaMetadataNamespace + "properties")

.Elements()

select new GenericType()

{

PropertyName = p.Name.LocalName,

IsNull = string.Equals("true",

p.Attribute(AstoriaMetadataNamespace + "null") == null ? null :

p.Attribute(AstoriaMetadataNamespace + "null").Value,

StringComparison.OrdinalIgnoreCase),

ValueType =

p.Attribute(AstoriaMetadataNamespace + "type") == null ? null : p.Attribute(AstoriaMetadataNamespace + "type").Value,

Value = p.Value

};

// Тип сущности сохранен в свойство "EntityType".

// Его необходимо считать для создания соответствующих типов сущностей и их сохранения.  
 string type = (from s in q.AsQueryable()

where s.PropertyName == "EntityType"

select s.Value.ToString()).FirstOrDefault();

if (string.Equals(type, "Blogs"))

{

Blogs blog = ReadBlog(e.Entity as BaseEntity, q);

this.Blogs.Add(blog.RowKey, blog);

}

else

{

// Поскольку комментарий сохраняет в собственном ключе id блога, необходимо

// выполнить его синтаксический разбор, чтобы воссоздать отношение.

// Прочитанный комментарий сохраняется вместе с блогом, который должен быть уже

// прочитан ранее, поскольку ключ блога < ключа комментария.

Comments comment = ReadComment(e.Entity as BaseEntity, q);

string[] tokens = comment.RowKey.Split('\_');

this.Blogs[tokens[0]].Comments.Add(comment);

}

}

private Blogs ReadBlog(BaseEntity entity, IEnumerable<GenericType> blogProperties)

{

Blogs blog = new Blogs();

blog.PartitionKey = entity.PartitionKey;

blog.RowKey = entity.RowKey;

blog.Timestamp = entity.Timestamp;

// Чтобы извлечь интересующие нас свойства, либо перебираем все свойства,

// либо используем Linq.

foreach (GenericType t in blogProperties)

{

if (string.Equals(t.PropertyName, "Message"))

{

blog.Message = GetTypedEdmValue(t.ValueType, t.Value, t.IsNull).ToString();

}

}

return blog;

}

private Comments ReadComment(BaseEntity entity, IEnumerable<GenericType> properties)

{

Comments comment = new Comments();

comment.PartitionKey = entity.PartitionKey;

comment.RowKey = entity.RowKey;

comment.Timestamp = entity.Timestamp;

// Чтобы извлечь интересующие нас свойства, либо перебираем все свойства,

// либо используем Linq.

foreach (GenericType t in properties)

{

if (string.Equals(t.PropertyName, "Comment"))

{

comment.Comment = GetTypedEdmValue(t.ValueType, t.Value, t.IsNull).ToString();

}

else if (string.Equals(t.PropertyName, "Rating"))

{

comment.Rating = (int)GetTypedEdmValue(t.ValueType, t.Value, t.IsNull);

}

}

return comment;

}

private static object GetTypedEdmValue(string type, string value, bool isnull)

{

// Создаем значение в зависимости от типа

if (isnull) return null;

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

switch (type)

{

case "Edm.String": return value;

case "Edm.Byte": return Convert.ChangeType(value, typeof(byte));

case "Edm.SByte": return Convert.ChangeType(value, typeof(sbyte));

case "Edm.Int16": return Convert.ChangeType(value, typeof(short));

case "Edm.Int32": return Convert.ChangeType(value, typeof(int));

case "Edm.Int64": return Convert.ChangeType(value, typeof(long));

case "Edm.Double": return Convert.ChangeType(value, typeof(double));

case "Edm.Single": return Convert.ChangeType(value, typeof(float));

case "Edm.Boolean": return Convert.ChangeType(value, typeof(bool));

case "Edm.Decimal": return Convert.ChangeType(value, typeof(decimal));

case "Edm.DateTime": return XmlConvert.ToDateTime(value, XmlDateTimeSerializationMode.RoundtripKind);

case "Edm.Binary": return Convert.FromBase64String(value);

case "Edm.Guid": return new Guid(value);

