Cloud上の分散データベース

-- BigTable, SimpleDB, Azure SDS --

早稲田大学
丸山不二夫

はじめに

Cloudのエンタープライズ利用の焦点の一つは、Cloud上のデータベースをどのよ うに利用するのかということである。
企業内の 全てのシステムが一度にクラウドに移行するわけではないとしても、データベース部分を Cloudに依存するというソリューションは、Scalabilityの担保という点でも、コ スト削減という点でも、Cloud利用の一つの有力な選択肢たりうる。

はじめに

今回は、 Cloud上のデータベースとして先行している、GoogleのBigTable、AmazonのSimpleDB、MicrosoftのAzure SDSを取り上げる。
Cloud上のデータ ベースは、従来のリレーショナル・データベースとは異なっている部分が多い。こうした比較を通じて、それぞれの違いとともに、新しい共通の変化の方向を探りたい。

基本的な視点

- 分散データベース登場の背景
  - ネットワーク上の情報の爆発的増大
  - データベースのWeb-Scaleへの大容量化
- データベースへのScale-outの手法の導入
  - Scale-out
  - Availabilityの確保

基本的な視点

- 分散メモリーキャッシュの活用
  - 大規模化に伴う、処理の高速化への対応。
  - データ保持・操作の主要な舞台が、ファイルからメモリーに変わりつつある。
- データモデルの変化
  - リレーショナルからKey/Value型へ
- インターネットを通じたデータベースの利用へ
  - RESTの利用
  - On Premise と Cloud の共存

Cloud 上の分散データベース

- 2006年 Google: BigTable + Google App Engine
  http://labs.google.com/papers/bigtable-osdi06.pdf
- 2007年 Amazon: SimpleDB
  http://www.amazon.com/SimpleDB-AWS-Service-Pricing/b?ie=UTF8&node=342335011
  - Amazon: Dynamo
    http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html
- 2008年 Microsoft: SDS(SQL Data Service)
  http://www.microsoft.com/sql/dataservices/default.mspx

Agenda

- はじめに
- Cloudデータベース登場の背景
- Google BigTable (2006年)
- Amazon SimpleDB (2007年)
- Microsoft Azure SDS (2008年)
- まとめ

Cloudデータベース登場の背景

-- Google BigTableを例に

Bigtableとは何か？

- Bigtable は、数千台のサーバ上のペタバイト単位の非常に大きなサイズにまでスケールするようにデザインされた、構造化されたデータを管理する、分散ストレージ・システムである。
- Googleの多くのプロジェクトは、Bigtableにデータを格納している。Webのindexing, Google Earth, Google Finance等がそうである。これらのアプリケーションは、データのサイズに関しても、遅延に対する要求でも、全く異なる要求をBigtableに課している。

Bigtableの背景
Googleが持つ(半)構造化データ

- URL：コンテンツ、メタデータ、リンク、アンカー、ページランク、…
- 地理的位置情報：物理的なentity（店、レストランなど）、道路、衛星画像データ、ユーザによる注釈
- ユーザ毎のデータ：ユーザ設定情報、最近のクエリ/検索結果、…

Bigtableの背景
データ規模の巨大さ

- 数十億のURL、各ページに多くのバージョン（各バージョンは大きいもので20KB程度）
- 数億人のユーザ、毎秒数千回のクエリ
- 100TBを超える衛星画像データ

Bigtableの背景
データ規模の巨大さ

Project name	Table size (TB)	Compression ratio	# Cells (billions)	# Column Families	# Locality Groups
Crawl	800	11%	1000	16	8
Crawl	50	33%	200	2	2
Google Analytics	20	29%	10	1	1
Google Analytics	200	14%	80	1	1
Google Base	2	31%	10	29	3
Google Earth	0.5	64%	8	7	2
Google Earth	70	–	9	8	3
Orkut	9	–	0.9	8	5
Personalized Search	4	47%	6	93	11

BigTable Spec. Table2 p11

Bigtableの背景
なぜ商用DBを使用しないのか？

- たいていの商用DBにとって、データの規模が大きすぎる。
- たとえ扱えるとしても、コストが非常に高くなる。
- 内製化することで、低いコストで、多くのプロジェクトにシステムを適用できることを意味する。
- パフォーマンスの向上をDB層の上で実行するのは難しいが、低レベルのストレージの最適化によりパフォーマンスが著しく向上する。

大規模なシステムを作ることは、楽しくてやりがいのあることである Jeff Dean

Bigtableの目標

- 非同期のプロセスが、絶えずデータの異なった部分を更新すること。現在の大部分のデータへ、いつでもアクセスがあることが望ましい
- 非常に高いレートでの読み書き(1秒につき何百万オペレーションもの実行)
- 継続的なデータの変化について調べる:例えば、複数のクローラが訪問するウェブページのコンテンツ

BigTableの構成ブロック

- Google File System(GFS) ： rawストレージ
- Scheduler ： マシンにジョブをスケジュールする
- Lock Service ： 分散ロック管理
  同時に(100 バイト程度の)小さなファイルを確実にロックする
- MapReduce ： 簡単にされた大規模データ処理

Google BigTable

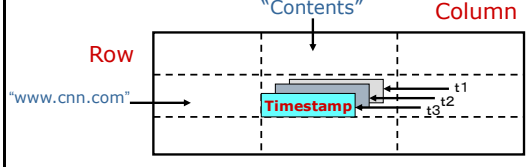
- データモデル
- Scale-outの手法
- 実装の特徴
- Google App Engine

BigTable
データモデル

- 分散多次元疎マップ
- Row
- Column
- Time Stamp

BigTableのデータモデル

- 分散多次元疎マップ
  (row,colum,timestamp) --> cell



- Googleの大部分のアプリケーションに適合する

BigTableのデータモデル
Row

- 名前は、任意の文字
- 1行のデータへのアクセスは、atomicである
- 行は、データを格納する際に、自動的に作成される
- 行は辞書式順序で並べられている
- 通常、行は1台または数台のマシン上に、辞書式順序の近い順番にまとめられる

BigTableのデータモデル
Column

- Columnは二つのレベルの名前の構造を持つ。
  family:optional_Qualifier
- Colum family:
  - アクセス・コントロールの単位
  - 型の情報に関連付けられている
- Qualifierは、大きさの限界のないカラムを可能にする。

BigTableのデータモデル
Timestamp

- セル中に、異なったバージョンのデータを格納するのに利用される。新しいwriteのデフォールトは、現在の時刻。
- 検索用のオプション
  - 最も最近のK個の値を返す
  - 全てのtimestamp範囲の値を返す
- カラム・ファミリーは属性を持つ。
  - 最も最近のK個の値を保持せよ
  - K秒より古くなるまで、値を保持せよ。

