Windows Azure Table

GVHD: Phan Trung Hiếu Sinh viên thực hiện:

Nguyễn Minh Tuấn – 07520450

Nguyễn Anh Duy - 07520058

Table of Contents

[1 Giới thiệu 2](#_Toc290114467)

[2 Mô hình dữ liệu bảng 3](#_Toc290114468)

[3 Phân vùng bảng 6](#_Toc290114469)

[3.1 Tác động của việc phân vùng 6](#_Toc290114470)

[3.1.1 Khả năng mở rộng của bảng 6](#_Toc290114471)

[3.1.2 Giao dịch nhóm thực thể 7](#_Toc290114472)

[3.1.3 Miền thực thể 7](#_Toc290114473)

[3.2 Chọn một PartitionKey 7](#_Toc290114474)

[3.2.1 Giao dịch nhóm thực thể 7](#_Toc290114475)

[3.2.2 Truy vấn hiệu quả 8](#_Toc290114476)

[3.2.3 Khả năng mở rộng 8](#_Toc290114477)

[4 Lập trình Bảng 9](#_Toc290114478)

[4.1 Đặt tên phiên bản 10](#_Toc290114479)

[4.2 Chạy thử ví dụ 11](#_Toc290114480)

[4.3 Định nghĩa lớp thực thể cho bảng 12](#_Toc290114481)

[4.4 Tạo bảng 12](#_Toc290114482)

[4.5 Thêm 1 Blog 13](#_Toc290114483)

[4.6 Truy vấn Blog 14](#_Toc290114484)

[4.7 Cập nhật một Blog 14](#_Toc290114485)

[4.8 Xóa một Blog 15](#_Toc290114486)

[4.9 Giao dịch nhóm thực thể 15](#_Toc290114487)

[4.9.1 Lỗi 17](#_Toc290114488)

[4.10 Cách làm tốt nhất khi sử dụng DataServiceContext 20](#_Toc290114489)

[4.11 Sử dụng API REST 21](#_Toc290114490)

[5 Cập nhật đồng thời 21](#_Toc290114491)

[5.1 Cập nhật không điều kiện 23](#_Toc290114492)

[6 Phân trang các kết quả truy vấn 23](#_Toc290114493)

[6.1 Lấy N thực thể trên cùng 24](#_Toc290114494)

[6.2 Thẻ tiếp nối 24](#_Toc290114495)

[7 Mô hình nhất quán (Consistency Model) 25](#_Toc290114496)

[7.1 Tính nhất quán bảng đơn 25](#_Toc290114497)

[7.2 Tính nhất quán Cross-Tab 25](#_Toc290114498)

[8 Mẹo và thủ thuật 27](#_Toc290114499)

[8.1 Lấy những đối tượng mới nhất (Mô phỏng thứ tự giảm dần) 27](#_Toc290114500)

[8.2 Lấy ra bằng cách sử dụng tiền tố 28](#_Toc290114501)

[8.3 Ví dụ phân vùng dữ liệu 28](#_Toc290114502)

[8.3.1 Tình huống nghiên cứu Micro Blogging 29](#_Toc290114503)

[8.3.2 Dynamically selecting the granularity of the PartitionKey 32](#_Toc290114504)

[8.3.3 Những kiểu thực thể khác nhau trong cùng một bảng 33](#_Toc290114505)

[8.4 Cập nhật và đặt tên phiên bản 36](#_Toc290114506)

[8.4.1 Thêm thuộc tính mới 37](#_Toc290114507)

[8.4.2 Xóa kiểu thuộc tính 37](#_Toc290114508)

[8.4.3 Thay đổi kiểu thuộc tính 38](#_Toc290114509)

[9 Tóm lược 38](#_Toc290114510)

# Giới thiệu

Windows Azure là nền tảng đám mây của Microsoft. Nó được gọi là “hệ điều hành đám mây”, nó cung cấp những khối kiến trúc cần thiết để các lập trình viên mở rộng và nâng cấp những dịch vụ sẵn có. Windows Azure cung cấp:

* Tính toán ảo hóa.
* Lưu trữ mở rộng.
* Quản lí tự động.
* Bộ phát triển SDK phong phú.

Bộ lưu trữ Windows Azure cho phép các lập trình viên lưu dữ liệu của họ trên đám mây. Ứng dụng có thể truy xuất dữ liệu từ bất kì nơi đâu, tại bất kì thời điểm nào, lưu trữ lượng dữ liệu tùy ý trong khoảng thời gian bất kì và nó đảm bảo rằng dữ liệu được lưu trữ bền vững và không bị thất lạc. Bộ lưu trữ Windows Azure cung cấp một bộ trừu tượng hóa dữ liệu phong phú:

* Windows Azure Blob – cung cấp lưu trữ cho các mẫu dữ liệu lớn.
* Windows Azure Table – cung cấp bộ lưu trữ có cấu trúc để lưu trữ tình trạng sử dụng.
* Windows Azure Queue – cung cấp điều phối công việc không đồng bộ để khởi động dịch vụ liên lạc.

Windows Azure Table là bộ lưu trữ có cấu trúc được cung cấp bởi nền tảng Windows Azure. Nó hỗ trợ các bảng có khả năng mở rộng cực lớn trên đám mây, nó có thể chứa hàng tỉ thực thể và terabyte dữ liệu. Hệ thống sẽ tự động mở rộng một cách hiệu quả đến hàng ngàn máy chủ.

Bộ lưu trữ có cấu trúc được cung cấp dưới dạng bảng, mỗi bảng chứa một tập các thực thể, mỗi thực thể chứ một tập các thuộc tính được đặt tên. Một số điểm nổi bật của Windows Azure Table :

* Hỗ trợ LINQ, dịch vụ dữ liệu ADO .NET và REST.
* Tập hợp các kiểu dữ liệu phong phú cho các giá trị thuộc tính.
* Hỗ trợ số lượng bảng và thực thể không giới hạn và không bị giới hạn về kích thước bảng.
* Dùng mô hình optimistic concurrency cho những thao tác cập nhật và xóa.
* Đối với các truy vấn lấy ra số lượng lớn các kết quả hoặc truy vấn bị ngắt quãng. Từng phần kết quả được trả về với một cái thẻ nối tiếp cho phép truy vấn lại tiếp tục tại nơi mà nó đã dừng lại.

# Mô hình dữ liệu bảng

Sau đây tóm tắt các mô hình dữ liệu cho Windows Azure Table:

* **Storage Account (Tài khoản lưu trữ) -** Ứng dụng phải sử dụng một tài khoản hợp lệ để truy xuất bộ lưu trữ Windows Azure. Bạn có thể tạo một tài khoản mới qua trang web của Windows Azure. Người dùng sẽ nhận được một khóa bí mật 256-bit cho mỗi tài khoản được tạo. Khóa bí mật này dùng để chứng thực những truy vấn từ người dùng đến hệ thống lưu trữ. Đặc biệt, một chữ kí HMAC SHA256 cho mỗi request sử dụng khóa bí mật này. Chữ ký được chuyển qua với mỗi request để xác thực các request của người dùng.

Tên tài khoản là một phần của hostname trên đường dẫn URL. Hostname để truy xuất các bảng là <accountName>.table.core.windows.net.

* **Table (Bảng)** – chứa một bộ các thực thể. Một ứng dụng có thể tạo ra nhiều bảng chỉ với một tài khoản lưu trữ.
* **Entity (Thực thể)** – Thực thể (một thực thể tương tự như một dòng) là mẫu dữ liệu cơ bản được lưu trữ trong một bảng. Một thực thể chứa một tập các thuộc tính. Mỗi bảng có 2 thuộc tính tên là “PartitionKey” và “RowKey” tạo ra khóa duy nhất cho thực thể.
* **Property (Thuộc tính)** – Nó đại diện cho một giá trị duy nhất trong thực thể. Tên thuộc tính phân biệt hoa, thường. Một tập hợp phong phú các kiểu dữ liệu được hỗ trợ cho giá trị thuộc tính.
* **PartitionKey** – Đây là thuộc tính khóa đầu tiên của mỗi bảng. Hệ thống sử dụng khóa này để tự động cấp phát các thực thể của bảng qua nhiều nút lưu trữ.
* **RowKey** – là thuộc tính khóa thứ hai của bảng. Đây là ID duy nhất của thực thể trong phân vùng mà nó thuộc về. PartitionKey kết hợp với RowKey xác định duy nhất một thực thể trong một bảng.
* **Timestamp** – là thuộc tính có giá trị **Datetime**, được lưu giữ bởi hệ thống để ghi lại thời điểm thực thể thay đổi cuối cùng.
* **Partition** – Một tập hợp các thực thể trong một bảng với cùng giá trị khóa PartitionKey.
* **Sort Order (Thứ tự sắp xếp)** – Các thực thể trong bảng được sắp xếp theo PartitionKey vào sau đó là theo RowKey. Điều này có nghĩa là các truy vấn xác định theo các khóa này sẽ hiệu quả hơn, và tất cả các kết quả được trả về được sắp xếp theo PartitionKey và sau đó theo RowKey.

Một bảng có một giản đồ linh hoạt. Windows Azure Table lưu giữ tên và kiểu dữ liệu cho mỗi thuộc tính trong một thực thể.

Sau đâu là một số thông tin chi tiết bổ sung cho Bảng, Thực thể và Thuộc tính :

* Bảng
  + Tên bảng chỉ chứa các kí tự và số.
  + Tên bảng không bắt đầu với một kí tự số.
  + Tên bảng phân biệt hoa – thường.
  + Tên bảng phải có độ dài từ 3 đến 63 kí tự.
* Tên thuộc tính

Chỉ được chứa các kí tự chữ cái và dấu ‘\_’.

