Project: Third party data

MKP Capital Management (abbreviated as MKP) consumes large amount of data from various third party data-provider companies. Recently, they want to integrate a new data set of commercial property transactions from RCA into their internal process, which includes integrating the data into the database, feeding the data into the internal model and the in-house analytics platform.

Before I work on the project, the situation was that the team was able to download an excel file from the data provider’s website portal using their purchased login credentials and account. The data in the excel file is then loaded into a data table in MKP’s database. However, this was a manual process and the data table was not updated automatically as new data came through over the course of time. Therefore, one of the main deliverable is to automate the process with C# and Microsoft SOL Server, making sure new data is merging and updating correctly in the internal database. In addition to this, the team also would like to integrate the data in their in-house analytics software, and add new functionality using the new dataset.

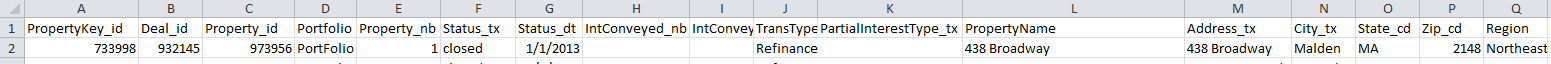
The first step is trying to download the data in csv format using the provider’s API instead of the website portal. To start with, I was initially given a sample excel file and an official instruction documented by the data provider on how to extract data from the several endpoints provided by their API.

I have written a C# class that is able to extract authentication token from these endpoints and eventually retrieve commercial property transactions. With this piece of code, one can specify if you would like to download the complete set of data or just the differential set of data. The downloaded file can be specified into one of these formats including csv, JSON or xml. In this implementation, I have chosen csv format. The reason is that C# already has a package called csvHelper that is extremely useful for parsing csv data into C# objects. Another reason is that csv file can be opened in excel, which makes the initial observations of the data easier. The current C# class name is RCADownload.

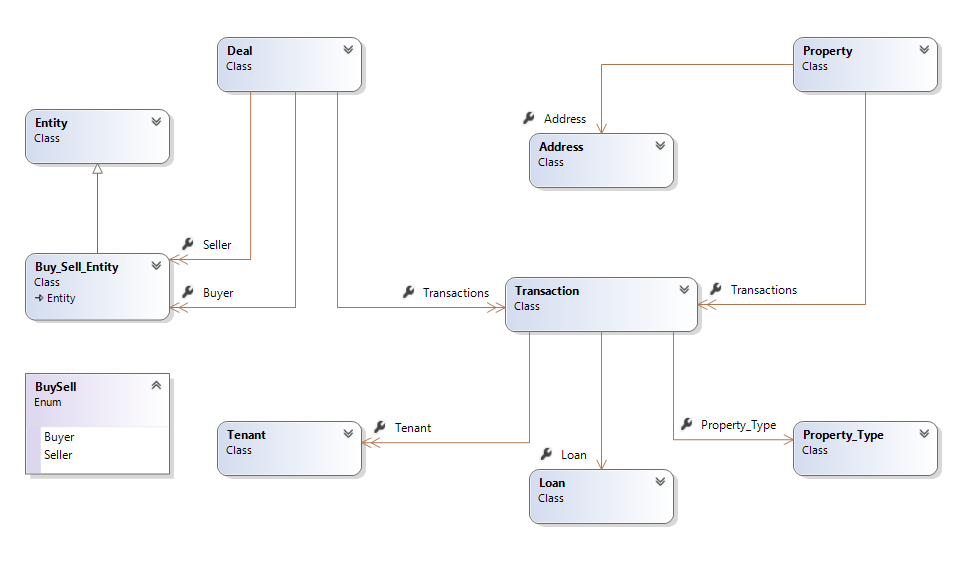
There are a couple of things that worth mentioning in this RCADownload.cs class:

1. The Boolean variable fullSetFlag controls whether you want to download the complete set of property transactions or just the differential set from the last download.
2. When a new download is initiated and is successful, previous downloaded file will be moved to a folder called “archive”. This makes sure we can keep good track of what has happened in the past history.
3. The last line in the file is not a valid csv row. Therefore, the program also deletes the last line after it downloads the file.

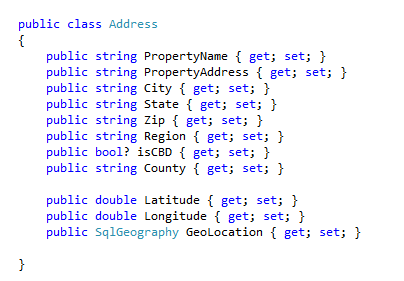
After opening up the downloaded file, we can see that, for each property transaction, there are more than 130 columns of related information. Besides, there are roughly 160000 rows within the complete data set. Below is a screenshot of how the data looks like in csv. As you can see, it is quite hard to fully understand what each column stands for by only looking at the column names.