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

}

}

}

Имея приведенные выше классы, можно читать сущности разных типов, используя следующий код:

DataServiceQuery<BaseEntity> query = ((

from a in context.CreateQuery<BaseEntity>(DataContext.TableName)

where a.PartitionKey == user

select a)) as DataServiceQuery<BaseEntity>;

// выполняем запрос так, чтобы иниццировать

// событие OnReadingEntity BlogReader

BlogReader reader = new BlogReader(context);

query.Execute().ToList();

// выполняем доступ к блогам, хранящимся в BlogReader

foreach (Blogs blog in reader.Blogs.Values)

{

Console.WriteLine("{0}", blog);

foreach (Comments comment in blog.Comments)

{

Console.WriteLine("\t{0}", comment);

}

}

## Обновление и управление версиями

Каждое приложение сталкивается с необходимостью обновления текущей схемы для удовлетворения новым требованиям.

Важно отметить, что Windows Azure Table не хранит схему, хранятся только пары <имя, типизированное значение> для каждого свойства сущности. Поэтому в одной таблице могут присутствовать две сущности с разными свойствами. Также свойства двух сущностей таблицы могут иметь одно имя, но разные типы. Например, может быть сущность А со свойством «Rating» типа Int и сущность B со свойством «Rating» типа Double. При выполнении запроса сервер игнорирует сущность, если тип свойства в условии запроса не соответствует типу свойства данной сущности. В рассматриваемом выше примере по запросу для поиска «Rating > 1.2» будет выбрана только сущность В, сущность А не войдет в результат из-за несоответствия типа свойства.

При обновлении свойства важно, чтобы приложение обслуживало обе его версии до тех пор, пока не будут обновлены все имеющиеся в хранилище данные. Далее в данном разделе обсуждается практическая польза от сохранения версий свойств для каждой сущности, что позволяет приложениям обрабатывать разные виды обновлений, и использование ADO.NET Service для выполнения обновления.

Во всех описываемых ниже сценариях обновления имеется две возможности применения обновления:

1. Обновление при доступе, т.е. обновление сущности до более новой версии происходит при каждом ее извлечении для обновления.
2. Обновление в фоновом режиме, т.е. обновление выполняется в фоновом режиме с использованием рабочей роли. Для этого рабочая роль извлекает все сущности, относящиеся к старой версии (посредством столбца версии), и обновляет их.

Как говорилось ранее, для операций чтения приложение должно обрабатывать обе версии. Прежде чем углубиться в детали добавления и удаления свойств, рассмотрим важное булево свойство DataServiceContext IgnoreMissingProperties (Игнорировать недостающие свойства). Значение IgnoreMissingProperties определяет, будет ли ADO.NET Data Service формировать исключение, если класс сущности на стороне клиента не описывает свойство, возвращенное сервером. Например, если для сущности Blog сервер возвращает свойство Rating, но свойство Rating не определено в клиентском описании класса для Blog, ADO.NET Data Service формирует исключение, только если IgnoreMissingProperties имеет значение false.

// Задаем true, чтобы сделать допустимым отсутствие некоторых свойств в

// клиентском описании сущности, что может возникнуть в результате

// добавления/удаления свойств в сущностях, хранящихся на сервере.

context.IgnoreMissingProperties = true;

### Добавление нового свойства

Если приложение добавляет новое свойство, оно должно быть определено и в описании сущности.

Поскольку также могут выполняться и предыдущие версии приложения, рекомендуется всегда задавать свойству DataServiceContext IgnoreMissingProperties значение true, как показано выше. Это обеспечивает возможность чтения сущностей клиентам старых версий, в которых новых свойств еще нет. В противном случае, ADO.NET Data Service будет формировать исключение для каждого свойства в возвращенном пакете REST, не описаного классом сущности.

### Удаление типа свойства

Процедуру удаления свойства рассмотрим с помощью примера удаления свойства Rating сущностей блога.

1. Создадим новое описание сущности Blog, не включающее свойство Rating.