* Một thực thể có thể có tối đa 255 thuộc tính bao gồm các thuộc tính bắt buộc - PartitionKey, RowKey và Timestamp. Tất cả các thuộc tính khác trong thực thể có tên được định nghĩa theo nhu cầu ứng dụng.
* PartitionKey và RowKey có kiểu chuỗi, và mỗi khóa bị giới hạn trong kích thước 1 KB.
* Timestamp là thuộc tính chỉ đọc.
* Không có giản đồ cố định – Không có giản đồ được lưu trữ bởi Windows Azure Table, vì thế tất cả các thuộc tính được lưu trữ theo cặp <tên, kiểu dữ liệu>. Điều này có nghĩa rằng 2 thực thể trong cùng một bảng có thể có thuộc tính khác nhau. Một bảng có thể có hai thực thể với tên thuộc tính giống nhau, nhưng có kiểu dữ liệu khác nhau cho giá trị thuộc tính. Tuy nhiên, tên thuộc tính phải là duy nhất với mỗi thực thể.

• Kết hợp kích thước của tất cả các dữ liệu trong một thực thể không thể vượt quá 1MB. Kích thước này bao gồm kích thước của tên thuộc tính cũng như kích thước của giá trị thuộc tính hoặc kiểu của chúng, bao gồm cả hai thuộc tính khoá bắt buộc (PartitionKey và RowKey).

* Hỗ trợ các kiểu thuộc tính: Binary, Bool, DateTime, Double, GUID, Int, Int64, String.

Xem bảng bên dưới để biết giới hạn.

| **Kiểu thuộc tính** |  |
| --- | --- |
| Binary | Kiểu mảng các byte 64 KB. |
| Bool | Kiểu Boolean. |
| DateTime | Kiểu thời gian 64-bit. Miền giá trị được hỗ trợ từ ngày 1/1/1601 đến 12/31/9999. |
| Double | Kiểu số thực 64-bit. |
| GUID | Kiểu dữ liệu để chỉ định ID 128-bit. |
| Int | Kiểu số nguyên 32-bit. |
| Int64 | Kiểu số nguyên 64-bit. |
| String | Kiểu chuỗi. Giá trị chuỗi có thể lên đến 64 KB. |

# Phân vùng bảng

Windows Azure Table cho phép các bảng mở rộng đến hàng ngàn nút lưu trữ bằng cách phân phối các thực thể trong bảng. Khi phân phối các thực thể, nó mong muốn đảm bảo một tập hợp các thực thể luôn nằm cùng trên một nút lưu trữ. Một ứng dụng điều chỉnh thiết lập này bằng cách chọn giá trị thích hợp cho thuộc tính PartitionKey trong mỗi thực thể.

Hình minh họa 1 - Ví dụ phân vùng

Hình mình họa mô tả một bảng chứa nhiều phiên bản của nhiều tài liệu khác nhau. Mỗi thực thể trong bảng tương ứng với một phiên bản của một tài liệu cụ thể. Trong ví dụ này, PartitionKey của bảng là tên tài liệu, và RowKey là tên phiên bản. Tên tài liệu cùng với tên phiên bản xác định duy nhất một thực thể trong bảng. Trong ví dụ này, tất cả các phiên bản của cùng một tài liệu tổ chức vào một phân vùng riêng lẻ.

## Tác động của việc phân vùng

Bây giờ chúng ta mô tả mục đích của các phân vùng và làm thế nào để chọn một PartitionKey.

### Khả năng mở rộng của bảng

Hệ thống lưu trữ đạt được khả năng mở rộng khá lớn bằng cách phân bố các phân vùng qua nhiều nút lưu trữ. Hệ thống quan sát mô thức sử dụng các phân vùng, và tự động cân bằng các phân vùng này qua tất cả các nút lưu trữ. Điều này cho phép hệ thống và ứng dụng của bạn mở rộng để đáp ứng nhu cầu lưu thông của các bảng. Nghĩa là, nếu có nhiều lưu thông đến một vài phân vùng, hệ thống sẽ tự động lan chúng ra đến nhiều nút lưu trữ, vì thế tải lưu thông sẽ được lan ra qua nhiều máy chủ. Tuy nhiên, một phân vùng ,có nghĩa là với tất cả các thực thể có cùng PartitionKey, sẽ được đáp ứng bởi một nút lưu trữ. Tuy nhiên, lượng dữ liệu được lưu trữ trong một phân vùng không bị giới hạn bởi dung lượng lưu trữ của một nút.

### Giao dịch nhóm thực thể

Đối với các thực thể được lưu trữ trong cùng một bảng và cùng một phân vùng, ứng dụng có thể thực hiện một giao dịch bao hàm nhiều thực thể. Điều này cho phép giao dịch thực hiện nhiều thao tác Thêm/ Xóa/ Sửa qua nhiều thực thể với chỉ một request batch đến hệ thống lưu trữ, với điều kiện tất cả các thực thể ở cùng phân vùng trong một bảng. Hoặc tất cả các thao tác thực hiện thành công trong một giao dịch hoặc chúng bị thất bại hoàn toàn.

### Miền thực thể

Các thực thể trong một phân vùng được lưu trữ cùng nhau. Điều này cho phép truy vấn hiệu quả trong một phân vùng.

Trong ví dụ trên, tất cả các phiên bản của cùng một tài liệu tổ chức trong một phân vùng riêng. Do đó, việc truy xuất “tất cả các phiên bản của một tài liệu xác định” sẽ đạt hiệu quả cao, bởi vì chúng ta truy xuất đến một phân vùng duy nhất. Mặt khác, một truy vấn “tất cả các phiên bản của các tài liệu được chỉnh sửa trước 5/30/2007” không bị giới hạn ở một phân vùng duy nhất. Bởi vì truy kiểm tra tất cả các phân vùng trên nhiều nút lưu trữ, một truy vấn như vậy sẽ tốn nhiều chi phí hơn.

## Chọn một PartitionKey

Việc lựa chọn một PartitionKey đóng vai trò quan trọng để một ứng dụng có khả năng mở rộng tốt. Có một sự đánh đổi ở đây giữa việc cố gắng để được lợi từ miền thực thể, ở đó bạn đạt được những truy vấn hiệu quả qua các thực thể trong cùng một phân vùng, và khả năng mở rộng của các bảng, ở đây càng nhiều phân vùng trong bảng thì càng dễ dàng ở cho Windows Azure Table lan truyền tải qua nhiều máy chủ.

### Giao dịch nhóm thực thể

Nếu ứng dụng của bạn cần sử dụng giao dịch nhóm thực thể, một PartitionKey cần phải được lựa chọn để qui mô của nó có thể bao trùm hết tất cả các thực thể bạn cần thực hiện giao dịch trên đó. Tùy theo nhu cầu truy vấn của bạn, cần chọn PartitionKey mà nó chỉ nhóm cùng các thực thể cần được nhóm lại với nhau, để thực hiện giao dịch nhóm thực thể trên chúng.

### Truy vấn hiệu quả

Chúng ta nhận thấy rằng truy vấn có tần suất cao sử dụng PartitionKey như một điều kiện lọc truy vấn. Sử dụng ParititionKey trong bộ lọc truy vấn hạn chế thực thi truy vấn đến một phân vùng đơn hoặc thành phần con của phân cùng (tùy thuộc vào điều kiện sử dụng), do đó cải thiện được hiệu suất truy vấn.

Nếu PartitionKey không phải là một phần của truy vấn, thì truy vấn phải đi qua tất cả các phân vùng trong bảng để tìm ra thực thể cần tìm, điều này thực sự không mang lại hiệu quả.

Sau đây là một số hướng dẫn cơ bản và các đề xuất cho cách chọn một PartitionKey để bảng của bạn cho hiệu quả truy vấn cao:

1. Đầu tiên xác định các thuộc tính quan trọng trong bảng của bạn. Đây là những thuộc tính thường xuyên sử dụng như bộ lọc truy vấn.
2. Lựa chọn những khóa có khả năng từ những thuộc tính quan trọng này.
   1. Điều quan trọng là xác định các truy vấn chi phối đến khối lượng công việc của ứng dụng. Từ những truy vấn chi phối này, lựa chọn các thuộc tính được sử dụng trong bộ lọc truy vấn.
   2. Đây là tập hợp các thuộc tính khóa ban đầu của bạn.
   3. Sắp xếp các thuộc tính khóa theo tầm quan trọng trong truy vấn của bạn.
3. Các thuộc tính khóa này xác định độc nhất các thực thể chưa ? Nếu chưa, thêm định danh duy nhất (ID) vào tập hợp các khóa.
4. Nếu bạn chỉ có một thuộc tính khóa, sử dụng nó làm PartitionKey.
5. Nếu bạn chỉ có 2 thuộc tính khóa, sử dụng thuộc tính đầu tiên làm ParitionKey, và thuộc tính thứ 2 làm RowKey.
6. Nếu bạn có hơn 2 thuộc tính khóa, bạn có thể ghép chúng thành hai nhóm – nhóm ghép đầu tiên là PartitionKey, và nhóm thứ hai là RowKey. Với PartitionKey có thể chứa hai khóa cách nhau bởi dấu “-“.

### Khả năng mở rộng

Bây giờ ứng dụng có tập hợp các khóa tiềm năng của nó, bạn cần chắc chắn rằng phân vùng đã chọn có khả năng mở rộng :

1. Dựa vào PartitionKey trên, nó có dẫn đến những phân vùng có thể trở nên quá nóng khi đáp ứng từ một máy chủ đơn lẻ không? Một cách để xác định điều này bằng cách thực hiện kiểm thử “stress test Table partition”. Để thực hiện kiểm thử này, cần tạo ra một bảng tương tự sử dụng những khóa của bạn và sau đó gây ứng suất lớn đến khối lượng công việc của bạn lên một phân vùng đơn lẻ để đảm bảo rằng phân vùng có thể đáp ứng thông lượng mong muốn cho ứng dụng của bạn.
2. Nếu kiểm thử “stress test Table partition” thành công, bạn đã hoàn thành.
3. Nếu kiểm thử “stress test Table partition” không thành công, lựa chọn một PartitionKey có tính chất tốt hơn. Điều này có thể được thực hiện bằng cách chọn một PartitionKey khác hoặc chỉnh sửa PartitionKey hiện tại (ví dụ, bằng cách liên kết nó với thuộc tính khóa tiếp theo). Mục đích của điều này là tạo ra nhiều phân vùng để một phân vùng riêng lẻ không trở nên quá lớn hoặc quá nóng.
4. Chúng ta đã thiết kế hệ thống có thể mở rộng và có khả năng xử lí một khối lượng lưu thông lớn. Tuy nhiên với tần suất truy vấn cực lớn có thể dẫn đến lỗi truy vấn. Trong trường hợp này, cần giảm tần suất truy vấn của bạn hoặc khử lỗi.