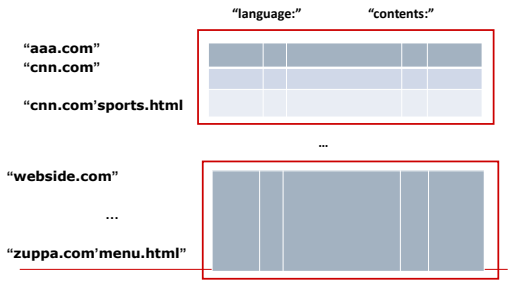
BigTable
Scale-outの手法

- Tablet
- Tabletの分割
- Tabletの位置指定
- Tablet Server

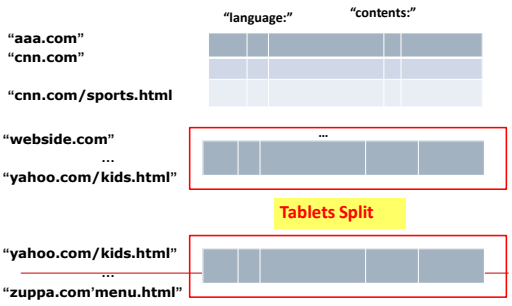
Tablets

- 大きなテーブルは、行の境界で、タブレットに分割される
- タブレットは行の連続する範囲を維持する
- クライアントは、局所性を達成するために、行のキーを選択できる
- 1つのタブレットあたりの100MB~200MBが目処

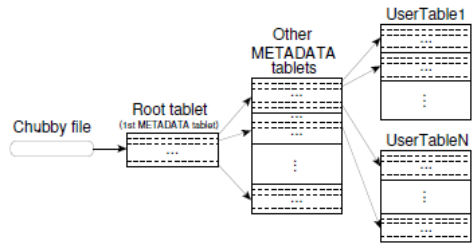
Tablets
Tablets の分割



Tablets
Tabletsの分割



Tablets
Tabletの位置指定



Tablets
Tabletの位置指定

- タブレットはサーバからサーバへ移動するので、与えられた行に対して、クライアントはどのようにして正しいマシンをみつけるのだろうか？
- ひとつの方法：BigTable マスターを使用する。大きなシステムでは、ほとんど間違いなく、セントラルサーバはボトルネックになるだろう
- 他の方法：タブレットのロケーション情報を含む特別なテーブルを、BigTable のセルそれ自身の中に格納する。

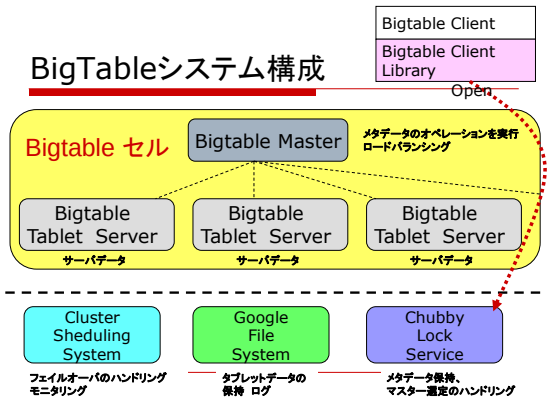
Tablets
Tabletの位置指定

- Tabletの位置に対する、3段階の階層的な検索スキーマ（位置とは、サーバのIPとPortのこと）
- 第1段階：ロックサービスから起動されて、META0のオーナーをポイント
- 第2段階：META0のデータを使用し、適切なMETA1タブレットを検索
- 第3段階：META1テーブルが、その他全てのテーブルの位置を保持
- META1テーブルは、それ自身が複数のテーブルに分割可能

Tablets
指定できるデータ量

- それぞれのMETADATAの行は、約1KBのデータをメモリー上に持つ。
- 普通の大きさの128MBのMETADATAのtabletは、この3段階の位置指定の方式では、2^34個のtabletの指定が可能である。
- これは、128MBのtabletなら、2^61バイトものデータ量になる。

BigTableシステム構成



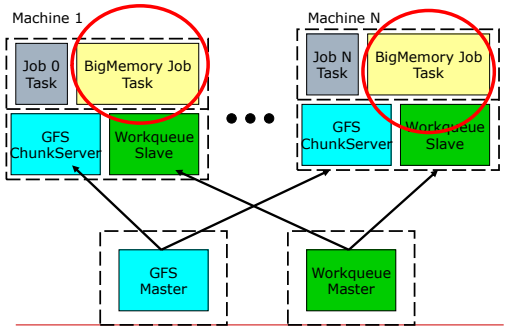
Tablets
Tablets Server

- 100程度までのタブレットに責任を持つようにマシンを立てる
- 早いリカバリ
- 100台のマシンが、故障したマシンから一つのタブレットをピックアップすればよい
- きめ細かいロードバランシング
- 過負荷のマシンから、タブレットを移す

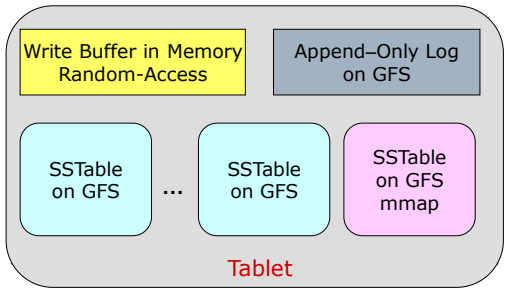
BigTable
実装の特徴

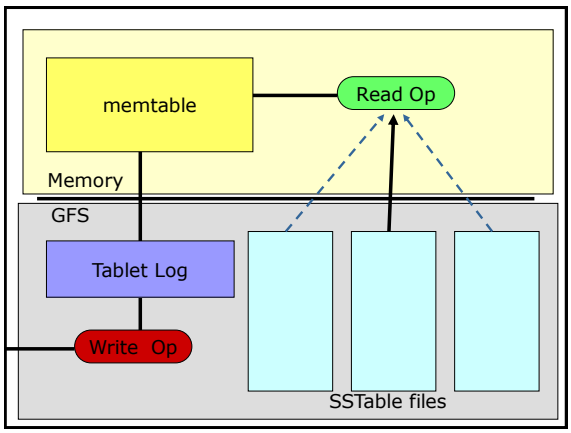
- Tabletの表現とSSTable
- Tablet Serving
- データ圧縮
- Locality Group
- Shared Log

Google Basic Computing Cluster



Tabletの表現 / SSTable





Tablet Serving
write operation

- 書き込みの操作がtabletサーバに届くと、サーバは、そのリクエストの形式と権限をチェックする。
- 権限チェックは、Chubbyファイルの権限を持つ書き手のリストを読むことで行われる。
- 正当な書き込みは、コミットlogに書き込まれる。
- 書き込みがコミットされてから、その中身がmemtableに挿入される。

Tablet Serving
Read Operation

- 読み出しの要求がtabletサーバに届くと、同様に、形式と権限がチェックされる。
- チェックされた正当な読み出しは、SSTableの並びとmemtableのマージしたviewの中で実行される。
- SSTableもmemtableも、辞書式順にソートされたデータ構造であるので、マージは効率的に行われる。