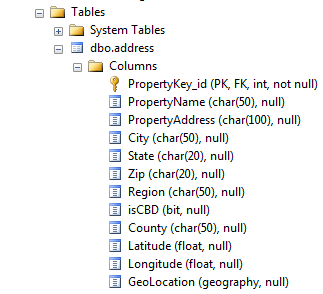
The next stage will be to list all the fields/columns available in this csv and try to categorize them into different groups. The initial step I have taken is to contact the data provider RCA directly, and ask for their documentation about how they actually defined these fields. They have sent me an excel sheet that have contained these information. By my own observation on the data together with their field definitions, I am able to come up with this design of the C# objects to hold the property transaction information. These C# objects will serve as an intermediary when the program is interacting with the database. Each row in the csv file represents one property transaction, and all these 130+ fields are grouped into one of the C# classes. There are also associations between these groups. For example, a deal or a property can have multiple transactions, and an address is associated with a property. For more details, please kindly refer to the C# class diagram and the RCAclass.cs class below.



Address is one of the C# classes and below is how it is written in the code RCAclass.cs.



We will use this design to setup the original SQL data table as well. Please refer to the screenshot below.



Now that we have the csv file and our C# design, we then need to parse the data from the csv into C# objects and then save them into the database. CsvHelper in C# is used for data parsing and stored procedure in SQL is used for updating to the database.

For the CsvHelper to work, I need to define a customized mapping between the C# class attributes and the actual columns in the csv file. This mapping is maintained in RCAclass.cs and RCAUpdateDao.cs, and below is an example. The actual column name and the class attribute name do not have to be the same. For example, Address\_tx is the actual column name appearing in the csv, but PropertyAddress is a more sensible names used in C# and SQL data table.

Example 1.

public sealed class AddressMap : CsvHelper.Configuration.CsvClassMap<Address>

{

public AddressMap()

{

Map(m => m.PropertyName).Name("PropertyName");

Map(m => m.PropertyAddress).Name("Address\_tx");

// some more attributes omitted here

}

}

Example 2.

thisDeal.Buyer.Add(1, new Buy\_Sell\_Entity()

{

Entity\_indicator = BuySell.Buyer,

EntityCapGroup = csv.GetField<string>("BuyerCapGroup1"),

EntityCapType = csv.GetField<string>("BuyerCapType1"),

EntityCountry = csv.GetField<string>("BuyerCountry"),

EntityJV = csv.GetField<string>("BuyerJV"),

name = csv.GetField<string>("BuyerName1") }

);

thisDeal.Buyer.Add(2, new Buy\_Sell\_Entity()

{

Entity\_indicator = BuySell.Buyer,

EntityCapGroup = csv.GetField<string>("BuyerCapGroup2"),

EntityCapType = csv.GetField<string>("BuyerCapType2"),

EntityCountry = csv.GetField<string>("BuyerCountry2"),

EntityJV = csv.GetField<string>("BuyerJV2"),

name = csv.GetField<string>("BuyerName2") });

// the original csv file does not have buyerJV3

thisDeal.Buyer.Add(3, new Buy\_Sell\_Entity()

{

Entity\_indicator = BuySell.Buyer,

EntityCapGroup = csv.GetField<string>("BuyerCapGroup3"),

EntityCapType = csv.GetField<string>("BuyerCapType3"),

EntityCountry = csv.GetField<string>("BuyerCountry3"),

name = csv.GetField<string>("BuyerName3")

});

This C# class RCAUpdateDao.cs is responsible for reading the csv file, parsing the data into C# objects, constructing the data tables from C# objects, and finally interacting with the stored procedure in order to update in the database.

var csv = new CsvReader(new StreamReader(fileName));

csv.Configuration.RegisterClassMap(new TransactionMap());

while (csv.Read())

{

Transaction tran = csv.GetRecord<Transaction>();

// following is omitted…

}

The above code constructs the transaction object according to the transaction mapping.