# Lập trình Bảng

Các hoạt động cơ bản sau đây được hỗ trợ trên các bảng và các thực thể

* Tạo bảng hoặc thực thể.
* Lấy một bảng hoặc thực thể, với bộ lọc.
* Cập nhật một thực thể (không có cập nhật bảng).
* Xóa một bảng hoặc thực thể.
* Giao dịch nhóm thực thể - hỗ trợ các giao dịch qua các thực thể trong cùng một bảng và cùng một phân vùng.

Để sử dụng bảng trong ứng dụng .Net, bạn có thể sử dụng dịch vụ dữ liệu ADO.NET.

Bảng phía dưới tóm tắt các API. Bởi vì API dịch vụ dữ liệu.NET ADO.NET thực hiện truyền các gói tin REST, nên ứng dụng có thể chọn sử dụng trực tiếp REST.

| **Nghiệp vụ** | **ADO.NET Data Services** | **HTTP Verb** | **Tài nguyên** | **Mô tả** |
| --- | --- | --- | --- | --- |
| Truy vấn | LINQ Query | GET | Bảng | Trả về danh sách các bảng trong tài khoản lưu trữ. Nếu một bộ lọc hiện có, nó sẽ trả về các bảng tương ứng với bộ lọc này. |
| Thực thể | Trả về tất cả các thực thể trong một bảng xác định hoặc tập con các thực thể nếu các tiêu chuẩn lọc được xác định. |
| Cập nhật toàn bộ thực thể | UpdateObject & SaveChanges(SaveChangesOptions.ReplaceOnUpdate) | PUT | Thực thể | Cập nhật giá trị thuộc tính trong một thực thể. Thao tác PUT thay thế toàn bộ thực thể và có thể được sử dụng để thay thế các thuộc. |
| Cập nhật một phần thực thể | UpdateObject & SaveChanges() | MERGE | Thực thể | Cập nhật giá trị thuộc tính với một thực thể. |
| Tạo một thực thể mới | AddObject & SaveChanges() | POST | Bảng | Tạo một bảng mới trong tài khoản lưu trữ. |
| Thực thể | Thêm một thực thể mới vào bảng. |
| Xóa thực thể | DeleteObject & SaveChanges() | DELETE | Bảng | Xóa một bảng trong tài khoản lưu trữ. |
| Thực thể | Xóa một thực thể từ bảng. |
| Giao dịch nhóm thực thể. | SaveChanges(SaveChangesOptions.Batch) | POST | $batch | Hỗ trợ giao dịch nhóm thực thể được cung cấp thông qua một loạt hoạt động trên các thực thể có cùng partition key trong một bảng duy nhất. Trong ADO.NET Data Services, tùy chọn các lệnh SavaChanges. In ADO.NET Data Services, the option to SaveChanges dictates that the request needs to be sent as a single transaction. |

Các hoạt động cấp cao sẽ được thảo luận chi tiết hơn ở phần sau như :

* Phân trang.
* Xử lí xung đột do truy xuất đồng thời.

## Đặt tên phiên bản

Với tất cả các giải pháp của bộ lưu trữ Windows Azure, chúng tôi đã giới thiệu header HTTP mới gọi được gọi là “x-ms-version”. Tất cả những thay đổi đến API sẽ được đặt tên phiên bản trên header này. Điều này cho phép các phiên bản trước của các lệnh xử lí trên hệ thống lưu trữ tiếp tục làm việc, và chúng ta mở rộng khả năng của các lệnh hiện tại và giới thiệu những lệnh mới.

x-ms-version nên được chỉ rõ cho tất cả các request đến bộ lưu trữ Windows Azure. Nếu có một request anonymous mà không có phiên bản, thì phiên bản cũ nhất của lệnh sẽ được xử lí trên hệ thống lưu trữ.

Phiên bản được hỗ trợ hiện tại là “x-ms-version: 2009-04-14”. Phiên bản này có thể được sử dụng cho tất cả các lệnh và request gửi đến bộ lưu trữ Windows Azure.

// thêm header phiên bản sử dụng sự kiện SendingRequest.

context.SendingRequest +=

new EventHandler<SendingRequestEventArgs>(

delegate(object sender, SendingRequestEventArgs requestArgs)

{

HttpWebRequest request = requestArgs.Request as HttpWebRequest;

request.Headers.Add(

"x-ms-date",

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

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

// ... add authorization header using shared key lite

});

## Chạy thử ví dụ

Trong ví dụ dưới đây, chúng tôi mô tả các hoạt động trên bảng “Blogs”. Bảng này được sử dụng để chứa các blog trong ứng dụng MicroBlogging.

Ứng dụng MicroBlogging có hai bảng - Channels và Blogs. Có danh sách các kênh, và các blog được đưa lên một kênh riêng. Cho ứng dụng này, người dùng sẽ đăng ký vào các kênh và họ nhận được các blog mới cho những kênh này mỗi ngày.

Trong ví dụ này, chúng tôi chỉ tập trung vào các bảng blogs, và ví dụ về các bước sau cho bảng Blog:

* + 1. Định nghĩa giản đồ của bảng.
    2. Tạo bảng.
    3. Thêm blog vào bảng.
    4. Lấy danh sách các blog từ bảng.
    5. Cập nhật blog trong bảng.
    6. Xóa blog từ bảng.
    7. Thêm nhiều Blog vào một bảng.

## Định nghĩa lớp thực thể cho bảng

Giản đồ cho một bảng được định nghĩa qua một lớp C#. Đây là mô hình sử dụng dịch vụ dữ liệu ADO.NET.

Định nghĩa thực thể cho các thực thể Blog sẽ được lưu trong bảng Blog. Mỗi thực thể blog có các thông tin sau :

1. ChannelName - Tên kênh (blog được gửi lên kênh này).
2. The posted date – Ngày gửi.
3. Text – Nội dung của blog.
4. Rating – Độ yêu mến của blog này.

Trong bảng “Blogs”, chúng ta chọn tên kênh làm PartitionKey và ngày gửi làm RowKey. PartitionKey và RowKey là những cái khóa của bảng “Blogs” và điều này được chỉ định bằng cách khai báo các khóa trong thuộc tính của lớp “DataServiceKey”. Bảng “Blogs” được phân chia bởi ChannelName. Điều này cho phép các ứng dụng lấy hiệu quả các blog mới nhất cho một kênh mà người dùng đã đăng ký. Thêm vào các khóa, các thuộc tính đặc trưng của người dùng được khai báo như một thuộc tính. Tất cả các thuộc tính với hàm get và set được public được lưu trữ trong bảng Windows Azure. Vì thế, trong ví dụ bên dưới:

* Text và Rating được lưu trữ cho những đối tượng thực thể trong bảng Azure.
* RatingAsString thì không được lưu trữ bởi vì nó không có định nghĩa làm set.
* ID thì không được lưu trữ vì các hàm truy xuất không được public.

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

public class Blog

{

// ChannelName

public string PartitionKey { get; set; }

// PostedDate

public string RowKey { get; set; }

// User defined properties

public string Text { get; set; }

public int Rating { get; set; }

public string RatingAsString { get; }

protected string Id { get; set; }

}

## Tạo bảng

Tiếp theo, chúng ta thấy làm thế nào để tạo ra những bảng “Blogs” cho tài khoản lưu trữ của bạn. Tạo ra một bảng cũng giống như tạo một thực thể trong một bảng chính gọi là “Tables”. Mọi tài khoản lưu trữ có một bảng chính đã được định nghĩa, và mọi bảng được sử dụng bởi một tài khoản lưu trữ phải đăng kí tên bảng với bảng chính này. Định nghĩa lớp cho bảng chính này được trình bày dưới đây, thuộc tính TableName đại diện cho tên của bảng được tạo ra.

[DataServiceKey("TableName")]

public class TableStorageTable

{

public string TableName { get; set; }

}

Việc tạo ra một bảng được thực hiện như sau.

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

DataServiceContext context = new DataServiceContext(serviceUri);

TableStorageTable table = new TableStorageTable("Blogs");

// Tạo một bảng mới bằng cách thêm thực thể mới vào bảng chính gọi là "Tables"

context.AddObject("Tables", table);

// SaveChanges đưa đến một lời gọi đến máy chủ

DataServiceResponse response = context.SaveChanges();

## Thêm 1 Blog

Để thêm một thực thể, ứng dụng phải làm như sau.

1. Tạo một đối tượng C# mới và thiết lập tất cả các thuộc tính.
2. Tạo một đối tượng của DataServiceContext, đại diện cho một kết nối đến máy chủ trong dịch vụ lưu trữ ADO .NET cho tài khoản lưu trữ của bạn.
3. Thêm đối tượng C# vào context.
4. Gọi phương thức SaveChanges trên DataServiceContext để gửi yêu cầu đến máy chủ. Tại thời điểm này một truy vấn HTTP được gửi đến máy chủ với thực thể định dạng ATOM XML.

Sau đây là đoạn mã ví dụ :

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();

## Truy vấn Blog

Thực thể được truy vấn sử dụng LINQ. Trong ví dụ này chúng ta lấy tất cả những blog có rating = 3.

Khi một truy vấn được liệt kê (ví dụ với lệnh foreach), một truy vấn được gửi đến máy chủ. Máy chủ gửi kết quả theo định dạng ATOM XML. Thư viện dịch vụ dữ liệu ADO .NET lưu kết quả vào đối tượng C# cho ứng dụng sử dụng.