データ圧縮
memtableからSSTableへの変換

- 書き込みが実行されると、memtableのサイズが増大する。
- それが、ある値を超えると、memtableは凍結され、新しいmemtableが生成される。
- 凍結されたmemtableはSSTableに変換され、GFSに書き出される。
- この「小圧縮」は、tabletサーバのメモリーの利用量を減らし、サーバが死んだ場合のふっリカバリの際に、読み込むlogの量を少なくする。

データ圧縮
merging compaction

- 小圧縮の度に、新しいSSTableが生成される。こうしたことが続けば、readの要求の度に、沢山のSSTableのmerge操作が必要になる。
- そうしたことを避ける為に、SSTableの数に制限を設けて、バックグラウンドで定期的に、merging compactionを実行する。
- これは、いくつかのSSTableとmemtableの中身を読み込んで、新しいSSTableを書き出す。

データ圧縮
major compaction

- 全てのSSTableを、一つのSSTableに書き換えるmerging compactionを、「大圧縮（major compaction）」という。
- 大圧縮以外の圧縮では、元のSSTableに含まれているdeleteのマークがついたデータは、そのまま移されるが、大圧縮のときには、こうしたデータは、消去される。

BigTable
Google App Engine

2008年4月　ベータ版

……

2008年11月　1.1.7

Google App Engine
webapp

```
import cgi
import wsgiref.handler
from google.appengine.api import users
from google.appengine.ext import webapp

class MainPage(webapp.RequestHandler):
  def get(self):
    self.response.out.write("""
      <html>
        <body>
          <form action="/sign" method="post">
            <div><textarea name="content" rows="3" cols="60">
                    </textarea></div>
            <div><input type="submit" value="Sign Guestbook"></div>
          </form>
        </body>
      </html>""")
```

Google App Engine
webapp

```
class Guestbook(webapp.RequestHandler):
    def post(self):
      self.response.out.write('<html><body>You wrote:<pre>')
      self.response.out.write(cgi.escape(self.request.get('content')))
      self.response.out.write('</pre></body></html>')

  def main():
    application = webapp.WSGIApplication(
                                  [('/', MainPage),
                                   ('/sign', Guestbook)],
                                  debug=True)
    wsgiref.handlers.CGIHandler().run(application)

  if __name__ == "__main__":
    main()
```

Google App Engine
データモデル

- App Engineは、スケーラブルなdatastore(**データストア**)で、*entities (エンティティ)* というデータ・オブジェクトを格納・検索する。
- **エンティティ**は、一つ以上の*properties (プロパティ)* を持つ。プロパティは、**名前**と、サポートされているいくつかの型の**値**を持つ。
- **データストア**のAPIは、**データモデル**を定義するメカニズムを特徴づける。**モデルは、エンティティの種類**(kind)**を記述する。**モデルには、型と、そのプロパティの設定が含まれている。
- アプリケーションは、Pythonのクラスを使ってモデルを定義し、クラスの属性(attribute)でプロパティを記述する。

Google App Engine
Pythonクラスによるテーブルの定義

```
from google.appengine.ext import db

class Greeting(db.Model):
  author = db.UserProperty()
  content = db.StringProperty(multiline=True)
  date = db.DateTimeProperty(auto_now_add=True)
```

```
class Guestbook(webapp.RequestHandler):
  def post(self):
    greeting = Greeting()

    if users.get_current_user():
      greeting.author = users.get_current_user()

    greeting.content = self.request.get('content')
    greeting.put()
    self.redirect('/')
```

```
class MainPage(webapp.RequestHandler):
  def get(self):
    self.response.out.write('<html><body>')

    greetings = db.GqlQuery(
      "SELECT * FROM Greeting ORDER BY date DESC LIMIT 10")

    for greeting in greetings:
      if greeting.author:
        self.response.out.write('<b>%s</b> wrote:' %
                            greeting.author.nickname())
      else:
        self.response.out.write('An anonymous person wrote:')
      self.response.out.write('<blockquote>%s</blockquote>' %
                          cgi.escape(greeting.content))
```

```
# Write the submission form and the footer of the page
    self.response.out.write("""
        <form action="/sign" method="post">
          <div><textarea name="content" rows="3" cols="60">
              </textarea></div>
          <div><input type="submit" value="Sign Guestbook"></div>
        </form>
      </body>
    </html>""")
```

Google App Engine
GQL

```
if users.get_current_user():
    user_pets = db.GqlQuery("SELECT *
  FROM Pet WHERE pet.owner = :1",
              users.get_current_user())
    for pet in user_pets:
      pet.spayed_or_neutered = True

    db.put(user_pets)
```

Google App Engine
GQL

```
# Parameters can be bound with positional arguments.
    query = db.GqlQuery("SELECT * FROM Story WHERE
    title = :1 "
                "AND ANCESTOR IS :2 "
                "ORDER BY date DESC",
                'Foo', key)

# Or, parameters can be bound with keyword arguments.
    query = db.GqlQuery("SELECT * FROM Story WHERE
    title = :title "
                "AND ANCESTOR IS :parent "
                "ORDER BY date DESC",
                title='Foo', parent=key)
```

Google App Engine
Memcache API

Google App Engineでは、ハイパフォーマンスでスケーラブルなwebアプリケーションは、パーシステントなストレージのキャッシュとして、あるいは、その代わりとして、分散メモリーキャッシュを利用できる。

Google App Engine
Memcache API

```
def get_data():
    data = memcache.get("key")
    if data is not None:
      return data
    else:
      data = self.query_for_data()
      memcache.add("key", data, 60)
      return data
```

Amazon SimpleDB

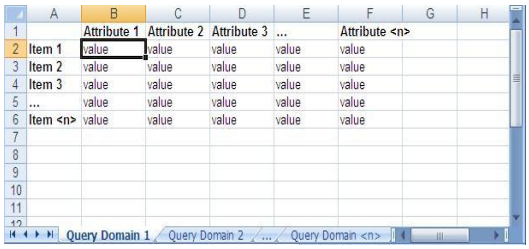
- データモデル
- プログラミングスタイルとRESTの利用
- Scalability
- Amazon Dynamo

SimpleDB
データモデル

SimpleDB
データモデル

- CREATE a new ***domain*** to house your unique set of structured data.
- GET, PUT or DELETE ***items*** in your domain, along with the attribute-value pairs that you associate with each item.
- QUERY your data set using simple set of operators

SimpleDB
データモデル



SimpleDB
データモデル

ID	Category	Subcat.	Name	Color	Size	Make	Model
Item_01	Clothes	Sweater	Cathair Sweater	Siamese	Small, Medium, Large		
Item_02	Clothes	Pants	Designer Jeans	Paisley Acid Wash	30x32, 32x32, 32x34		
Item_03	Clothes	Pants	Sweatpants	Blue, Yellow, Pink	Large		
Item_04	Car Parts	Engine	Turbos			Audi	S4
Item_05	Car Parts	Emissions	02 Sensor			Audi	S4
Item_06	Motorcycle Parts	Bodywork	Fender Eliminator	Blue		Yamaha	R1
Item_07	Motorcycle Parts, Clothing	Clothing	Leather Pants	Black	Small, Medium, Large		

