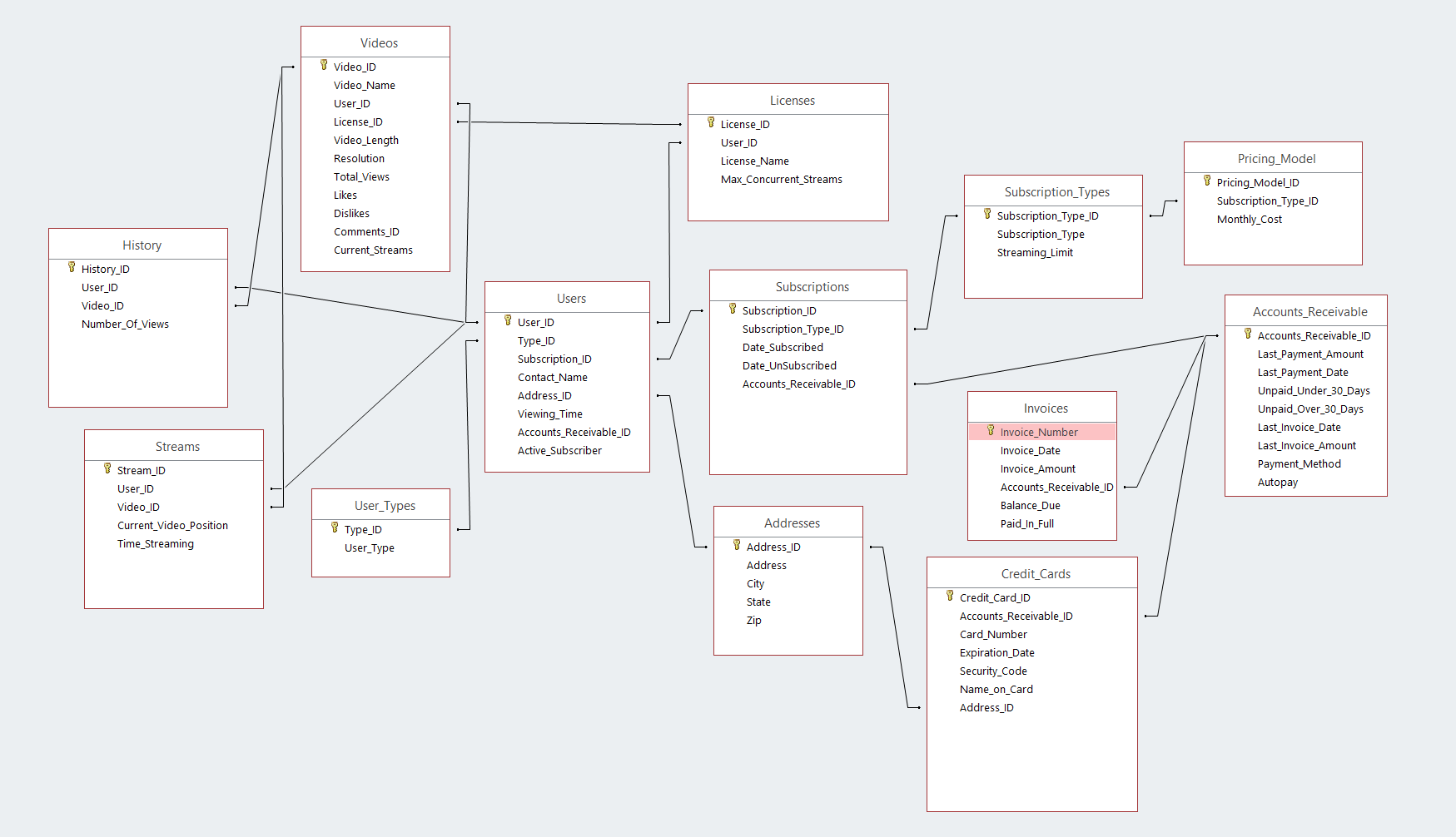


Lifion Developer Candidate Case Study

Video Streaming Service

**ER Diagram**

**SQL Query –**

SELECT TOP 20

Users.Contact\_Name,

Videos.Video\_Name,

History.Number\_Of\_Views

FROM (

History

INNER JOIN Users

ON Users.User\_ID = History.User\_ID

)