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

DataServiceContext context = new DataServiceContext(serviceUri);

// Truy vấn LINQ sử dụng DataServiceContext cho tất cả các thực thể từ

// bảng Blogs có rating = 3

var blogs =

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

where blogs.Rating == 3

select blog;

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

## Cập nhật một Blog

Để cập nhật một thực thể, ứng dụng nên làm như sau.

1. Tạo một DataContext với MergeOption được gán là OverwriteChanges hoặc PreserveChanges (sẽ được nói rõ ở mục 4.10). Để chắc rằng Etag được đảm bảo một cách chính xác cho mỗi đối tượng mà nó nhận được.
2. Nạp thực thể cần cập nhật vào DataContext bằng LINQ. Lấy nó từ máy chủ sẽ đảm bảo rằng Etag đã được cập nhật trong thực thể và những thao tác cập nhật và xóa tiếp theo sẽ cập nhật Etag trong header if-match. Chỉnh sửa đối tượng C# đại diện cho thực thể.
3. Thêm đối tượng C# trở về cùng một DataContext để cập nhật. Sử dụng cùng DataContext đảm bảo chúng ta sẽ tự động tái sử dụng Etag đã được lấy trước đó cho đối tượng này.
4. Gọi SaveChanges để gửi truy vấn đến máy chủ.

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();

## Xóa một Blog

Xóa một thực thể tương tự như cập nhật một thực thể. Sử dụng DataServiceContext để lấy thực thể và sau đó gọi phương thức DeleteObject trong context thay vì phương thức UpdateObject.

Blog blog =

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

where blog.PartitionKey == "Channel9"

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

select blog).FirstOrDefault();

context.DeleteObject(blog);

DataServiceResponse response = context.SaveChanges();

## Giao dịch nhóm thực thể

Giao dịch nhóm thực thể hỗ trợ thực thi lên đến 100 lệnh thêm-xóa-sửa trong một lệnh batch đơn đối với thực thể trong một bảng và có cùng giá trị PartitionKey. Những lệnh này được thực hiện một thành công tất cả hoặc thất bại toàn bộ.

Ở đây có một vài thuật ngữ :

* Change set – Nhóm một hoặc nhiều lệnh Thêm-Xóa-Sửa cần được xử lí.
* Batch – Một batch chứa một hoặc nhiều change set hoặc truy vấn.

Sau đây là những giới hạn khi sử dụng lệnh batch cho Windows Azure Table:

* Chúng ta cho phép tối đa 100 thao tác trong một change set.
* Chúng ta cho phép chỉ một change set đơn hoặc truy vấn với mỗi thao tác batch. Truy vấn không thể được kết hợp thao tác Thêm-Xóa-Sửa trong một thao tác batch. Chúng ta chỉ hỗ trợ những truy vấn trả về chỉ một dòng, nghĩa là bộ lọc có PartitionKey và RowKey.
* Một batch đơn lẻ có thể nhiều nhất 4MB kích thước.
* Tất cả các lệnh trong change set phải thuộc cùng một phân vùng đơn lẻ trong một bảng duy nhất. Nghĩa là giá trị partition key phải giống nhau cho tất cả các thực thể và các thao tác nên xử lí dựa theo một bảng đơn lẻ.
* Mỗi thực thể có thể chỉ có xuất hiện một lần trong change set. Bạn không thể có change set mà thực thi nhiều thao tác trên cùng một thực thể. Change set có nghĩa thực hiện nhiều thao tác trên nhiều thực thể khác nhau trong cùng một phân vùng và bảng như một giao dịch nguyên tử.
* Một lệnh đơn lẻ trong change set được thực thi theo thứ tự mà chúng xuất hiện trong change set.

Trong khi đưa ra một giao dịch, ETag được gửi trong mỗi change set. Trên máy chủ, chúng tôi kiểm tra xem ETag có khớp với giá trị được lưu trữ trên máy chủ. Nếu có một Etag không khớp, toàn bộ giao dịch không thành công.

Sau đây tiếp tục ví dụ Blog cho thấy lợi ích của giao dịch nhóm thực thể để thêm, xóa, sửa qua nhiều thực thể từ một phân vùng trong một thao tác batch.

// newBlogs là danh sách những blog mới trong phân vùng Channel\_19

// deletedBlogs là danh sách những blog cần xóa trong phân vùng Channel\_19

// updatedBlogs is danh sách những blog cần cập nhật trong phân vùng Channel\_19

// Chúng ta giả định rằng deletedBlogs và updatedBlogs đã được nạp vào

// context trong những truy vấn trước đó.

// Thêm nhiều đối tượng blog cho phân vùng "Channel\_19".

// newBlogs là danh sách blog được chọn để thêm vào

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

{

context.AddObject(newBlogs[index]);

}

// Xóa đối tượng blogs cho Channel\_19.

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

{

context.DeleteObject(deletedBlogs[index]);

}

// Cập nhật đối tượng blogs ch Channel\_19.

// updatedBlogs là danh sách các Blog được chọn để tăng rating.

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

{

updatedBlogs[index].Rating++;

context.UpdateObject(updatedBlogs[index]);

}

// Tất cả các thao tác Thêm/ Xóa/ Sửa được xử lí với một request batch

DataServiceResponse response =

context.SaveChanges(SaveChangesOptions.Batch);

Chỉ định SaveChangeOptions.Batch chỉ dẫn lệnh SaveChanges rằng tất cả những thay đổi chưa thực hiện được gom nhóm vào một change set đơn lẻ và gửi một giao dịch nhóm thực thể đơn lẻ đến hệ thống lưu trữ. Giao dịch batch sau đó được kiểm tra trên máy chủ để chắc rằng nó tuân thủ các hạn chế trên.

Bây giờ, nếu SaveChanges trên được thực hiện mà không có SaveChangesOptions.Batch, sau đó mỗi thực thể Thêm/Xóa/Sửa sẽ được gửi như một yêu cầu riêng lẻ đến hệ thống lưu trữ và thực thi riêng biệt.

Dưới đây là xử lí truy vấn đơn lẻ trong một lệnh batch. Chú ý, một lệnh batch có thể là một giao dịch nhóm thực thể như ở trên, hoặc truy vấn đơn lẻ như ở dưới.

// Thực hiện một truy vấn trong batch- Chú ý bộ lọc nên để theo dạng

// PartitionKey == 'Giá trị' && RowKey == 'Giá trị'

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);

### Lỗi

Khi một truy vấn batch bị lỗi, sẽ không có thao tác nào thành công. Tuy nhiên, điều quan trọng là làm cho ứng dụng biết được cái lệnh gây ra toàn bộ giao dịch thất bại. Để xác định điều này, chúng ta trả về những lệnh được đánh số tuần tự trong thông báo lỗi như được thấy trong phần đánh dấu dưới đây. Tuy nhiên, nếu lỗi xảy ra bởi vì chứng thực hoặc bất kì lí do nào khác, điều này sẽ không được quy cho cho một lệnh đơn nào, do đó các con số sẽ không được cung cấp. Do đó, payload trong một sự kiện thất bại sẽ bao gồm:

1. Khóa của thao tác bị thất bại.
2. Các số không dựa trên chỉ số của hoạt động gây ra thất bại trong batch (chỉ số bắt đầu từ 0).
3. Chuỗi mô tả thất bại.

Ví dụ Response:

HTTP/1.1 202 Accepted

Cache-Control: no-cache

Transfer-Encoding: chunked

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

Server: Table Service Version 1.0 Microsoft-HTTPAPI/2.0

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

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

6E7

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

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

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

Content-Type: application/http

Content-Transfer-Encoding: binary

HTTP/1.1 400 Bad Request

Content-ID: 4

Content-Type: application/xml

Cache-Control: no-cache

DataServiceVersion: 1.0;

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

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

<code>InvalidInput</code>

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

</error>

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

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

0

Khi một lỗi xảy ra, bạn có thể lấy tuần tự số lệnh bằng đoạn mã sau :

try

{

// ... save changes

}

catch (InvalidOperationException e)

{

DataServiceClientException dsce = e.InnerException as DataServiceClientException;

int? commandIndex;

string errorMessage;

ParseErrorDetails(dsce, out commandIndex, out errorMessage);

}

void ParseErrorDetails(

DataServiceClientException e,

out string errorCode,

out int? commandIndex,

out string errorMessage)

{

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

commandIndex = null;

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

if (indexOfSeparator > 0)

{

int temp;

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

{

commandIndex = temp;

errorMessage = errorMessage.Substring(indexOfSeparator + 1);

}

}

}

void GetErrorInformation(

string xmlErrorMessage,

out string errorCode,

out string message)

{

message = null;

errorCode = null;

XName xnErrorCode = XName.Get(

"code",

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

XName xnMessage = XName.Get(

"message",

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

using (StringReader reader = new StringReader(xmlErrorMessage))

{

XDocument xDocument = null;

try

{

xDocument = XDocument.Load(reader);

}

catch (XmlException)

{

// The XML could not be parsed. This could happen either because the connection

// could not be made to the server, or if the response did not contain the

// error details (for example, if the response status code was neither a failure

// nor a success, but a 3XX code such as NotModified.

return;

}

XElement errorCodeElement =

xDocument.Descendants(xnErrorCode).FirstOrDefault();

if (errorCodeElement == null)

{

return;

}

errorCode = errorCodeElement.Value;

XElement messageElement =

xDocument.Descendants(xnMessage).FirstOrDefault();

if (messageElement != null)

{

message = messageElement.Value;