SimpleDB
プログラミングスタイル

RESTベースの多言語サポート

- PHP
- C#
- Java
- Perl
- VB.NET

SimpleDB
REST PutAttributes Request

```
https://sdb.amazonaws.com/?Action=PutAttributes
&DomainName=MyDomain
&ItemName=Item123
&Attribute.1.Name=Color
&Attribute.1.Value=Blue
&Attribute.2.Name=Size
&Attribute.2.Value=Med
&Attribute.3.Name=Price
&Attribute.3.Value=0014.99
&Version=2007-11-07
&Timestamp=2007-06-25T15%3A01%3A28-07%3A00
&SignatureVersion=2
&SignatureMethod=HmacSHA256
&AWSAccessKeyId=<Your AWS Access Key ID>
```

SimpleDB
REST Responce

```
<PutAttributesResponse
   xmlns="http://sdb.amazonaws.com/doc/2007-11-07">
   <ResponseMetadata>
      <StatusCode>Success</StatusCode>
      <RequestId>
        f6820318-9658-4a9d-89f8-b067c90904fc
      </RequestId>
      <BoxUsage>0.0000219907</BoxUsage>
   </ResponseMetadata>
</PutAttributesResponse>
```

SimpleDB
ItemのAttributesの取得

```
GetAttributes request = new GetAttributes();
String domainName = "MyStore";
String itemName = "Item_03";
request.setDomainName(domainName);
request.setItemName(itemName);
GetAttributesResponse response =
    service.getAttributes(request);
if (response.isSetGetAttributesResult()) {
    GetAttributesResult  getAttributesResult =
        response.getGetAttributesResult();
    java.util.List<Attribute> attributesList =
        getAttributesResult.getAttribute();
    .........
```

```
String itemNameOne = "Item_01";
java.util.List<ReplaceableAttribute> attributeListOne =
  new ArrayList<ReplaceableAttribute>(7);
attributeListOne.add(
  new ReplaceableAttribute("Category", "Clothes", false));
attributeListOne.add(
  new ReplaceableAttribute("Subcategory", "Sweater", false));
attributeListOne.add(
  new ReplaceableAttribute("Name", "Cathair Sweater", false));
attributeListOne.add(
  new ReplaceableAttribute("Color", "Siamese", false));
attributeListOne.add(
  new ReplaceableAttribute("Size", "Small", false));
attributeListOne.add(
  new ReplaceableAttribute("Size", "Medium", false));
attributeListOne.add(
  new ReplaceableAttribute("Size", "Large", false));
PutAttributes putAttributesActionOne =
  new PutAttributes(domainName, itemNameOne, attributeListOne);
invokePutAttributes(service, putAttributesActionOne);
```

Muliti Valueの設定

SimpleDB
DomainからのItemの取得

```
Query request = new Query();
String queryExpression = "['Category' = 'Clothes']";
request.withDomainName("MyStore").
  withQueryExpression(queryExpression);
QueryResponse response = service.query(request);
if (response.isSetQueryResult()) {
  QueryResult  queryResult = response.getQueryResult();
  java.util.List<String> itemNamesList  =
    queryResult.getItemName();
    .....
}
```

SimpleDB
Query Expression

- [‘city’ = ‘Seattle’ or ‘city’ = ‘Portland]
- [‘weight’ >= ‘065’]
- [‘year’ < ‘2000’ and ‘year’ > ‘1995’]
- [‘first name’ = ‘John’] intersection [‘lastname’ = ‘Smith’]
- [‘tag’ starts-with ‘Amazon’] union [‘description’ = ‘SimpleDB’]
- not [‘country’ = ‘USA’ or ‘country’ = ‘UK’]

SimpleDB
SELECT

select *output_list*
from *domain_name*
[where *select_expression*]
[*sort_instructions*]
[limit *limit*]

```
select `Invisible`` Pink`` Unicorn`` Sightings`
from `My Domain`
where location = 'O''Brien, TX' and
  `timestamp-1` > '' order by `timestamp-1`
```

SimpleDB
SELECT

```
AmazonSimpleDBConfig config = new AmazonSimpleDBConfig();
config.setSignatureVersion("0");
AmazonSimpleDB service =
    new AmazonSimpleDBClient(accessKeyId,
                             secretAccessKey, config);
SelectRequest request = new SelectRequest();

//request.setSelectExpression("select * from MyStore");
request.setSelectExpression(
    "select * from MyStore where Size = 'Large`'");
invokeSelect(service, request);
```

SimpleDB
SELECT

```
AmazonSimpleDBConfig config = new AmazonSimpleDBConfig();
config.setSignatureVersion("0");
AmazonSimpleDB service =
    new AmazonSimpleDBClient(accessKeyId,
                             secretAccessKey, config);
SelectRequest request = new SelectRequest();

//request.setSelectExpression("select * from MyStore");
request.setSelectExpression(
    "select * from MyStore where Size = 'Large'");
invokeSelect(service, request);
```

SimpleDB
Scalability

SimpleDB
データセットの分割

- 例えば、次のようにして、データセットを4つの異なるDomainに分割することができる。
- MD5のようなハッシュ関数を使って、itemのハッシュ値を計算する。
- 次のように、itemの担当Domainを決める。
  - 最後の2bitが 00だったら、そのitemをDomain0に置く
  - 最後の2bitが 01だったら、そのitemをDomain1に置く
  - 最後の2bitが 02だったら、そのitemをDomain2に置く
  - 最後の2bitが 03だったら、そのitemをDomain3に置く

Amazon Dynamo

Dynamo: Amazon's Highly Available Key-value Store

SOSP'07, October 14–17, 2007

http://s3.amazonaws.com/AllThingsDistributed/sosp/amazon-dynamo-sosp2007.pdf

Amazon Dynamo
Abstract

巨大なスケールのシステムの信頼性は、世界で最大規模のe-コマースの取引を行っている、我々Amazon社が直面している、最も大きな挑戦の一つである。ほんの軽微な欠陥でも、重大な経済的な結果をもたらし、利用者の信頼に大きな影響を与えかねない。Amazon社のプラットフォームは、世界中の多くのwebサイトにサービスを提供しているのだが、世界中の多数のデータセンターに設置された、数万のサーバ、ネットワーク機器の上で実装されている。この規模では、大小の機器がいつも連続して故障しており、これらのハード故障のただなかで、永続する状態を管理するやり方は、ソフトウェアのシステムの信頼性とスケーラビリティに対する要求に拍車をかけている。

Amazon Dynamo
Abstract

この論文は、Amazon社のいくつかのコアなサービスが、"always-on"の経験を提供するために利用してきた可用性の高いkey-valueストレージシステムである、Dynamoのデザインと実装について述べたものである。

このレベルの可用性を達成するために、Dynamoは、ある種のシステムエラーのシナリオでは、整合性を犠牲にしている。それは、開発者が利用できる新しいインターフェースを提供するというやりかたで、オブジェクトのバージョニングとアプリケーションによって支援された競合の解決策とを、広い範囲にわたって利用している。