INNER JOIN Videos

ON Videos.Video\_ID = History.Video\_ID

ORDER BY History.Number\_Of\_Views DESC;

**Algorithm –** can also be found at - <https://github.com/adjit/Lifion/blob/master/Lifion.cs>

using System;

using System.Collections.Generic;

public class CheckSubs

{

/\* Function doesn't need it's own class \*/

public CheckSubs(Dictionary<int, Subscriber> subscribersAR, List<Invoice> invoices)

{

/\* Couple of things here --

\*

\* subscribersAR will be a dictionary of subscribers using the Accounts\_Receivable\_ID

\* as the key for easy AR access.

\*

\* Alternatively, you could pass just plain list of subscribers to the function and

\* put the subscribers into a dictionary with the Accounts\_Receivable\_ID, but that

\* would be better off handled by some main process that would create necessary

\* dictionaries. Let's say multiple functions would use the subscribersAR - don't

\* want to go through the process of setting up a dictionary multiple times.

\*

\* Since the Database will handle account ageing and updating account balances

\* and amounts due under 30 days and amounts due over 30 days, finding which

\* subscribers are delinquent is a trivial process of checking which accounts

\* have balances over 30 days.

\*

\* However, lets say it's ok to be over 30 days but under 60 days. The db

\* doesn't have Over 60 days - you could add that data column in your database

\* which will be updated by the ageing process and then again this would be a

\* trivial process. Check if subscribers have amounts due over 60 days.

\*

\* For this case, will assume that the over 30 and under 30 variables are

\* not available and need to check invoices

\*

\*/

DateTime now = DateTime.Now.Date;

/\* Dictionary using the Accounts\_Receivable\_ID as it's Key and has a list of

\* delinquent invoices as its Value.

\*/

Dictionary<int, List<Invoice>> delinquentSubs = new Dictionary<int, List<Invoice>>();

/\* Loop through all invoices that are passed

\*/

for (int i = 0; i < invoices.Count; i++)

{

/\* If the invoice is younger than 60 days move on, subscriber can

\* still pay. The value doesn't have to be 60 days. You could add

\* a variable that would used to check. Just using 60 for this

\* example

\*/

if (now.Subtract(invoices[i].invoiceDate).Days < 60) continue;

/\* Otherwise, invoice is way over-due and the customer needs to

\* be informed their account will be cancelled. But, the subscriber

\* may have multiple open invoices, so just put delinquent invoices

\* into the delinquentSubs dictionary using the Accounts\_Receivable\_ID

\* as the key

\*/

else

{

List<Invoice> delinquentInvoices;

int key = invoices[i].ARID;

/\* Try to get the key value in delinquentSubs. If that value doesn't

\* exist, then create a new invoice list for that key, and then add

\* the delinquent invoice to that list.

\*/

if (!delinquentSubs.TryGetValue(key, out delinquentInvoices))

{

delinquentInvoices = new List<Invoice>();

delinquentSubs[key] = delinquentSubs;

}

/\* Could choose to notify the delinquent sub here, but what if they have multiple

\* invoices outstanding. Could result in multiple messages being sent. Why not just

\* collect all delinquent invoices and then send them all at once. \*

\*/

delinquentSubs[key].Add(invoices[i]);

}

}

/\* Go through all items in the delinquentSubs dictionary and notify

\* the subscribers of delinquent invoices.