}

}

}

## Cách làm tốt nhất khi sử dụng DataServiceContext

Ở đây có một số áp dụng tốt nhất khi sử dụng DataServiceContext

* DataServiceContext không có một thời gian sống dài. Thay vì có một DataServiceContext riêng lẻ cho sự sống của một tiến trình, có một lời khuyên là tạo một đối tượng DataServiceContext mỗi lần bạn muốn thực hiện một tập các giao và sau đó xóa đối tượng này khi hoàn thành.
* Khi sử dụng cùng một thực thể DataServiceContext qua tất cả các lệnh thêm/xóa/sửa, nếu ở đó có lỗi khi thực hiện SaveChanges, thao tác bị lỗi được giữ theo dõi trong DataServiceContext và cố gắng thử lại trong lần SaveChanges tiếp theo được gọi.
* DataServiceContext có MergeOption, được sử dụng để quản lí làm thế nào DataServiceContext xử lí các thực thể được theo dõi. Những giá trị có thể là:
  + AppendOnly: Đây là giá trị mặc định ở đó DataServiceContext không tải phiên bản thực thể từ máy chủ nếu nó đã có sẵn trong bộ nhớ của nó.
  + OverwriteChanges : DataServiceContext luôn luôn tải phiên bản thực thể từ máy chủ do đó tiếp tục cập nhật nó và ghi đè lên thực thể trước đó.
  + PreserveChanges: Khi một phiên bản thực thể tồn tại trong DataServiceContext, nó không được tải từ bộ nhớ ổn định. Bất kì thuộc tính nào thay đổi thì đối tượng trong DataServiceContext được giữ vững nhưng Etag được cập nhật do đó đây là lựa chọn tốt để phục hồi từ những lỗi đồng thời truy xuất.
  + NoTracking: Phiên bản thực thể không được lưu vết bởi DataServiceContext. Cập nhật một thực thể không được lưu vết sẽ yêu cầu sử dụng AttachTo để cập nhật Etag để đảm bảo thực hiện được cập nhật và đó lựa chọn này không được khuyên dùng.

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

context.UpdateObject(blog);

context.SaveChanges();

## Sử dụng API REST

Tất cả những thao tác ở trên cho kết quả thông báo HTTP đến và đi từ server. Một ứng dụng có thể chọn làm việc ở cấp độ HTTP/REST thay vì sử dụng thư viện .Net.

# Cập nhật đồng thời

Để cập nhật một thực thể yêu cầu những bước sau.

1. Lấy thực thể từ máy chủ.
2. Cập nhật đối tượng cục bộ, và gửi nó trở lại máy chủ.

Chúng ta giả định rằng có hai xử lí đồng thời cố gắng cập nhật cùng một thực thể. Bởi vì bước 1 và bước 2 không phải là atomic,nên một trong số chúng có thể kết thúc thay đổi phiên bản cũ của thực thể. Windows Azure Table sử dụng mô hình optimistic concurrency để giải quyết vấn đề này.

1. Mỗi thực thể có một phiên bản được duy trì trên hệ thống. Phiên bản này được thay đổi bởi máy chủ sau mỗi lẫn cập nhật.
2. Khi một thực thể được lấy ra, máy chủ gửi phiên bản này về phía client bằng HTTP ETag.
3. Khi client gửi yêu cầu cập nhật đến máy chủ, nó gửi Etag này đến máy chủ bằng header If-Match.
4. Nếu phiên bản của thực thể trên máy chủ giống với Etag trong header If-Match, việc thay đổi được chấp nhận, và thực thể được lưu trữ lấy phiên bản mới nhất trên máy chủ. Phiên bản mới được trả về phía client bằng Etag header.
5. Nếu phiên bản của thực thể trên máy chủ khác với Etag trong header If-Match, việc thay đổi bị từ chối, và lỗi HTTP “điều kiện tiên quyết thất bại” được trả về phía client.

Khi ứng dụng client nhận được lỗi “điều kiện tiên quyết không thành công”, hành vi đặc trưng của nó là thử lại thao tác bằng đoạn mã phía bên dưới.

1. Nó nên thử lại đối tượng một lần nữa, do đó lấy được phiên bản mới nhất.
2. Thực hiện cập nhật cục bộ và gửi nó trở lại máy chủ.

Nếu ứng dụng sử dụng thư viện .NET client, mã lỗi HTTP được biểu hiện đến ứng dụng qua exception (DataServiceRequestException).

Trong ví dụ bên dưới, hai client khác nhau thực thi cùng đoạn mã để thay đổi nội dung. Hai client gán thuộc tính “Text” với hai giá trị khác nhau. Sau đây là một chuỗi các sự kiện có khả năng minh họa làm thế nào để cập nhật đông thời được xử lí.

1. Cả hai client lấy thực thể về. Và cũng lấy Etag theo thực thể ví dụ là “v1”.
2. Mỗi client cập nhật thuộc tính Text ở cục bộ.
3. Mỗi client gọi UpdateObject và SaveChanges.
4. Mỗi client gửi truy vấn HTTP đến máy chủ với header “If-Match:v1”
5. Một trong những client đến máy chủ trước.
   1. Máy chủ so sánh header “If-Match” với phiên bản của thực thể. Chúng bằng nhau
   2. Máy chủ thực hiện thay đổi.
   3. Thực thể nhận một phiên bản mới “v2” trên máy chủ.
   4. Một header mới “Etag:v2” được gửi lại qua response đến client.
6. Client khác đến máy chủ tiếp theo. Tại thời điểm này, client đầu tiên đã thay đổi ứng dụng.
   1. Máy chủ so sánh header “If-Match” với phiên bản của thực thể. Chúng không bằng nhau, bởi vì phiên bản của thực thể đã được thay đổi thành “v2”, trong khi phiên bản của request là “v1”.
   2. Máy chủ từ chối truy vấn.

context.MergeOption = MergeOption.PreserveChanges;

Blog blog =

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

where blog.PartitionKey == "Channel9"

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

select blog).FirstOrDefault();

blog.Text = "Hi there again";

try

{

context.UpdateObject(blog);

DataServiceResponse response = context.SaveChanges();

}

catch (DataServiceRequestException e)

{

OperationResponse response = e.Response.First();

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

{

// query the object again to retrieve the latest etag

// and update it

}

}

## Cập nhật không điều kiện

Nếu một ứng dụng muốn cập nhật một thực thể không điều kiện nên thực hiện theo những bước sau

1. Tạo mới một DataServiceContext hoặc nếu sử dụng một DataServiceContext tồn tại, tách rời đối tượng như ví dụ bên dưới.
2. Nối thực thể với context và sử dụng “\*” bằng giá trị Etag mới.
3. Cập nhật thực thể.
4. Gọi SaveChanges.

// gán merge option thành overwrite để cho phép các thực thể được cập nhật

context.Detach(blog);

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

blog.Text = "Hi there again";

try

{

context.UpdateObject(blog);

DataServiceResponse response = context.SaveChanges();

}

catch (DataServiceRequestException e)

{

// Lỗi xử lí – nhưng có không quăng lỗi PreCondition

}

# Phân trang các kết quả truy vấn

Đối với các truy vấn có thể trả lại số lượng lớn các kết quả, hệ thống cung cấp hai cơ chế:

1. Có khả năng lấy N thực thể đầu tiên sử dụng chức năng Take(N) của LINQ.
2. Một thẻ tiếp nối cho biết nơi tập các kết quả tiếp theo bắt đầu.

## Lấy N thực thể trên cùng

Hệ thống hỗ trợ trả về N thực thể trên cùng phù hợp với truy vấn. Ví dụ, trong .Net, bạn có thể sử dụng chức năng Take(N) của LINQ để lấy N thực thể trên cùng.

Lấy 100 thực thể trong ví dụ dưới đây.

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

DataServiceContext svc = new DataServiceContext(serviceUri);

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

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

{

// do something with each blog

}

Chức năng này cũng được hỗ trợ qua giao thức REST với chuỗi truy vấn $top=N. Ví dụ, truy vấn “GET http://<serviceUri>/Blogs?$top=100” sẽ được trả về 100 thực thể trên cùng, ứng với truy vấn. Việc lọc được thực hiện trên máy chủ lấy chỉ 100 thực thể để gửi trong response trả về cho client.

## Thẻ tiếp nối

Số lượng thực thể được trả về trong truy vấn có thể bị giới hạn bởi một trong những nguyên nhân sau :

1. Truy vấn xác định số lượng tối đa thực thể được trả về.
2. Số lượng thực thể lớn hơn số lượng thực thể tối đa được cho phép trong một response từ máy chủ (hiện tại là 1000).
3. Số lượng kích thước thực thể trong một response lớn hơn kích thước tối đã của một response (hiện tại là 4MB bao gồm tên thuộc tính nhưng loại trừ khi cú pháp xml được sử dụng cho REST).
4. Thời gian xử lí lâu hơn thời gian chờ của máy chủ (hiện tại là 60 giây).

Trong mỗi trường hợp này, response bao gồm một thẻ nối tiếp qua header custom. Đối với một truy vấn trên các thực thể của bạn, header custom đại hiện một thẻ nối tiếp là :

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

Truy vấn tiếp theo sẽ như sau

http://<serviceUri>/Blogs?<originalQuery>&NextPartitonKey=<someValue>&NextRowKey=<someOtherValue>

Client sẽ lặp lại điều này cho đến khi không còn thẻ tiếp nối. Lúc này, tất cả những kết quả phù hợp đã được lấy ra.

Thẻ tiếp nối phải được xem như một giá trị mờ. Nó phản ảnh thời điểm bắt đầu cho truy vấn tiếp theo, và nó có thể không tương ứng với thực thể thực tế trong bảng. Nếu một thực thể mới được thêm vào có Khóa(thực thể mới) > Khóa (thực thể cuối cùng nhận được trong truy vấn), nhưng Khóa(thực thể mới) < “Thẻ tiếp tục”, thì thực thể mới này sẽ không được trả về khi truy vấn thực hiện lại sử dụng thẻ tiếp tục. Tuy nhiên, nếu thực thể mới được thêm vào có khóa (thực thể mới) > thẻ tiếp tục, nó sẽ được trả về trong tập con truy vấn sử dụng thẻ tiếp tục.

# Mô hình nhất quán (Consistency Model)

Bây giờ chúng ta mô tả mô hình nhất quán được cung ứng bởi Windows Azure Table.