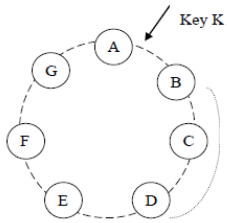
Amazon Dynamo
基本的な要請

- データをいつでも書き込み可能であること。
  - エラーや並行する書き込みのロックで、書き込みを拒否されることはない
- 単一管理者ドメイン。
  - ノードは、信頼できること。
- 階層的な名前空間や、複雑なRelational Schemaをサポートしない。
- 遅延を許さないアプリへの対応。
  - 数百ミリ秒のうちに、99.9%の読み書きが終了すること

Amazon Dynamo
Zero-hop DHT

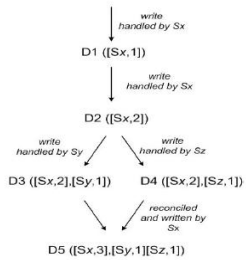
- 我々は、ChordやPastryのようなDHTシステムで取り入れられている、典型的なデザイン、複数のノードをルーティングするといったデザインを取らなかった。
- Multi-hopのルーティングは、応答時間の変異を増大させ、結果として、高い頻度で遅延を増大させる。
- Dynamoは、zero-hop DHTと特徴づけることができる。それぞれのノードは、他のノードへのルーティング情報を抱えている。

Amazon Dynamo
Consistent Hashing



ノードB,C,Dが、Kを含む、範囲(A,B]のキーを格納する。

Amazon Dynamo
Data Versioning



それぞれのデータは、[Node,Counter]のペアの配列を、バージョンとして抱えている。
書き込みの前に、バージョンは読み取られ、書き込みが成功すると、Counterはインクリメントされる。
同じデータで、バージョンの異なるものが、一時的には存在しうる。

Amazon Dynamo
Sloppy Quorum

- システムが、読み出しが成功したと認めるノードの数をR、書き出しが成功したと認めるノードの数をW、レプリカの数をNとしよう。
- Dynamoでは、読み書きによる遅延を少なくするために、RもWも、Nより小さい値に設定している。
- 通常は、読み書きの候補ノードリストの最初のN個が、quorumのメンバになるのだが、Dynamoでは、最初のN個ではなく、候補リストのダウンしていないN個のメンバに対して、読み書きを行う。

Amazon Dynamo
Storageノードの実装

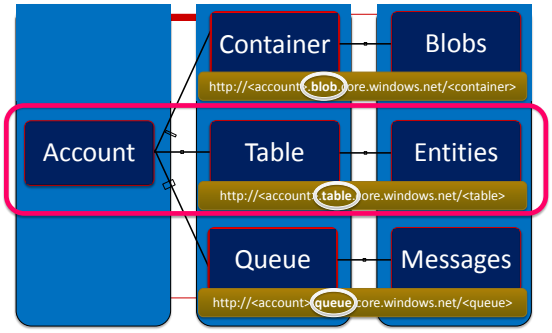
- Dynamoのstorageノードは、次の三つのコンポーネントから成り、すべてJavaで実装されている。
- Local Persistent Engine
  - Berkeley Database (BDB) Transactional Data Store2, BDB Java Edition, MySQL等
- Membership and Failure Detection
- Request Coordination

Microsoft Azure SDS

- データモデル
- Scale-outとAvailability
- ScalabilityへのP2P/DHTの利用
- プログラミングモデル　REST編
- プログラミングモデル　C#編

Azure SDS
データモデル

Windows Azure
Data Storage （Blob, Table, Queue）



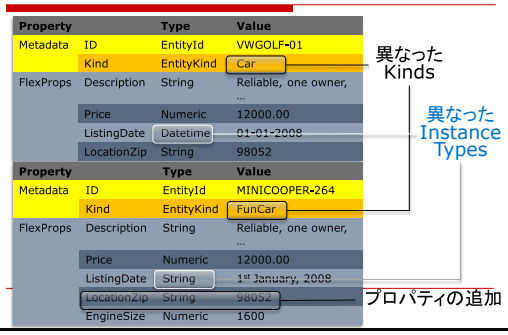
Azure SDS
データモデル

- テーブル
  - ストレージのAccountがあれば、複数のテーブルを生成できる
  - テーブル名は、Accountで範囲づけられる
- データはテーブルに蓄えられる
- テーブルはエンティティ（行）の集合
- エンティティはプロパティ（カラム）の集合

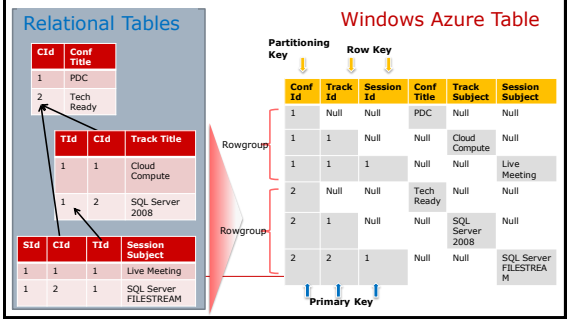
エンティティのプロパティの型は、エンティティごとに違っていても構わない

Azure SDS
データモデル

SDSのエンティティの特徴



Azure SDS
Relational DBのテーブルとの違い



Azure SDS
エンティティとプロパティ

- それぞれのEntityは、255個までのプロパティを持つ
- 全てのエンティティにとって必須のプロパティ
  - Partition Key
  - Row Key
- 全てのエンティティは、システムが管理するversionをプロパティとして持つ

Azure SDS
エンティティとプロパティ

- その他のプロパティについては、決まったスキーマはない
- テーブルのスキーマ情報は、存在しない

- それぞれのプロパティは、 <名前, 型を持つ値> のペアとして蓄えられる
- 同じテーブル内の二つのエンティティは、違ったプロパティを持つことができる

Azure SDS
サポートされているプロパティの型

- Partition Key と Row Key
  - String (64KBまで)
- プロパティの型
  - String (64KBまで)
  - Binary (64KBまで)
  - Bool
  - DateTime
  - GUID
  - Int
  - Int64
  - Double

Azure SDS
Partition Key

Azure SDS
Partition Key とPartitions

- 全てのテーブルは、Partition Keyを持つ
  - テーブルの第一カラム
  - テーブルをPartitionに分割するのに利用される
- テーブルPartition
  - あるテーブル内で、同じPartitionKeyの値を持つ全てのエンティティ
- Partition Keyは、プログラムから操作可能
  - アプリケーションが、Partitionの粒度をコントロールして、スケーラビリティを持つことを可能とする

SDS Azure: Partition の例

Partition Key Document Name	Row Key Version	Property 3 Modification Time	.....	Property N Description
Examples Doc	V1.0	8/2/2007	.....	Committed version
Examples Doc	V2.0.1	9/28/2007		Alice's working version
FAQ Doc	V1.0	5/2/2007		Committed version
FAQ Doc	V1.0.1	7/6/2007		Alice's working version
FAQ Doc	V1.0.2	8/1/2007		Sally's working version