[DataServiceKey("PartitionKey", "RowKey")]

public class BlogV2

{

// <Имя канала>\_<9999-YYYY>\_<13-Месяц>

public string PartitionKey { get; set; }

// Чтобы сортировать по RowKey от самых недавних до

// самых старых блогов, определяем RowKey как

// DateTime.MaxValue.Ticks - DateTime.UtcNow.Ticks

public string RowKey { get; set; }

// Определяемые пользователем свойства

public string Text { get; set; }

public DateTime CreatedOn { get; set; }

// ПРИМЕЧАНИЕ: удаляем это свойство так, чтобы оно

// также могло быть удалено из хранилища.

// public int Rating { get; set; }

}

1. Теперь, чтобы удалить свойство, будем извлекать сущности блога с сервера и обновлять их, используя новое описание сущности:

// Обеспечиваем, что исключение не будет формироваться в случае

// отсутствия описания Rating в BlogV2 (как обсуждалось в разделе 8.4)

context.IgnoreMissingProperties = true;

// Извлекаем сущности блога в BlogV2.

// ПРИМЕЧАНИЕ: Маркеры продолжения опущены для краткости

var blogs =

(from blog in context.CreateQuery<BlogV2>("Blogs")

select blog);

foreach(BlogV2 blog in blogs)

{

context.UpdateObject(blog);

}

1. Вызываем SaveChanges с SaveChangesOptions.ReplaceOnUpdate, чтобы удалить на сервере все недостающие свойства.

// В результате вызова SaveChangesOptions.ReplaceOnUpdate на сервер

// отправляется запрос "PUT", а не

// "MERGE", что обеспечит удаление всех недостающих свойств.

context.SaveChanges(SaveChangesOptions.ReplaceOnUpdate);

### Изменение типа свойства

Когда схема требует изменения типа свойства, одним из вариантов является создание нового свойства заданного типа с новым именем, затем копирование данных из старого свойства в новое и, наконец, удаление старого свойства. Для реализации этих задач используются приемы, описанные выше в разделах 8.4.1 и 8.4.2.

# Лучшие практики Windows Azure Table

В данном разделе будут рассмотрены различные рекомендуемые лучшие практики. Применение этих лучших практик поможет сделать приложение надежным, масштабируемым и гибким.

## Создание таблицы

В приложениях логика создания таблицы должна быть отделена от основной бизнес-логики. Например, возьмем приложения, в которых таблица создается перед каждой вставкой. В этом случае в качестве индикатора, чтобы избежать повторного создания таблицы, используются исключения «Table already exists» (Таблица уже существует). Таким образом, получаем большое количество лишних обращений к хранилищу, что в итоге приводит к ненужным транзакциям и, следовательно, дополнительным расходам.

Рекомендуется выполнять логику создания таблицы как отдельный сценарий при первой настройке сервиса, при запуске роли, или когда ошибка «TableNotFound» (Таблица не найдена) возвращается с HTTP-кодом состояния 404. Логика создания таблицы не должна строиться на исключении «Table already exists», перед созданием таблицы должна выполняться попытка извлечения этой таблицы.

Заметьте, что после удаления таблицы, таблица с таким же именем не может быть создана еще в течение примерно 30 секунд, пока не будет выполнена сборка мусора.

## Асинхронная версия ADO.NET Data Services API

Асинхронные API ввода/вывода позволяют приложениям более продуктивно использовать ЦП, чем синхронные, которые блокируют выполнение приложения в ожидании ответа с сервера. ADO.NET Data Services обеспечивают асинхронные версии операций CRUD. Для повышения производительности рекомендуется максимально широко использовать асинхронные версии. Например, для выполнения запроса лучше использовать метод BeginExecute DataServiceContext.

## Настройки DataServiceContext

* При сбое операции обновления/удаления/добавления изменение все равно отслеживается и последующий SaveChanges повторит соответствующий запрос. Рекомендуется использовать либо новый DataServiceContext, либо явно отсоединить сущность, для которой произошел сбой.
* Хорошей практикой является всегда задавать IgnoreMissingProperties равным true для контекста данных, что обеспечивает лучшую обратную совместимость.