## Tính nhất quán bảng đơn

Trong một bảng đơn, hệ thống cung cấp giao thức ACID đảm bảo cho tất cả các giao dịch thêm/ xóa/ sửa cho một thực thể đơn lẻ.

Trong một phân vùng, snapshot isolation được cung cấp cho các truy vấn. Một truy vấn sẽ có cái nhìn nhất quán của phân vùng từ thời điểm bắt đầu truy vấn và xuyên suốt trong quá trình giao dịch. Snapshot có những thuộc tính sau.

1. Một giao dịch sẽ không thấy những thay đổi chưa commit từ các giao dịch khác đang xử lí đồng thời. Nó chỉ thấy những thay đổi đã commit trước khi bắt đầu xử lí truy vấn này ở máy chủ.
2. The snapshot isolation mechanism allows reads to happen concurrently with an update to the partition without blocking that update.
3. Cơ chế snapshot isolation cho phép thao tác đọc thực hiện đồng thời với một thao tác cập nhật lên phân vùng không bị khóa bởi lệnh cập nhật.

Snapshot isolation chỉ được hỗ trợ với một phân vùng và với một truy vấn đơn. Hệ thống không hỗ trợ snapshot isolation qua những phân vùng của bảng.

## Tính nhất quán Cross-Tab

Ứng dụng chịu trách nhiệm bảo đảm tính nhất quán qua nhiều bảng.

Trong ví dụ MicroBlogging, chúng ta có hai bảng – Channels và Blogs. Ứng dụng chịu trách nhiệm bảo đảm tính nhất quán giữa bảng Channels và bảng Blogs. Ví dụ, khi một kênh được xóa khỏi bảng Channels, ứng dụng nên xóa những blog tương ứng với kênh đó từ bảng Blogs.

Nhiều lỗi có thể xảy ra khi ứng dụng đồng bộ hóa trạng thái của nhiều bảng. Ứng dụng cần được thiết kế để xử lí những lỗi này.

Trong ví dụ trước, khi một kênh bị xóa khỏi bảng Channels, ứng dụng cũng cần xóa những blog cho kênh này trong bảng Blob. Trong suốt xử lí này, ứng dụng có thể bị lỗi. Để xử lí những lỗi này, ứng dụng cần tiếp tục giao dịch trong Windows Azure Queues, để một worker có thể tiếp tục việc xóa kênh và blog thuộc kênh đó nếu xảy ra thất bại.

Chúng ta tiếp tục với ví dụ là chúng ta có hai bảng “Channels” và Blogs. Một thực thể trong bảng “Channels” có những thuộc tính sau: { Name là PartitionKey, chuỗi trống là RowKey, Owner, CreatedOn} và thực thể trong bảng “Blogs” có {Channel Name là PartitionKey, CreatedOn là RowKey, Title, Blog, UserId}. Bây giờ khi một kênh được xóa, chúng ta cần chắc rằng những blog liên kết với kênh này cũng được xóa. Dưới đây là những bước chúng ta cần làm:

1. Tạo một hàng đợi là “DeleteChannelAndBlogs”
2. Khi một yêu cầu xóa đến cho một kênh tại một web role, xếp entry vào hàng đợi định rõ tên của kênh.
3. Tạo một worker role để nghe những entry được thêm vào hàng đợi “DeleteChannelAndBlogs”.
4. Worker role lấy một entry đầu hàng đợi từ hàng đợi DeleteChannelAndBlogs. Lấy ra một entry xác định tên của kênh cần được xóa. Nếu entry đã được xóa bởi worker role thì entry sẽ được xóa khỏi hàng đợi. Nếu chưa, thì entry sẽ được woker role sử dụng. Khi worker lấy entry nó sẽ thực hiện như sau:
   1. Đánh đấu kênh không hiệu lực trong bảng “Channels”, do đó từ thời điểm này, không ai được đọc nó.
   2. Xóa tất cả những thực thể trong bảng Blog với PartitionKey=”channel name” được nhận trong entry.
   3. Xóa kênh từ bản “Channels”
   4. Xóa entry từ hàng đợi.
   5. Trở về.

# Mẹo và thủ thuật

Trong phần này chúng ta sẽ thảo luận những chiến lược phổ biến mà ứng dụng có thể chạy và những giải pháp có thể sử dụng.

## Lấy những đối tượng mới nhất (Mô phỏng thứ tự giảm dần)

Trong ứng dụng quản lí thư , tin tức, blog, …, Hầu hết những đối tượng mới thêm vào thường được yêu cầu hiển thị đầu tiên. Với tất cả các dòng được sắp xếp theo PartitionKey và RowKey theo thứ tự bảng chữ cái, điều này đòi hỏi một chút khéo léo để lấy những đối tượng đầu tiên được tạo gần đây nhất.

Để làm được đều này, chúng ta có thể gán RowKey với một chuỗi chiều dài cố định tương đương DateTime.MaxValue.Ticks - DateTime.UtcNow.Ticks.

Ví dụ:

Blog blog= new Blog();

// Note the fixed length of 19 being used since the max tick value is 19 digits long.

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

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

blog.RowKey = rowKeyToUse;

Vì thế một blog b1 được viết vào ngày 10/1/2008 10:00:00 AM sẽ có RowKey là 2521794455999999999, và b2 được viết vào ngày 10/2/2008 10:00:00 AM sẽ có RowKey là 2521793591999999999 và do đó b2 sẽ đứng trước b1.

Để lấy tất cả blog được viết sau ngày 10/1/2008 10:00:00 AM, chúng ta sẽ sử dụng truy vấn sau:

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;

Chú ý: Bắt buộc chuỗi độ dài cố định nên đủ lớn để xử lí một phạm vi lớn. Ví dụ : Độ dài tối đã của DateTime.MaxValue.Ticks - DateTime.UtcNow.Ticks là 19 chữ số. Tuy nhiên, nếu chúng ta khởi đầu xử lí ngày bắt đầu 2/15/6831 2:13:20 PM, độ dài là 18 cơ số. Việc sắp xếp cho chúng ta kết quả không như mong muốn nếu chúng ta không đệm những giá trị với số 0 ở đầu bởi vì việc sắp xếp là theo thứ tự từ điển nghĩa là 92>10000. Tuy nhiên nếu chúng ta đệm nó thích hợp, 10000 > 00092.

Từ phiên bản CTP, chúng ta cho phép chỉ một RowKey. Tuy nhiên, RowKey hiện tại có thể được sử dụng để đại diện cho nhiều khóa qua việc nối chuỗi. Ví dụ, chúng ta giả định rằng một blog có thể được đánh giá từ 0-5 và chúng ta muốn chắc rằng blog được liệt kê theo danh sách mà chúng được sắp xếp (Rating sắp xếp giảm, Created time sắp xếp giảm). Điều này nghĩa là chúng sắp xếp giảm theo đánh giá trước, rồi sau đó với sắp xêp với mỗi đánh giá theo thời gian tạo. Ở đó chúng ta có thể gán RowKey là <Rating>+<Ticks>. Vì thế nói về ví dụ trên, trong hai blog trên chúng ta giả định blog1 được đánh giá là 5 và blog2 được đánh giá là 1. Có Rating đứng trước tick cho phép blog1 được liệt kê trước blog2 mặc dù sự thật là blog2 được thêm vào sau blog1.

Bởi vì PartitionKey và RowKey phải là kiểu chuỗi, bạn có thể sử dụng những qui ước sau để sử dụng những kiểu khác nhau trong những thuộc tính này. Điều này đảm bảo rằng các giá trị được sắp xếp theo cách dự kiến. Ví dụ :

* Số nguyên – Lưu trữ nó với giá trị kích thước cố định dẫn đầu là những số 0.
* DateTime – lưu trữ nó bằng yyyy-mm-dd, hoặc yyyy-mm-dd-hh-ss.

## Lấy ra bằng cách sử dụng tiền tố

Ta có thể sử dung >= và < trong điều kiện lọc . Ví dụ :

var blogs =

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

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

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

select blog;

## Ví dụ phân vùng dữ liệu

Phân vùng dữ liệu đòi hỏi gom nhóm các thực thể của một bảng vào những phân vùng để một phân vùng trở thành đơn vị nhỏ nhất được điều khiển bởi việc cân bằng tải. Windows Azure Table sử dụng PartitionKey để gom nhóm các thực thể vào các phân vùng. Trong phần này chúng ta thảo luận làm thế nào để phân vùng dữ liệu của bạn để đạt hiệu quả cao nhất.

### Tình huống nghiên cứu Micro Blogging

Chúng ta lấy ví dụ của ứng dụng micro blogging ở đó người dung có thể tạo những kênh (hoặc danh mục) và tạo những blog trong chúng. Khi người dùng chọn một kênh, một trang chứa các blog được hiển thị.

Chúng ta giả định rằng một blog trong bảng Blog được xác định duy nhất bằng tên của kênh và ngày blog được tạo. Truy vấn chi phôi chúng ta tối ưu hóa bảng là “Lấy 10 blog mới nhất cho một kênh”.

Trong ví dụ này chúng ta xem xét những lựa chọn khác nhau để chọn PartitionKey ở đó chúng ta sẽ thay đổi kích thước phân vùng từ rất lớn đến kích thước phân vùng chỉ cho một thực thể đơn. Điều này thay đổi kích thước và số lượng phân vùng.

#### Có một vài phân vùng lớn

Một lựa chọn hiển nhiên cho PartitionKey là tên kênh. Chúng ta cũng sẽ sử dụng các phương pháp được mô tả trong mục 8.1 để chắc rằng các blog được sắp xếp theo RowKey từ blog được thêm vào mới nhất đến blog cũ nhất trong kênh. Dưới đây là định nghĩa lớp thực thể cho bảng Blog với cách phân vùng như trên:

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

public class Blog

{

// Channel name

public string PartitionKey { get; set; }

// To sort RowKey from newest to oldest blog,

// RowKey is DateTime.MaxValue.Ticks - DateTime.UtcNow.Ticks

public string RowKey { get; set; }

// User defined properties

public string Text { get; set; }

public DateTime CreatedOn { get; set; }

public int Rating { get; set; }

}

Điều này cho phép chúng ta truy vấn hiệu quả tất cả các blog trong kênh “Football” và hiển thị chúng, và chúng sẽ được sắp xếp từ blog được thêm vào mới nhất đến cũ nhất dựa trên cách RowKey được thiết lập ở trên.

var blogs =

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

where blog.PartitionKey == "Football"

select blog;

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

Với giản đồ phân vùng này, đơn vị để cân bằng tải là kênh. Do đó, những blog thuộc về những kênh khác nhau nằm trong những phân vùng khác nhau và những phân vùng này có thể được lan qua nhiều máy chủ khác nhau để phục vụ nhu cầu lưu thông của bảng “Blogs”.