Partition 1
Partition 2

Partition： あるテーブル内で、同じ PartitionKeyの値を持つ全てのエンティティ
アプリケーションは、Partitionの粒度をコントロールできる

SDS Azure
Partitionの目的

- パフォーマンスとエンティティの局所性
  - 同じPartition内のエンティティは、一緒に格納される
    - 効率的な検索とキャッシュの局所性
- テーブルのスケーラビリティ
  - Partitionの利用パターンをモニタし、
  - 自動的にPartitionのロードバランスを行う
    - それぞれのPartitionは、異なるストレージノードに格納されることができる
    - テーブルのトラフィックの必要に見合うようにスケールできる

SDS Azure: パフォーマンスとスケーラビリティ

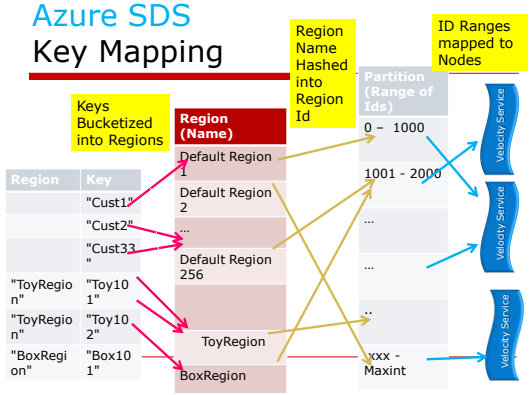
Partition Key Document Name	Row Key Version	Property 3 Modification Time	.....	Property N Description
Examples Doc	V1.0	8/2/2007	.....	Committed version
Examples Doc	V2.0.1	9/28/2007		Alice's working version
FAQ Doc	V1.0	5/2/2007		Committed version
FAQ Doc	V1.0.1	7/6/2007		Alice's working version
FAQ Doc	V1.0.2	8/1/2007		Sally's working version

Partition 1

Partition 2

- FAQ DOCの全てのVersionを取得するのは効率的に出来る。なぜなら、一つのPartitionにアクセスすればよいから
- アクセスをスケールアウトするために、二つのPartitionを、異なるサーバに置くことができる

Azure SDS
Key Mapping

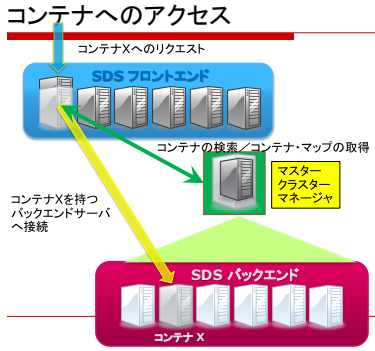


Azure SDS
Partition Keyをどう選ぶか

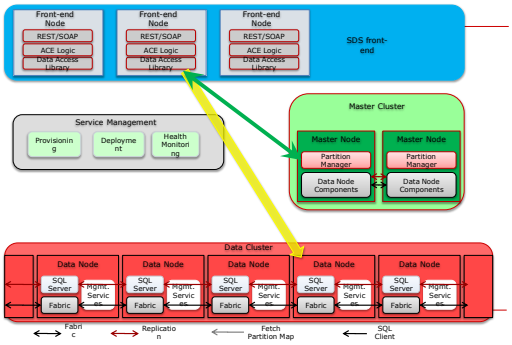
- 検索でよくつかわれるものをPartition Keyとして使う
  - Partition Key が Queryの一部であれば
    - 一つのPartition内のエンティティを取得するのに、高速にアクセスできる
  - もし、Partition KeyがQueryで指定されないと
    - その時には、すべてのPartitionがスキャンされなければいけない

Azure SDS
Scale-outとAvailability

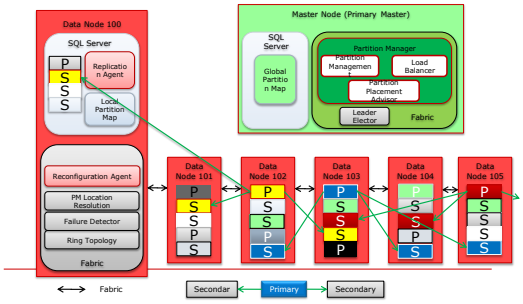
Azure SDS
コンテナへのアクセス



Azure SDS
コンテナへのアクセス



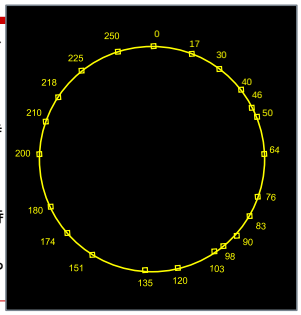
SDS Azure
Availability　PrimaryとSecondary



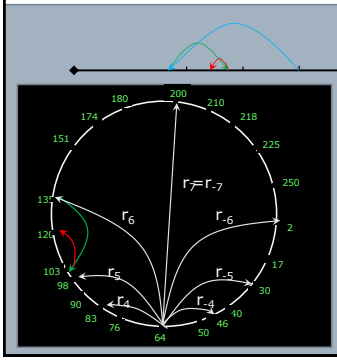
Azure SDS
ScalabilityへのP2P/DHTの利用

Azure SDS
リング構造

- 全てのノードはユニークなIDを与えられる。(典型的には、128-bit か160-bitの数値である)
- アクティブなノードは、順序を持つ双方向リンクで、しっかり結び付けられ、自分たちで、その形を維持する。
- 最高位のIDと最低位のIDを持つノードは、相互にリンクする
- こうして、アクティブなノードたちはリングの形をとるようになる。
- リングは、最初の一つのノードからブートストラップされる。



Azure SDS
リング構造／DHT

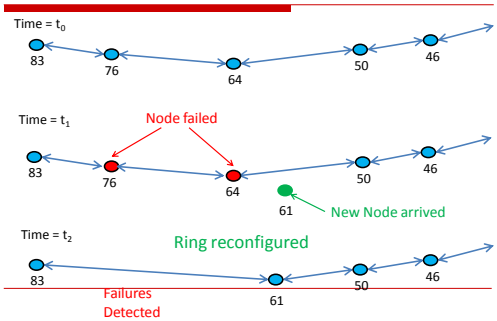


ノード 64のルーティングテーブル:
Successor = 76
Predecessor = 50
Neighborhood = (83, 76, 50, 46)
Routing nodes = (200, 2, 30, 46, 50, 64, 64, 64, 64, 64, 83, 98, 135, 200)