## Схема секционирования

Рекомендуется выбирать схему секционирования, исходя из производительности, масштабируемости и возможности расширения в будущем. Более подробно эти вопросы рассматриваются в разделах 3 и 8.3.

## Безусловные обновления и удаления

Теги ETag можно рассматривать как версии сущностей. Они могут использовать для контроля взаимосовместимости с помощью заголовка If-Match при обновлениях/удалениях. Astoria сохраняет этот etag, который передается с полезной нагрузкой каждой сущности. Для получения более детальных сведений Astoria отслеживает сущности в контексте с помощью context.Entities, который является коллекцией EntityDescriptors (Дескрипторы сущностей). EntityDescriptor имеет свойство «Etag», значение которого сохраняет Astoria. При каждом обновлении/удалении ETag передается на сервер. Astoria по умолчанию отправляет обязательный заголовок «If-Match» со значением этого etag. На стороне сервера Windows Azure Table проверяет, соответствует ли присланный в заголовке «If-Match» etag значению свойства Timestamp в хранилище данных. Если значения совпадают, сервер продолжает обработку и выполняет обновление/удаление; в противном случае, сервер возвращает код состояния 412, т.е. «Precondition failed» (Необходимое условие не выполнено), свидетельствующее о том, что рассматриваемая сущность, возможно, была изменена. Переданная клиентом «\*» в заголовке «If-Match» указывает серверу на то, что должно быть выполнено безусловное обновление/удаление, т.е. операция должна быть осуществлена независимо от того, была изменена сущность в хранилище или нет. Клиент может выполнить запрос на безусловное обновление/удаление следующим образом:

context.AttachTo("TableName", entity, "\*");

context.UpdateObject(entity);

Однако если данная сущность уже отслеживается, клиенту потребуется отсоединить ее перед присоединением:

context.Detach(entity);

## Обработка ошибок

При проектировании приложений для работы с Windows Azure Table важно обеспечить соответствующую обработку ошибок. В данном разделе рассматриваются вопросы, которые должны быть учтены при разработке приложения.

### Успешное завершение операций на стороне сервера, несмотря на ошибки сети и истечение времени ожидания сети

Даже если при выполнении некоторой операции возникла ошибка сети, внутренняя ошибка сервера или истекло время ожидания, операция все равно могла успешно завершиться на сервере. В этом случае рекомендуется выполнить операцию повторно и быть готовым к получению ошибки, обусловленной тем, что предыдущий запрос был завершен. Например, если возник сбой при удалении, попытайтесь удалить сущность еще раз и ожидайте ошибки «не найден», если предыдущая операция удаления все-таки была завершена успешно. Другой пример – добавление сущности. Даже если первый запрос завершается в результате превышения времени ожидания, сущность может быть вставлена в таблицу. В этом случае приложение должно уметь обрабатывать ошибку «entity already exists» (сущность уже существует), которую может получить при повторном выполнении этой операции.

### Повторные запросы и ошибки «Connection closed by Host»

Запросы, завершившиеся ошибками превышения времени ожидания или «Connection closed by Host» (Подключение разорвано хостом), возможно, не были обработаны Windows Azure Table. Например, если запрос PUT возвращается в результате превышения времени ожидания, последующий запрос GET может возвратить либо старое, либо обновленное значение. При получении многократных ошибок превышения времени ожидания необходимо выполнить запрос повторно после задержки произвольной длительности во избежание дальнейшей перегрузки системы.

### Конфликты при обновлениях

Как описывалось в разделе 5, приложение должно обрабатывать ошибку «Precondition Failed» путем извлечения последней версии сущности и формирования последующего обновления.

### Настройка приложения для обработки повторяющихся ошибок превышения времени ожидания

Ошибки превышения времени ожидания могут происходить из-за ошибок сети между приложением и центром обработки данных. Для глобальных сетей рекомендуется разбивать одну большую операцию на ряд меньших вызовов и включать в приложения обработку превышений времени ожидания/сбоев, чтобы можно было возобновлять выполнение после возникновения такой ошибки. Настройка размеров каждого запроса выполняется, исходя из имеющегося в распоряжении сетевого соединения.