#### Có nhiều phân vùng nhỏ

Bây giờ nếu một kênh có khả năng chứa đến 10 triệu blog với nhiều lĩnh vực thảo luận sôi nổi, nó có thể không phù hợp đến sử dụng tên kênh làm khóa phân vùng, điều này sẽ làm cho một phân vùng riêng lẻ chứa đến 10 triệu blog và các lĩnh vực thảo luận sôi nổi này có thể không được cân bằng đến nhiều máy chủ độc lập khác nhau.

Một giải pháp có thể thực hiện để tránh có những phân vùng monolithic là tạo PartitionKey bằng cách kết hợp tên kênh và ngày blog được tạo ra. Với PartitionKey này sẽ định nghĩa đơn nhất một thực thể, và mỗi phân vùng trong bảng như vậy sẽ có một thực thể duy nhất. Điều này thật tuyệt với để cân bằng tải, do nó cho hệ thống lưu trữ hoàn toàn tự do để cân bằng tải tất cả các thực thể bằng cách nó chọn qua nhiều máy chủ để đáp ứng nhu cầu ứng dụng. Tùy thuộc cách phân vùng như thế nào để phân phối qua nhiều máy chủ, điều này có thể làm những truy vấn như “Lấy cho tôi tất cả những blog trong ngày hôm qua hoặc tuần trước” có khả năng tốn kém, vì nó có thể yêu câu truy xuất rất nhiều nút lưu trữ ,gây ra nhiều tốn kém.

#### Kích thước phân vùng trung gian

Có một lựa chọn thay thế để gom nhóm blog dựa trên kênh cộng với giai đoạn thời gian thô (háng tuần/ hàng tháng/ hàng ngày) dựa trên độ phổ biến của các kênh. Nếu kênh trở nên nóng trong thời gian vài tháng, vài tuần hay vài ngày nào đó, thì nó sẽ thuận lợi để phân vùng những blog sử dụng thời gian mà blog được tạo ra. Bằng cách này, PartitionKey bao gồm cả tên kênh và thông tin giai đoạn thời gian. Điều này làm cho việc phân vùng mịn hơn và hệ thống linh hoạt hơn để lan truyền tải qua nhiều máy chủ để đáp ứng nhu cầu lưu thông. Trong khi đó, bởi vì chu kì thời gian được mã hóa vào PartitionKey, những truy vấn như “Lấy cho tôi tất cả blog trong kênh X trong ngày hôm qua” sẽ hiệu quả hơn, bởi vì chỉ có một phân vùng nhỏ cần được truy vấn.

Ví dụ như kênh “Football”, những giai đoạn nóng có thể suốt mùa giải từ tháng 9 đến tháng 2. Nó sẽ tốt hơn cho ứng dụng để phân vùng bảng theo tháng hoặc theo tuần cộng với thêm kênh. Nếu “week” được sử dụng trong PartitionKey, phân vùng nóng như Super Bowl có thể được cân bằng tải để đảm bảo tính sẵn sang cao và phục vụ truy vấn hiệu quả.

Bởi vì Windows Azure Table hỗ trợ một PartitionKey đơn, ứng dụng micro blogging có thể sử dụng

<Channel\_Name>\_<9999 - Year>\_<5 - Week#> hoặc <Channel\_Name>\_<9999 - Year>\_<13 - Month> là PartitionKey trong bảng này. Lựa chọn giữa việc sử dụng <9999 - Year>\_<13 - Month> hay <9999 - Year>\_<5 - Week#> tùy thuộc vào việc blog được dùng nhiều nhất trên cơ sở hàng tuần hoặc hàng tháng. Trong ví dụ này, we are subtracting the date component from an upper bound, để có các thực thể được sắp xếp từ mới nhất đến cũ nhất. Sau đâu là lớp thực thể ví dụ theo cách tiếp cận phân vùng này:

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

public class Blog

{

// <Channel name>\_<9999-YYYY>\_<13-Month>

public string PartitionKey { get; set; }

// To sort RowKey from newest to oldest blog,

// RowKey is DateTime.MaxValue.Ticks - DateTime.UtcNow.Ticks

public string RowKey { get; set; }

// User defined properties.

public string Text { get; set; }

public DateTime CreatedOn { get; set; }

public int Rating { get; set; }

}

Điều này cho phép chúng ta thực hiện truy vấn sau và hiển thị các blog cho một năm và tháng.

// for 10/2008, we will use <Channel name>\_7991\_03. 7991 = 9999 – 2008; 03 = 13 - 10

var blogs =

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

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

select blog;

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

Nếu muốn có tất cả các blog theo năm , thì truy vấn sau có thể được sử dụng.

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) { ... }

### Dynamically selecting the granularity of the PartitionKey

Trong ví dụ trên, nó có thể hiệu quả để sử dụng phương pháp phân vùng tự động. Sử dụng cùng ví dụ trên, Chúng ta có thể bắt đầu với <Channel\_Name>\_<9999 - Year>\_<13 - Month>. Nhưng trong những tháng được dự đoán có lượng lưu thông nóng, chúng ta có thể chọn làm giảm granularity of the partition size bằng cách sử dụng <Channel\_Name>\_<9999 - Year>\_<13 - Month>\_<5 - Week>. Ứng dụng cần nhận biết được giản đồ phân vùng tự động này. Ví dụ: Sử dụng luật được sử dụng để phân vùng blogs:

1. Between March to July i.e. off season use <Channel\_Name>\_<9999 - Year>\_<13 - Month>
2. Giữa tháng 3 đến tháng 7 nghĩa là lúc mùa giải kết thúc sử dụng <Channel\_Name>\_<9999 - Year>\_<13 - Month>.
3. Rest of the months (i.e. during the NFL football season) use <Channel\_Name>\_<9999 - Year>\_<13 - Month>\_<5 - Week>
4. Với những tháng còn lại (nghĩa là trong suốt mùa đá bóng) sử dụng <Channel\_Name>\_<9999 - Year>\_<13 - Month>\_<5 - Week>

Bây giờ để lấy tất cả blog được viết vào ngày 3/7/2008 UTC, ứng dụng có thể sử dụng truy vấn sau:

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

// long rowKeyEnd = DateTime.MaxValue.Ticks - d2.Ticks where d2 = 4/7/2008 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) { }

Tuy nhiên, để lấy những blogs được tạo vào 3/10/2008 UTC, ứng dụng có thể sử dụng truy vấn sau:

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

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

var blogs =

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

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

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

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

select blog).Take(1000);

foreach (Blog blog in blogs) { }

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

### Những kiểu thực thể khác nhau trong cùng một bảng

Trong nhiều ứng dụng, chúng ta chợt nảy ra dữ liệu đại diện nhiều thực thể khác nhau và có nhu cầu được lấy và xử lí cùng với nhau. Ví dụ, trong một trang mạng xã hội, ở đó người dùng có thể có blogs, photos,… Giao diện người dùng thường gom nhóm những thông tin này cùng với nhau cho một người dùng cá biệt. Ví dụ, trang chủ mạng xã hội của Joe, sẽ hiển thị những thay đổi gần đây nhất của Joe (nghĩa là những bài viết blog, photo, video,… mới nhất). Nó đã được xác định từ trang chủ hiển thị những thay đổi gần đây nhất, chúng ta muốn tối ưu hóa những truy vấn của chúng ta để lấy những thay đổi gần đây nhất một cách hiệu quả. Đưa ra những yêu cầu này, chúng ta có thể chắc chắc rằng dữ liệu được nhóm lại để lấy hiệu qua chỉ qua một bảng đơn, lưu trữ tất cả những thực thể, cùng với “Kind” cho mỗi thực thể, tất cả những thực thể được sắp xếp từ mới nhất đến cũ nhất. Chúng ta có thể phân vùng dữ liệu theo ID người dùng và RowKey có thể được sử dụng để phân biệt những thực thể khác nhau (nghĩa là <Time (newest to oldest)>\_<Entity Kind such as Photo, Blog, Movie, etc.>).

Điều này cho phép bạn sử dụng giao dịch nhóm thực thể đến thêm/xóa/sửa tất cả các thực thể liên quan thuộc về một người dùng riêng lẻ chỉ với 1 giao dịch (xem mục 4.9 ở trên)

Chú ý: Ngoài ra, <Entity kind>\_<Time (newest to oldest)> có thể được sử dụng làm RowKey nếu chiến lược chính là nhận những thực thể theo loại nó thuộc về. Nhưng bởi vì chúng ta muốn tối ưu hóa truy xuất trang chủ với những thay đổi gần nhất, chúng ta sẽ có Thời gian đứng trước Loại trong ví dụ này.

// This union of all entities will be used only for querying. While inserting and updating

// entities, we will use separate classes for each entity kind.

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

public class SocialNetworkEntity

{

// User Id

public string PartitionKey { get; set; }

// To sort the time from newest to oldest

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

// The ticks can be at most 19 digits

public string RowKey { get; set; }

// User defined properties will be union of all entity properties.