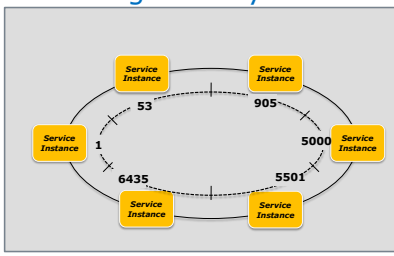
r_{±n}=a±2^n

ルーティングは、分散ハッシュテーブル(DHT)を作る基礎となる。

Azure SDS
Failure Detection



Routing Overlay - Partitioning

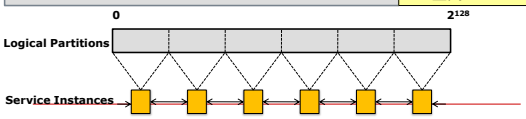


構造化された
P2Pネットワーク

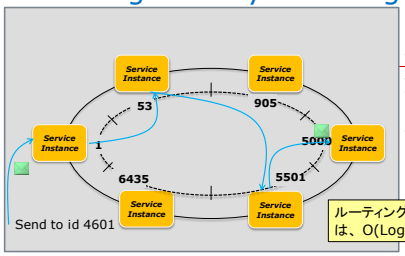
リング・トポロジー

動的で特定の中心を持たないメンバーシップ構造

連続した仮想的なアドレス空間



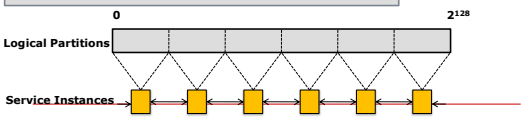
Routing Overlay - Messaging



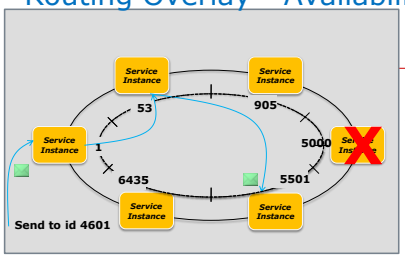
WCF トランスポート
チャンネルを使う
WCF TCP チャンネル

アドレスは、partition IDである

ルーティングのホップ数は、O(Log(n))



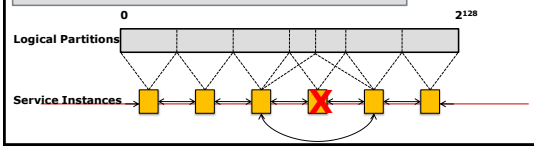
Routing Overlay - Availability



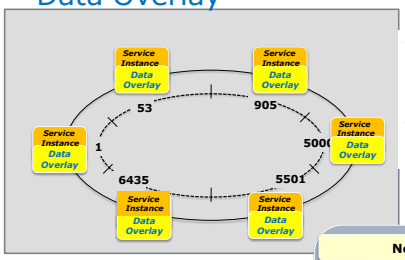
自己モニターリングと
自己修復機能

柔軟なホスト環境

Partitionsは、常にアドレス可能である



Data Overlay



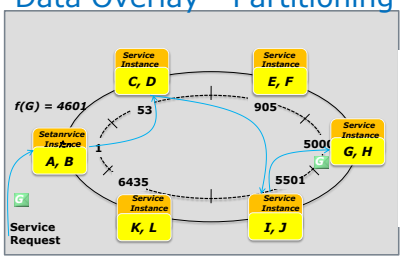
データ・オーバーレーは、ルーティング・オーバーレーの上に作られる

エンティティは、Partitionに格納される

Node 5000

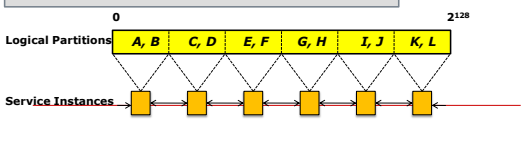
Partition Id	Name	Contents
4601	WF #12	<Message Q for #12>
2953	WF #67	<Message Q for #67>
...		...

Data Overlay - Partitioning

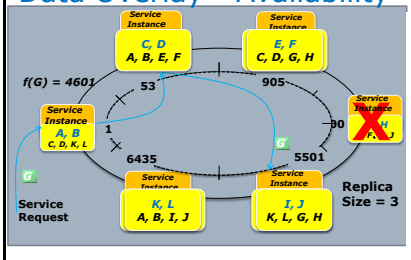


サービスは、エンティティからPartitionへのマッピングを持っている

Data Overlay の操作は、ノード上でローカルに行われる



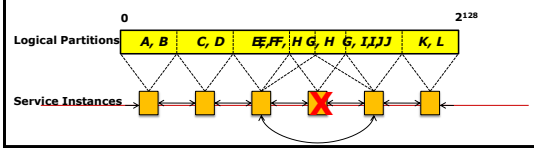
Data Overlay - Availability



可用性を高めるためにノードに複数のレプリカを置く

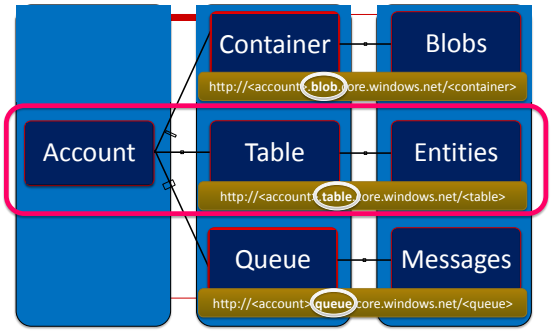
プライマリが壊れたら、自動的にセカンダリへのフェールオーバが行われる

可用性と無矛盾性は、システムデザイン時の指針になっている



Azure SDS
プログラミング・モデル　REST編

Windows Azure のData Storage（Blob, Table, Queue）



Azure SDS
REST のスタイル

URL

- http://<Account>.table.core.windows.net/<Container-ID>
- http://<Account>.table.core.windows.net/<Container-ID>/<Entity-ID>

HTTP Verb	SDS での操作
POST	生成
GET	検索、データ取得
POST	更新
DELETE	削除

Azure SDS
Authority(Account)の生成

URL

- https://table.core.windows.net/

HTTP　Verb：POST

Payload：

```
<s:Authority
   xmlns:s='http://schemas.microsoft.com
   /sitka/2008/03/'>
   <s:Id>coho</s:Id>
</s:Authority>
```

Azure SDS
Authority(Account) データの取得

URL

- https://coho.table.core.windows.net/

HTTP　Verb：GET

Payload：

```
<s:Authority
    xmlns:s="http://schemas.microsoft.com/sitka/2008/03/
    " xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance"
    xmlns:x="http://www.w3.org/2001/XMLSchema">
    <s:Id>coho</s:Id>
    <s:Version>3593083</s:Version>
</s:Authority>
```

Azure SDS
Container(Table)の生成

URL

- https://coho.table.core.windows.net/

HTTP　Verb：POST

Payload：

```
<s:Container
   xmlns:s='http://schemas.microsoft.com
   /sitka/2008/03/'>
   <s:Id>Wines</s:Id>
</s:Container>
```

Azure SDS
Entityの生成（続く）

URL

- https://coho.table.core.windows.net/Wines/

HTTP Verb：POST

Data Type：string, Base64Binary, Boolean, Decimal, DateTime

Meta Data Property	目的
ID	Entityの一意な識別子
Version	システムの生成するVersion番号条件付き更新、同期に使う
Kind	それぞれのEntityに任意の型を指定する