Система спроектирована с обеспечением возможности масштабироваться и обрабатывать большие объемы трафика. Однако при особенно высокой частоте запросов из-за балансировки нагрузки системой запросы могут завешаться ошибкой превышения времени ожидания. Сократить или вообще устранить ошибки такого рода поможет снижение частоты запросов. Вообще говоря, пользователи редко сталкиваются с этим, однако при возникновении частых или неожиданных ошибок превышения времени ожидания обратитесь к нам на форумы MSDN, где мы обсудим, как можно оптимизировать работу с Windows Azure Table и предотвратить эти ошибки.

### Обработка ошибок и составление отчетов

API REST выглядит как стандартный HTTP-сервер, взаимодействующий с существующими HTTP-клиентами (например, браузерами, клиентскими библиотеками HTTP, прокси, кэшами и т.д.). Чтобы гарантировать соответствующую обработку ошибок HTTP-клиентами, каждой ошибке Windows Azure Table поставлен в соответствие определенный HTTP-код состояния.

HTTP-коды состояния менее выразительные, чем коды ошибок Windows Azure Table, и содержат меньше информации об ошибке, тем не менее клиенты, понимающие HTTP, обычно обрабатывают ошибки правильно.

Поэтому при обработке ошибок или в сообщениях об ошибках Windows Azure Table для конечных пользователей следует использовать коды ошибок Windows Azure Table вместе с HTTP-кодом состояния, это обеспечит максимум информации об ошибке. Кроме того, при отладке приложения необходимо также обращать внимание на предназначенный для пользователя элемент <ExceptionDetails> XML-сообщения об ошибке.

Каждое сообщение об ошибке Windows Azure Table включает следующие HTTP-заголовки.

HTTP/1.1 204 No Content

Content-Length: 0

ETag: W/"datetime'2008-10-01T15%3A27%3A34.4838174Z'"

x-ms-request-id: 7c1b5e22-831d-403c-b88a-caa4443e75cb

Если у вас возникли подозрения по поводу обработки вашего запроса на сервере, можете обратиться в раздел Windows Azure Storage на форумах MSDN, используя приведенный выше x-ms-request-id. Это поможет нам диагностировать проблему.

## Настройка производительности .NET и ADO.NET

Здесь мы собрали самые распространенные проблемы, связанные с .NET и ADO.NET Data Services (также называемые Astoria), с которыми сталкиваются пользователи при работе с Windows Azure Table, и привели возможные их решения.

### Улучшение производительности десериализации при использовании сервисов данных ADO.NET.

В запросе, выполняемом с использованием сервисов данных ADO .Net, присутствуют два важных имени: имя CLR-класса сущности и имя таблицы в Windows Azure Table. Мы заметили, что если эти имена отличаются, на десериализацию каждой сущности, полученной в запросе, уходит примерно 8-15 мс. Подробнее этот вопрос рассматривается в разделе 4.10.

### Изменения значения по умолчанию для HTTP-соединений .NET

По умолчанию ряд HTTP-соединений .NET имеют значение 2, что подразумевает возможность обслуживания только 2 одновременных подключений. Это печально известная проблема, которая доставила немало хлопот многим разработчикам. Обнаруживается она появлением сообщения «underlying connection was closed...» (Используемое соединение закрыто) при установлении более двух одновременных соединений. Поведение по умолчанию можно изменить, задавая следующее в конфигурационном файле приложения ИЛИ в коде:

**Конфигурационный файл:**

<system.net>

<connectionManagement>

<add address = "\*" maxconnection = "48" />

</connectionManagement>

</system.net>

|  |
| --- |
| **Код:**  ServicePointManager.DefaultConnectionLimit = 48; |

Точное значение зависит от конкретного приложения. На сайте <http://support.microsoft.com/kb/821268> очень хорошо описывается, как это делается для приложений на стороне сервера. Также это значение может быть задано для конкретного URI, если вместо «\*» указать этот URI. При задании количества соединений через код, вместо класса ServicePointManager можно использовать класс ServicePoint, например:

ServicePoint myServicePoint = ServicePointManager.FindServicePoint(myServiceUri);

myServicePoint.ConnectionLimit = 48;

### Отключение «100-continue»

Что такое «100-continue»? При отправке запроса POST/PUT клиент может отложить отправку полезной нагрузки, передав заголовок «Expect: 100-continue».