List<Transaction> transactionList = new List<Transaction>();

Dictionary<int, Property\_Type> propertyType\_map = new Dictionary<int, Property\_Type>();

Dictionary<int, Loan> loan\_map = new Dictionary<int, Loan>();

// property\_id mapped to 1,2,3, three tenants here

Dictionary<int, Dictionary<int, Tenant>> tran\_tenant\_dict = new Dictionary<int, Dictionary<int, Tenant>>();

// deal\_id mapped to Deal

Dictionary<int, Deal> deal\_map = new Dictionary<int, Deal>();

// property\_key\_id mapped to property

Dictionary<int, Property> property\_map = new Dictionary<int, Property>();

Dictionary<int, Address> address\_map = new Dictionary<int, Address>();

For each inserted row/transaction, we will update the above data structure accordingly. Update on Loan is simple, but extra efforts need to be invested in buy-sell entity, property and deal, due to the current design of buy-sell entity and also the fact that property and deal can have multiple transactions. Below is a C# code example for updating deal.

int deal\_id = csv.GetField<int>("Deal\_id");

if (deal\_map.ContainsKey(deal\_id))

{

deal\_map[deal\_id].Transactions[tran.Property\_id] = tran;

}

else

{

Deal thisDeal = csv.GetRecord<Deal>();

thisDeal.Transactions = new Dictionary<int, Transaction>();

thisDeal.Transactions[tran.Property\_id] = tran;

deal\_map[thisDeal.Deal\_id] = thisDeal;

}

Once we have all the data stored in these C# data structures and C# objects, we then need to merge/update the data into the database. The following two lines are used for this purpose. The first line will call a function to convert address\_map into a data table. The second line then calls a stored procedure named Address\_Merge, using the constructed data table object as an input parameter.

DataTable addressTable = BuildAddressTable(address\_map);

RCADb.ExecCommand("Address\_Merge @address", RCADb.GetDbParameter("address", addressTable, SqlDbType.Structured, "dbo.AddressTableType"));

Within the stored procedure, it has the following logic. For all those rows matched with the existing rows by primary key(s), all these existing rows will then be updated using the data in the new rows. For all those rows that are not matched, these new rows will just be inserted into the data table in the database.

MERGE INTO dbo.address WITH(TABLOCKX) AS i

USING(SELECT \* FROM @address) u

ON i.PropertyKey\_id = u.PropertyKey\_id

WHEN MATCHED THEN

UPDATE SET

i.PropertyName = u.PropertyName,

i.PropertyAddress = u.PropertyAddress,

i.City = u.City,

i.State = u.State,

i.Zip = u.Zip,

i.Region = u.Region,

i.isCBD = u.isCBD,

i.County = u.County,

i.Latitude = u.Latitude,

i.Longitude = u.Longitude,

i.GeoLocation = u.GeoLocation

WHEN NOT MATCHED THEN

INSERT VALUES(

PropertyKey\_id,

PropertyName,

PropertyAddress,

City,

State,

Zip,

Region,

isCBD,

County,

Latitude,

Longitude,

GeoLocation

However, there are exceptions for the logic above. The exception cases are tenant and buy-sell entity due to the way they are currently designed. This is because each row/transaction can have up to 3 buyers, 3 seller and 3 tenants. Therefore, when we want to update the tenants, we need to dissociate and delete in the database all the existing tenants already under that property\_id. After that, we will insert into the data table in the database again using the new tenant information. Below is an example SQL query for this exception case.

ALTER PROCEDURE [dbo].[Tenant\_Merge]

(

@tenant TenantTableType READONLY

)

AS

delete from dbo.tenant where Property\_id in (select distinct(Property\_id) from @tenant)

INSERT INTO dbo.tenant (Tenant\_name, Property\_id, Tenant\_number)

SELECT u.Tenant\_name, u.Property\_id, u.Tenant\_number FROM @tenant AS u;

DELETE FROM dbo.tenant

WHERE (Tenant\_name IS NULL OR Tenant\_name = '')

Another thing that worth noting is that in C# the sequence to execute these merge/update database commands should remain as it is. Otherwise there is a potential in breaking the foreign key constraints in SQL.

At this stage, we are now able to insert rows and update records in the SQL data tables.

SECOND part.

Write two functions that are able to extract and query property by geographical radius and by zip code. This is in the C# class named RCADataDao.

Geographical radius needs support of SQL geography.

Both SQL geography and zip code are under Address, and Address is only associated with property. In order to extract all the information, we would need to use Property to extract Transaction information and then use the ID in transactions to find all other information such as loan and deal.

public Dictionary<int, Property> CreatePropertyDictWithAddressByRadius(double latitude, double longitutde, double radius)

{

using (IDataReader reader = RCADb.GetDataReader(

@"select \* from dbo.address ad

right join dbo.property pr on ad.PropertyKey\_id = pr.PropertyKey\_id

where geography::Point(@lat, @long, 4326).STDistance(GeoLocation) <= @radius",

RCADb.GetDbParameter("lat", latitude), RCADb.GetDbParameter("long", longitutde), RCADb.GetDbParameter("radius", radius)))

Geolocaiton!!! Explain a little bit with this geolocation.

public Dictionary<int, Property> CreatePropertyDictWithAddressByZip(string zip)

{

using (IDataReader reader = RCADb.GetDataReader(

@"select \* from dbo.address ad

right join dbo.property pr on ad.PropertyKey\_id = pr.PropertyKey\_id

where Zip = @zip",

RCADb.GetDbParameter("zip", zip)))

If we try to rewrite the original standard SQL query into C#, it looks like above.