Azure SDS
Entityの生成

Payload：

```
<RedWine
    xmlns:s='http://schemas.microsoft.com/sitka/2008/03/'
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xmlns:x='http://www.w3.org/2001/XMLSchema' >
    <s:Id>CohoSauvignon05</s:Id>
    <vineyard xsi:type='x:string'>Coho Vineyard</vineyard>
    <vintage xsi:type='x:string'>2005</vintage>
    <grape xsi:type='x:string'>Cabernet Sauvignon</grape>
    <region xsi:type='x:string'>New Zealand</region>
    <description xsi:type='x:string'>Full-bodied and
    fruity</description>
    <thumbnail xsi:type='x:base64Binary'>/9j/4AAQSkZJ…
    </thumbnail>
    <price xsi:type='x:decimal'>16.95</price>
</RedWine>
```

Azure SDS
Entityデータの取得（続く）

URL

- https://coho.table.core.windows.net/wines/CohoSauvignon05

HTTP Verb：GET

Azure SDS
Entityデータの取得

Payload：

```
<RedWine
    xmlns:s='http://schemas.microsoft.com/sitka/2008/03/'
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xmlns:x='http://www.w3.org/2001/XMLSchema' >
    <s:Id>CohoSauvignon05</s:Id>
    <s:Version>210441</s:Version>
    <vineyard xsi:type='x:string'>Coho Vineyard</vineyard>
    <vintage xsi:type='x:string'>2005</vintage>
    <grape xsi:type='x:string'>Cabernet Sauvignon</grape>
    <region xsi:type='x:string'>New Zealand</region>
    <description xsi:type='x:string'>Full-bodied and
    fruity</description>
    <thumbnail xsi:type='x:base64Binary'>/9j/4AAQSkZJ…
    </thumbnail>
    <price xsi:type='x:decimal'>16.95</price>
</RedWine>
```

EntitySetを取得する Container(Table)へのQuery

https://coho.table.core.windows.net/wines/?='from e in entities …'

HTTP Verb：GET

```
<s:EntitySet
    xmlns:s="http://schemas.microsoft.com/sitka/2008/03/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:x="http://www.w3.org/2001/XMLSchema">
    <RedWine>
        …
    </RedWine>
    <RedWine>
        …
    </RedWine>
</s:EntitySet>
```

Azure SDS
Entityデータの更新

- GET Request を https://coho.table.core.windows.net/wines/cohosauvignon05に送る。
- 返ってきたXML payloadのprice elementの値を変更する。
- 変更したXML payloadをPUTで、https://coho.table.core.windows.net/wines/cohosauvignon05に送る。
- ResponseのStatusコードを調べて、変更が正しく行われたかをチェックする。

Azure SDS
データの削除

- Entity cohosauvignon05の消去
  Delete Request を
  https://coho.table.core.windows.net/wines/cohosauvignon05に送る。

- Container(Table) wineの消去
  Delete Requestを
  https://coho.table.core.windows.net/winesに送る。

Azure SDS
A REST Interface For Data

Entity Data Model	• 基本のモデル • エンティティとその関係は、リソースとリンクとして表現される
URIs	• リソースは、すべて、URIを通じてアクセスされる • データ取得にとって柔軟なURIスキーム
HTTP	• リソースとしてのデータ。HTTPのVerbがそれを操作する。 • AtomPubのフォーマット • Leverage caching, proxies, authentication, …
Formats	• AtomPub, JSON

Azure SDS
REST: URLでのEntity等の指定

Entity-set	/Students
Single entity	/Students(1)
Member access	/Students(1)/Name
Link traversal	/Students(1)/ClassRegistrations
Deep access	/Students(1)/ClassRegistrations(2)/Grade
Raw value access	/Students(1)/Photo/$value

Azure SDS
REST: URLでの式の表現

Sorting	/Students?$orderby=Name desc
Filtering	/Classes?$filter=substringof(Name, 'Math')
Paging	/Students?$top=10&$skip=30
Inline expansion	/Students?$expand=ClassRegistrations

Azure SDS
プログラミングモデル　C#編

Example Table Definition
Table Entities are represented as Class Objects

```
[DataServiceKey("PartitionKey", "RowKey")]
public class Customer
{
    // Partition key - Customer Last name
    public string PartitionKey { get; set; }

    // Row Key - Customer First name
    public string RowKey { get; set; }

    // User defined properties here
    public DateTime CustomerSince { get; set; }

    public double Rating { get; set; }

    public string Occupation { get; set; }
}
```

Create Customers Table

Every Account has a master table called “Tables”

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

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

```
TableStorageTable table = new TableStorageTable("Customers");

context.AddObject("Tables", table);
DataServiceResponse response = context.SaveChanges();
```

Create And Insert Entity

Create a new Customer and Insert into Table

```
Customer cust = new Customer(
 “Lee”,                 // Partition Key = Last Name
 “Geddy”,               // Row Key = First Name
  DateTime.UtcNow,      // Customer Since
  2.0,                  // Rating
  “Engineer”            // Occupation);
```

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

```
context.AddObject(“Customers”, cust);
DataServiceResponse response = context.SaveChanges();
```

Query　シンタックス　LINQ

```
from e in entities
  [where condition]
  order by [property]
  select e
```

```
from e in entities where e.Kind == "RedWine"
    select e
from e in entities where e.Kind == "RedWine" &&
    e["region"] == "New Zealand" select e
from e in entities where e.Kind == "RedWine" &&
    (e["region"] == "New Zealand" || e["region"]
    == "South Africa") select e
```

Query A Table

- LINQ

```
DataServiceContext context = new
DataServiceContext(“http://myaccount.table.core.windows.net”);

var customers = from o in
                context.CreateQuery<Customer>(“Customers”)
                where o.PartitionKey == “Lee”
                select o;

foreach (Customer customer in customers) { }
```

- REST

```
GET http://myaccount.table.core.windows.net/Customers?
        $filter= PartitionKey eq ‘Lee’
```

Update And Delete Entity

```
Customer cust = (
    from c in context.CreateQuery<Customer> (“Customers”)
    where c.PartitionKey == “Lee” // Partition Key = Last Name
       && c.RowKey == “Geddy”     // Row Key = First Name
    select c)
    .FirstOrDefault();
```

```
cust.Occupation = “Musician”;
context.UpdateObject(cust);
DataServiceResponse response = context.SaveChanges();
```

```
context.DeleteObject(cust);
DataServiceResponse response = context.SaveChanges();
```

まとめ

- 分散データベースへの変化
  - データベースのWeb-Scaleへの大容量化。
  - データベースへのScale-outの手法の導入。
- 分散メモリーキャッシュの活用
  - 大規模化に伴う、処理の高速化への対応。
  - データ保持・操作の主要な舞台が、ファイルからメモリーに変わりつつある。
- リレーショナルから、Key/Value型へ
- インターネットを通じたデータベースの利用