1. Сервер будет использовать URI плюс заголовки, чтобы гарантировать возможность вызова.  
2. Сервер возвратит клиенту ответ с кодом состояния 100 (Continue).  
3. Клиент отправит остальные данные.

Это обеспечивает уведомление клиента о большинстве ошибок без лишних затрат. Однако ошибки могут возникать и при получении полезной нагрузки на стороне сервера. При использовании библиотеки .NET HttpWebRequest по умолчанию отправляет «Expect: 100-Continue» для всех запросов PUT/POST (несмотря на то, что MSDN предлагает делать это только для запросов POST).

В Windows Azure Tables/Blobs/Queue ошибки аутентификации и ошибки, возникающие в результате использования неподдерживаемых команд и пропущенных заголовков, могут быть протестированы всего лишь посредством получения заголовков и URI. Если клиентское приложение протестировано достаточно тщательно и гарантированно не будет отправлять каких-либо плохих запросов, клиенты Windows Azure могут отключить заголовок «100-continue», чтобы обеспечить отправку сразу всего запроса. Это особенно полезно, если клиенты передают небольшое количество данных в таблицы или сервис очереди. Эта настройка может быть отключена в коде или в конфигурационном файле.

|  |
| --- |
| **Код:**  // set it on service point if only a particular service needs to be disabled.  ServicePointManager.Expect100Continue = false; |

**Конфигурационный файл:**

<system.net>

<settings>

<servicePointManager expect100Continue="false" />

</settings>

</system.net>

Прежде чем отключать заголовок «100-continue», рекомендуем протестировать, как будет работать приложение с и без этого заголовка.

### Отключение Nagle может ускорить операции вставки/обновления

Замечено, что отключение Nagle обеспечивает существенное уменьшение задержки для операций вставки и обновления в таблице. Обычно это происходит при наличии большого числа небольших сущностей. Однако известно, что отключение Nagle неблагоприятно сказывается на пропускной способности и, следовательно, требуется тщательно протестировать приложение, чтобы понять, имеет ли такое отключение смысл.

Эта настройка может быть отключена в коде или в конфигурационном файле, как показано ниже.

|  |
| --- |
| **Код:**  ServicePointManager.UseNagleAlgorithm = false; |
|  |
|  |

**Конфигурационный файл:**

<system.net>

<settings>

<servicePointManager expect100Continue="false" useNagleAlgorithm="false"/>

</settings>

</system.net>

## Повторное использование имени удаленной таблицы

На этапе разработки регулярно может возникать необходимость в удалении таблиц. На данный момент удаленная таблица может быть воссоздана не ранее, чем через 40 секунд. При разработке приложения может быть полезным использовать в именах таблиц id или номер версии в качестве суффикса. Это избавило бы от вынужденного ожидания при воссоздании таблицы после ее удаления. Однако при запросах к таблице, имя которой не совпадает с именем класса сущности, в клиентской библиотеке ADO.NET Data Service возникают издержки производительности, поэтому в таких случаях должен быть задан ResolveType.

# Заключение

Windows Azure Table предоставляет структурированную платформу хранения. Она поддерживает высокомасштабируемые таблицы в облаке. Система эффективно масштабирует таблицы, автоматически распределяя секции по разным серверам и обеспечивая сбалансированную нагрузку по мере увеличения трафика. Она поддерживает широкий набор типов данных для свойств, доступ к ней может быть осуществлен средствами ADO.NET Data Services и REST. Контроль типов во время компиляции обеспечивается через ADO.NET Data Services, что позволяет работать с Windows Azure Table, как с любой другой структурированной таблицей. Это подразумевает поддержку таких функций как разбиение на страницы, нежесткая блокировка и маркеры продолжения для запросов, выполнение которых требует длительного времени. Текущие ограничения по свойствам и времени выполнения запросов позволяют нам обеспечивать высокомасштабируемую систему хранения таблиц.