Separately query for deal, transactions and property. For each query, maintain a separate dictionary. Afterwards, I need to associate each C# transactions objects with its parent C# deal object. In order to achieve this, just loop through all the entries in the transaction dictionary. For each transaction, find its parent deal by matching transaction.parentDealId = deal.DealId. Potentially, we can have multiple transactions having the same deal. When a second transaction under the same deal is process, this implementation will just add this transaction to the transaction dictionary in the parent deal together with other transactions. Besides, this is also a very similar case with property.

Appendix:

Below is a local main class I have used to call the other functions.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using MKP.MAP.DataAccess.RCA;

using MKP.MAP.Models.RCA;

//using MKP.IntexWrapper.Data.BidListAnalytics;

namespace RCA\_reader\_main

{

class Program

{

static void Main(string[] args)

{

//Random rnd = new Random();

//basicDIConnector.logInAndReturnFullDIFile("sarmstrong@mkpcap.com", "L=m-\*^4L", "https://dataintegration.rcanalytics.com/1.0/", ref rnd);

//try

//{

// RCAUpdateDao.Instance.run();

//}

//catch (Exception e)

//{

// Console.WriteLine(e.ToString());

//}

//public static SqlGeography Point(double latitude, double longitude, int srid);

Dictionary<int, Transaction> uber\_transaction = RCADataDao.Instance.GetNearbyPropertityTransactionsByGeoLoc(38.5740407, -90.2812476, 4000);

//Console.WriteLine("hello");

Dictionary<int, Transaction> uber\_transaction2 = RCADataDao.Instance.GetPropertyTransactionsByZip("94598");

//Console.WriteLine("hello");

}

}

}

Contruct data tbake,

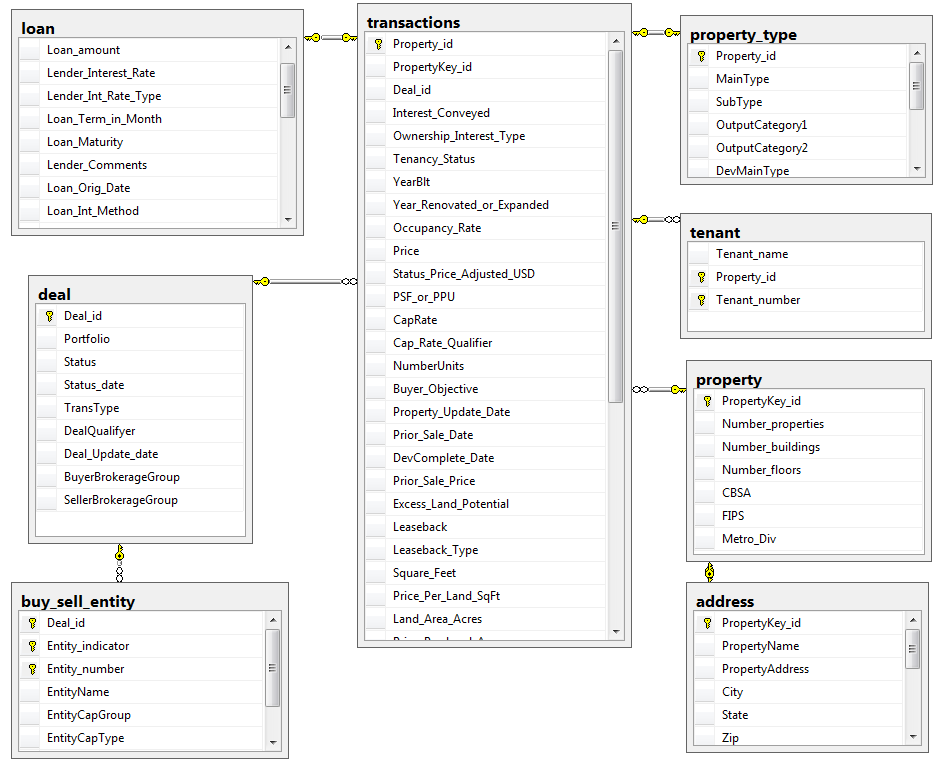
Use the datatable to call the store procedure in sql server.

The stored procedure will merge and update the data in each data table

C# Object:

* Define a table structure in the database to store all of the information
* Define C# objects to hold all the information
* Utilize the API to download all of the transactions and save to the database, using the defined C# objects as an intermediary
* Write the code to load the Transactions from the database into C# objects to be utilized by our models

Document the table structure, c# objects, and code that utilizes the API



The loading data by radius or zip code is complete.

As a summary, the work is around …..cs. … . .