\*/

foreach(var item in delinquentSubs)

{

subscribersAR[item.Key].notifyDelinquent(item.Value);

}

}

}

/\* Payment object - not used in this scenario, but would be

\* used to get more information about payments

\*/

public class Payment

{

public DateTime PaymentDate;

public int PaymentAmount;

}

/\* Invoice class which takes the corresponding data from the

\* database and turns it into a usable object

\*/

public class Invoice

{

public DateTime invoiceDate;

public int invoiceNumber;

public int invoiceBalance;

public int invoiceAmount;

public int ARID;

public Invoice(DateTime date, int num, int bal, int amt, int arID)

{

invoiceDate = date;

invoiceNumber = num;

invoiceBalance = bal;

invoiceAmount = amt;

ARID = arID;

}

}

public class Accounts\_Receivable

{

int Amount\_Due\_Under\_30;

int Amount\_Due\_Over\_30;

List<Payment> Payments;

public Accounts\_Receivable(int ARID)

{

/\* Query Database for AR data including

\* payments and load into class variables

\* Database should take care of Ageing and

\* updating and amount due

\*/

}

}

/\* Contact Information class that would be used when

\* sending subscribers notices. This object contains

\* the subscribers contact information.

\*/

public class Contact\_Information

{

public string contact;

public string address;

public string city;

public string state;

public string zip;

public string email;

public string phone;

}

public class Subscriber

{

/\* Only allow people to get variables

\* Changes need to be handled by the class

\*/

private string \_sub;

private int \_subID;

private int \_ARID;

private Accounts\_Receivable \_AR;

private Contact\_Information \_CI;

public string subscriber { get { return \_sub; } }

public int subscriber\_ID { get { return \_subID; } }

public int Accounts\_Receivable\_ID { get { return \_ARID; } }

public Accounts\_Receivable AR { get { return \_AR; } }

public Contact\_Information CI { get { return \_CI; } }

/\* Contacts, address's etc.

\* Class will be passed all variables or make

\* the proper function calls to retrieve the

\* database information and gather it

\*/

public Subscriber(string sub, int subID, int arID)

{

/\* Basic constructor - only set subscriber and ID's \*/

\_initialize(sub, subID, arID);

}

public Subscriber(bool get\_AR, string sub, int subID, int arID)

{

/\* Overload Subscriber to load various

\* items upon initialization. In this case:

\*

\* Flag to get AR upon initialization

\* - what if we didn't need that data?

\* pointless to load it in otherwise

\*

\* Can add more flags in the future

\*/

\_initialize(sub, subID, arID);

\_loadAR();

}

public void notifyDelinquent(List<Invoice> invoices)

{

/\* Send email / mail subscriber a delinquency notice of cancellation

\* and delinquent invoices

\*/

}

public void Load\_Accounts\_Receivable()

{

/\* Function to use if you want to load in AR later

\* or refresh

\*/

\_loadAR();

}

/\* Initializes the basic subscriber values that will always be passed.

\* Since there may be overloading in this class, instead of each function

\* setting these 3 variables this one function will handle that

\*/

private void \_initialize(string sub, int subID, int arID)

{

\_sub = sub;

\_subID = subID;

\_ARID = arID;