// Entity type repeated to allow easy search

public string EntityType { get; set; }

// Blog properties

public string Text { get; set; }

public string Category { get; set; }

// Photo properties

public string Uri { get; set; }

public string ThumbnailUri { get; set; }

// Common to blog and photo

public string Caption { get; set; }

// Common to blog and photo

public DateTime CreatedOn { get; set; }

public int Rating { get; set; }

// This class can be responsible for returning the appropriate

// kind so that business logic does not have to deal with

// the union. The setter is private which prevents it from

// being stored in the store.

public BlogEntity Blog { get; private set }

public PhotoEntity Photo { get; private set }

}

Chú ý: Lớp với hợp tất cả các thuộc tính theo tất cả các thực thể được gom vào một bảng được yêu cầu chỉ cs thể truy vấn bằng ADO.Net Services. If using REST, we do not need this since we have total control on serialization and deserialization. This means that when seeing the Kind on the REST result, we can then populate the correct type with the data fields returned by the REST request.

Bây giờ những truy vấn dưới đây có thể được sử dụng để lấy những thực thể để hiển thị trên trang chủ

var entities =

(from entity in

context.CreateQuery<SocialNetworkEntity>("SocialNetworkTable")

where entity.PartitionKey == userId

select entity).Take(100);

Và truy vấn sau có thể được sử dụng để lấy chỉ hình ảnh được hiển thị trên trang photo:

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);

Chú ý : Trong khi tạo RowKey, chúng ta cần chắc chắn rằng phần đầu là con số độ dài cố định. Chúng ta sử dụng 19 cơ số đển cho phép các thực thể trong phạm vi dài được sử dụng.

Nếu mục tiêu của ứng dụng là lấy các thực thể theo Thể loại của chúng, thì chúng ta muốn Thể loại đứng trước Thời gian trong RowKey. Nhưng trong ví dụ này, ứng dụng tập trung vào việc lấy và hiển thị N thay đổi sau cùng đến trang web, và RowKey được chọn đã tối ưu hóa cho điều này.

## Cập nhật và đặt tên phiên bản

Mỗi ứng dụng thực hiện nhiều chiến lược ở đó giản đồ hiện tại của nó cần được nâng cấp để xử lí những yêu cầu mới.

Điều quan trọng cần chú ý là Windows Azure Table không lưu trữ một giản đồ, nó chỉ lưu trữ cặp <tên, typed-value> cho mỗi thuộc tính trong một thực thể. Do đó, bạn có thể có hai thực thể trong cùng một bảng có những thuộc tính khác nhau. Bạn cũng có thể có hai thực thể trong cùng một bảng với tên thuộc tính giống nhau, nhưng với kiểu khác nhau. Ví dụ, bạn có thể có một thực thể A với thuộc tính “Rating” có kiểu Int, và thực thể khác B với thuộc tính “Rating” với kiểu Double. Khi thực hiện truy vấn, máy chủ lờ đi những thực thể nếu kiểu thuộc tính trong điều kiện truy vấn không hợp kiểu thuộc tính cho thực thể này.

Sử dụng ví dụ ở trên, nếu chúng ta truy vấn “Rating > 1.2”, thì điều này sẽ đánh giá đối với thực thể B, nhưng thực thể A sẽ bị bỏ qua, bởi vì có kiểu không phù hợp.

Bây giờ khi nâng cấp một thuộc tính nó rất là cấp bách để ứng dụng xử lí cả hai phiên bản cho đến khi tất cả client và dữ liệu lưu trữ tồn tại được nâng cấp. Phần còn lại của mục này tập trung vào tính hữu ích của việc duy trì thuộc tính phiên bản cho mỗi thực thể , cho phép ứng dụng xử lí những kiểu nâng cấp khác nhau và làm thế nào sử dụng ADO.NET Service thực hiện hoàn thành việc nâng cấp.

Trong tất cả các chiến lược nâng cấp được định nghĩa bên dưới, có hai khả năng để áp dụng viêc nâng cấp:

1. Nâng cấp theo truy xuất – nghĩa là bất cứ khi nào một thực thể được lấy ra để cập nhật, thực hiện cập nhật nó thành phiên bản mới hơn.
2. Nâng cấp phía sau – việc nâng cấp được thực hiện ở phía sau sử dụng vai trò worker ở đó worker lấy tất cả các thực thể thuộc về phiên bản cũ hơn (sử dụng cột phiên bản) và nâng cấp nó.

Trong suốt quá trình đọc, một ứng dụng sẽ cần xử lí cả hai phiên bản. Trước khi chúng ta đi vào chi tiết việc thêm hoặc xóa thuộc tính, chúng ta đi qua thuộc tính của DataServiceContext quan trọng gọi là IgnoreMissingProperties. IgnoreMissingProperties điều khiển ADO.NET Data Service văng một exception khi lớp thực thể phía client không xác định một tính chất được trả về bởi máy chủ. Ví dụ: Nếu máy chủ trả về một thuộc tính gọi là “Rating” trên thực thể Blog, nhưng “Rating” không được định nghĩa trong định nghĩa lớp của client cho Blog, sau đó ADO.Net Data Service then ADO.NET Data Service throws an exception only if IgnoreMissingProperties is false.

// Set this to allow client definition of an entity to have certain properties missing because of either

// an addition/deletion of property on entities stored on the server.

context.IgnoreMissingProperties = true;

### Thêm thuộc tính mới

Khi một ứng dụng thêm một thuộc tính mới, nó sẽ cần định nghĩa thuộc tính trong định nghĩa thuộc tính. Bởi vì phiên bản cũ của ứng dụng có thể cũng đang chạy, nó là một ý tưởng hay khi luôn luôn gán IgnoreMissingProperties trong DataServiceContext thành true như ở trên. Điều này cho phép các client cũ hơn mà không có những thuộc tính mới vẫn tiếp tục đọc các thực thể này. Nếu không thiết lập giá trị này, ADO.Net Data Service sẽ văng exception khi một thực thể không xác định được thuộc tính trả về từ gói REST.

### Xóa kiểu thuộc tính

Chúng tôi sẽ giải thích việc xóa thuộc tính qua ví dụ, ở đây chúng ta xóa thuộc tính “Rating” từ thực thể blog.

1. Chúng ta tạo một định nghĩa mới cho thực thể Blog mà không có “Rating”

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

public class BlogV2

{

// <Channel name>\_<9999-YYYY>\_<13-Month>

public string PartitionKey { get; set; }

// To sort RowKey from newest to oldest blog,

// RowKey is DateTime.MaxValue.Ticks - DateTime.UtcNow.Ticks

public string RowKey { get; set; }

// User defined properties.

public string Text { get; set; }

public DateTime CreatedOn { get; set; }

// NOTE: removing this property so that it can be deleted from the

// storage too.

// public int Rating { get; set; }

}

1. Bây giờ, xóa thuộc tính, chúng ta sẽ lấy những thực thể blog từ máy chủ và cập nhật nó sử dụng định nghĩa thực thể mới:

// Ensure that an exception is not thrown since Rating is not defined in BlogV2 (as

// explained in section 8.4 above)

context.IgnoreMissingProperties = true;

// Retrieve blog entities into BlogV2.

// NOTE: Continuation tokens is not used here for brevity

var blogs =

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

select blog);

foreach(BlogV2 blog in blogs)

{

context.UpdateObject(blog);

}

1. Chúng ta sẽ gọi SaveChanges với SaveChangesOptions.ReplaceOnUpdate để những thuộc tính bị thiếu sẽ được xóa trên máy chủ.

// SaveChangesOptions.ReplaceOnUpdate will result in a "PUT" request being sent to the

// server as against "MERGE" request which will ensure that any missing property is deleted.

context.SaveChanges(SaveChangesOptions.ReplaceOnUpdate);

### Thay đổi kiểu thuộc tính

Khi kiểu thuộc tính cần được thay đổi cho một giản đồ, một lựa chọn là tạo một thuộc tính mới, có tên mới với kiểu mong muốn, sau đó sao chép dữ liệu từ thuộc tính cũ vào thuộc tính mới và cuối cùng xóa thuộc tính cũ. Để thực hiện điều này sử dụng kĩ thuật mô tả trong mục 8.4.1 và 8.4.2

# Tóm lược

Windows Azure Table cung cấp một nền tảng lưu trữ có cấu trúc. Nó hỗ trợ những bảng có khả năng mở rộng cực lớn trên đám mây. Hệ thống mở rộng các bảng một cách hiểu quả nhờ tự động cân bằng tải các phân vùng qua nhiều máy chủ khác nhau. Nó hỗ trợ một tập hợp các kiểu dữ liệu phong phú cho các thuộc tính và có thể truy xuất nhờ Dịch vụ Dữ liệu ADO.Net và REST. Nó hỗ trợ các đặc trưng như phân trang, mô hình optimistic concurrency và sự nối tiếp cho các truy vấn dài.

Phụ lục

**Etag**

Azure Table handles concurrent updates or deletes through optimistic concurrency using an [ETag](http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.19) value that is changed each time an entity is updated. The TableServiceContext stores the ETag of every entity it is tracking and submits this ETag in an If-Match request header when an update is requested. Azure Table rejects the update request if the submitted ETag does not match the current ETag for the entity. In the Storage Services REST API an update can be [forced](http://msdn.microsoft.com/en-us/library/dd179427.aspx) regardless of the current ETag by setting the ETag to the widcard character **\*** in the If-Match request header.

**Tranh chấp dữ liệu**

**Pessimistic concurrency**

Có 2 kiểu kiểm soát tranh chấp dữ liệu, cách thứ nhất (gọi là pessimistic concurrency) là khi em muốn cập nhật cái gì thì em phải khóa cái đó lại, phòng trường hợp chưa cập nhật xong mà người khác lại sửa, dẫn đến dữ liệu bị sai. Khi nào làm xong thì mở khóa ra, anh khác lúc đó mới sửa được dữ liệu.

**Optimistic Concurrency**  
Kiểu thứ hai là sau khi đọc dữ liệu xong, nếu em muốn cập nhật thì em phải so sánh dữ liệu lúc cập nhật có còn giống lúc trước em đọc vào hay không, nếu không có nghĩa đã có người khác sửa rồi, và em không được cập nhật nữa. Kiểu này được gọi là Optimistic Concurrency.