}

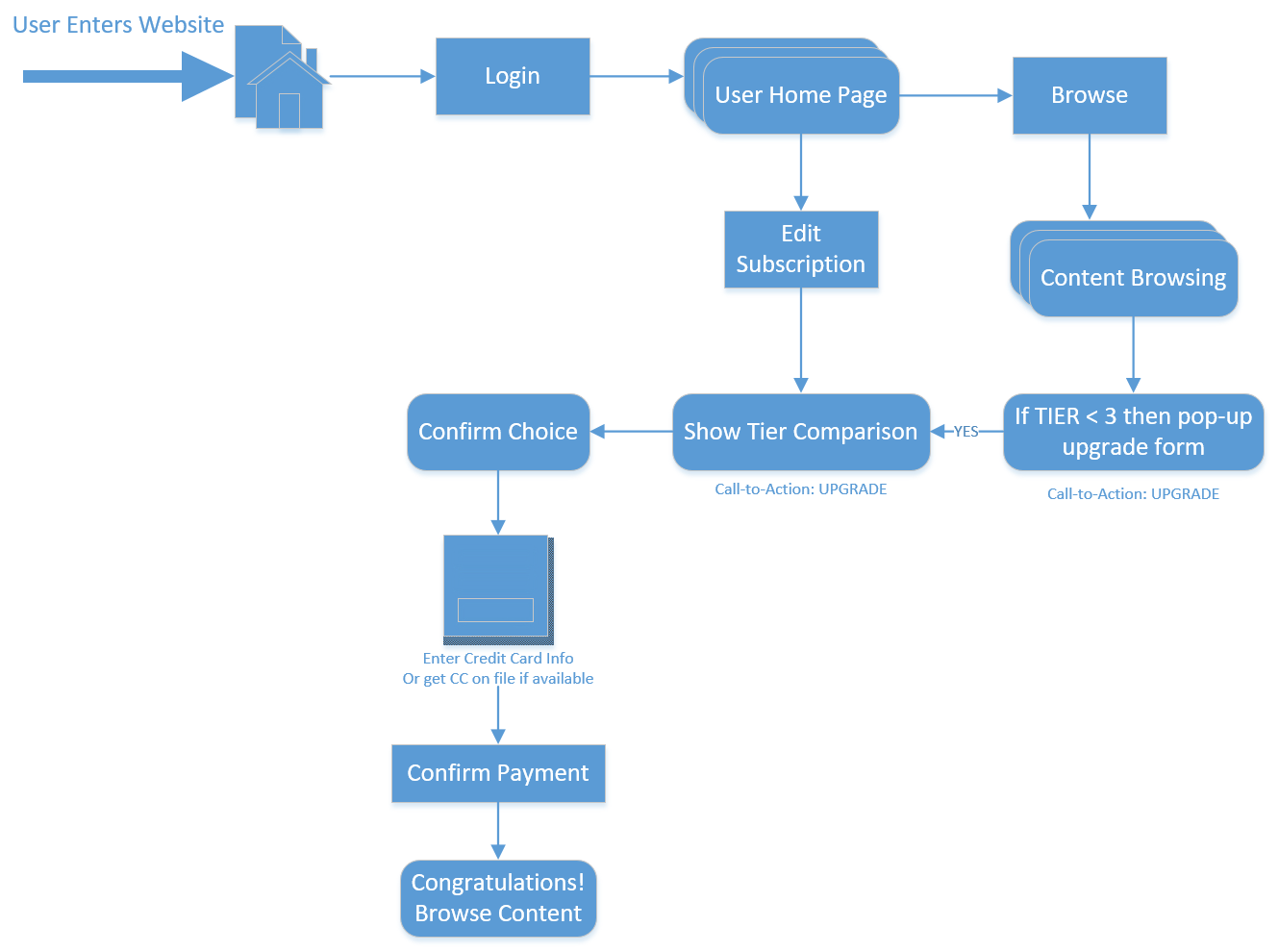
private void \_loadAR()

{

\_AR = new Accounts\_Receivable(\_ARID);

}

}

**Product/UX**

**Case Overview & Requirements**

“Vidsi” is a new video streaming service you are building for a startup company.

* Subscribers sign up for free and receive 2 hours of free streaming
* For paid subscribers, streaming limits are based on plan tier
* There are currently 3 tiered subscription plans - but they may change in the future
* There’s a catalog of videos to stream that are licensed from different content providers
* Subscribers stream videos and we need to store information about what’s being streamed
* We need to generate monthly invoices for subscribers that can be paid by credit card on file or by check
* For subscribers charged by credit card we need to store payment information (you can assume security and PCI compliance don’t exist - you can store credit card numbers to charge)
* Licenses with content providers only allow for a certain number of concurrent subscribers to watch a given video

**Data Modeling**

Given the above requirements please create a data model in the form of a complete physical entity relationship diagram (ERD). Your diagram should be comprehensive and sufficient for creating a relational database schema that will support the applications’ requirements.

**SQL Query**

Write a SQL query to return the top 20 video/user pairings. In other words, show which videos have been most re-watched by a single user.

For example:

User A, Video A, 30 Views

User B, Video A, 29 Views

User B, Video B, 19 Views

User A, Video C, 18 Views

**Algorithm**

Write a function that, given a list of invoices and subscribers, would find subscribers who haven’t paid their invoices and need to be sent a cancellation notice.

You can use pseudo code or any programming language of your choice to write this!

**Product/UX** (Optional)

Create a process flow/wireframes that would show how subscribers would change their subscription from free to paid tier.