Report - The Intelligent Web

Introduction

This project is to design and implement a website to allow users to track and analyse discussion relating to football teams, their players, and to automatically generate reports on upcoming games.

We have completed each of the requirements described in the specification, including additional features. Each of these requirements have their design choices and issues described below.

1.1. Querying the social Web

Issues

We ran into several issues while implementing this section, some of which we have outlined below:

When performing the analysis of the returned tweets we had to process each tweet's contents. Some of the words needed to be changed slightly to allow the processing of keywords. For example, '*@ManUtd*' needed to be the same as '*@manutd*', '*@ManUtd:*' and '*.@ManUtd*'.

When querying Twitter, the main issue we came across was understanding how to perform complex queries in one transaction, spending lots of time on the Twitter API page [3] we managed to work out how to do things like filter out retweets and get user specific tweets.

Design Choices

We have used the 'twit' module for nodejs to access the Twitter API, as the API only allows up to 100 tweets to be retrieved in a single transaction we designed a recursive function that would retrieve a specified number of tweets, or until a tweet id limit. This is the basis for our Twitter search, regardless of whether there is data in the database or not.
The analysis works by making a "wordCount" for each tweet as we process the data. This wordCount is a javascript object working like a dictionary where for each word within that dictionary we are given a value that is the number of occurrences the word has within the text. By adding the keys and values into an 'overall' word count we can produce an array of the words contained within the collection and slice the highest 20 values. Along the way we can also record for each tweet each user, and for each user we can retrieve their personal wordCount, their username and the number of tweets they have written.

Our maps implementation has been done by making use of the freely available Google Maps API. This allowed for an interactive Google Map to be embedded into our site and for markers to be shown at the locations of retrieved tweets. When these markers are clicked, we filter the tweets column to show just that tweet (described further in additional features). The locations of the tweets were found by using the 'place.full_name' information returned with tweets and by using geolocator, which is included in the maps API, a plottable longitude and latitude could be returned.

Some design choices made along the way. We implemented a "trending hashtags" feature, where in addition to being presented in the top 20 keywords, the top 5 popular hashtags will also be displayed. Secondly, if a Twitter user has only tweeted once, and they are in the top ten active users slice, they will not be displayed. This is simply because a user tweeting once does not have

any significance. We added in a filter for the top users section, whereby when you click a top user, the tweets are filtered to just their tweets, similar to the map filtering.
This design meets the requirements within the specification - Complex queries can be performed exceptionally fast and the analysis is precise and informative.

Limitations

When implementing the map we found that very few tweets contained location data, in a sample of 300 we found there were less than 10 locations in general - this makes any analysis of location pointless and the map quite sparse.

1.2. Storing information

Issues

This section is focused on integrating a database into our solution to allow tweets to be retrieved directly from it, or used as a cache alongside the Twitter API. There were several issues we came across in the implementation, the largest being the speed of accessing and writing data - We managed to get good results after changing the design slightly which will be discussed in the next section. Another issue we encountered was retrieving tweets from the database that hadn't been searched for directly before and handling errors in accessing the database gracefully.

Design Choices

The first choice we made was selecting MySQL for our database, via the DCS server (this means it will only work when the server is within the university network), rather than a local storage database which would not be as portable or scalable. We use three tables with relationships to each other via foreign keys, the tables are as follows; users - to store the user data, tweets - to store the tweet data, and media - to store the media data. This approach decreased the memory overhead by not repeating data for users or media that have been used in a tweet before.
You can use the database as either an independent source (/search/database) or as a cache (/search/twitter). When you request it as an independent source we will create a WHILE mysql statement from the query which will be used with the rest of the sql statement to get the tweets you desire. When you request the data as a cache it will initially do the same thing, but once it retrieved the tweets from the database it then finds the maximum ID in the set of tweets, it will then use that maximum ID to retrieve only new tweets from Twitter, up to a specified maximum number of tweets. Once the tweets have been retrieved from Twitter we then combine the two sets of tweets and send them back. We then go on to save all the tweets into the database for future calls.
Initially, our design only allowed us to search for queries in the database that had previously been searched for exactly - this was a slower and less versatile solution that relied on a few more tables and many more foreign keys, which slowed insertion into the database down greatly.
To speed up the database further we changed the connection from a single connection (causing a bottleneck) to a pool of connections - this required extra attention to make sure we closed the connection properly.
We met all the requirements for the implementation of the database and are happy with our final solution.

Limitations

Currently the database cannot be searched by location. It would not be difficult to implement this in the WHERE statement, but currently our interface does not support location specification. If we were to implement this it would be difficult to search by location for any location scales different to those stored in the database. For example, it would be difficult to search for tweets in either ‘United Kingdom’ or ‘South Yorkshire’ due to the location information being stored as ‘Sheffield, England’.

1.3. Producing a tool for sports journalists

Issues

When designing the interface we found our main issue to be providing an intuitive way for users to search for and view returned tweets, we wanted to give users the ability to really customise the search but without making it too complicated.
The largest issue when producing the tool for sports journalists was firstly learning the syntax for SPARQL in order to properly retrieve the required data. In particular it was confusing to discern what resource to use to properly retrieve the data. The solution to this was to practice writing queries online using the DBPedia Sparql interface [1] to retrieve arbitrary data before beginning to write relevant queries. There was a large issue with the correctness of the data present on DBPedia, this ranged from stadiums not being represented as a URI to factually incorrect information. The solution to this was to implement a large amount of IF statements along with OPTIONAL constructs within the original query. We were able to check if specific data was undefined and then display the data accordingly without crashing the server. The implementation of OPTIONAL constructs within the query also solved the issue of the query failing when one of the query terms was not found. Regarding the twitter querying there is a problem that the interface can become quite large for very specific queries. It was decided that as the query is for a specific use by a journalist that we would leave the interface as is.

Design Choices

Within the interface we give users the option to choose a team to query and multiple players, hashtags and keywords - this allows for the users to get the exact tweets they want especially by including AND/OR switches, mention switches and retweet switches. The design is ‘modular’ and allows users to add more search terms by simply pressing a + button. We provide two submit buttons, one to get tweets from just the database and another to get tweets from twitter and the database.
A big design choice for the tool was to implement player cards. This was implemented to simulate match stickers where the main information about the player is on the front and by clicking (turning over) the card we can find a short description of the player. Query-wise, the code was designed to be scalable and for the query to be easily changed. We can either query the club data, or player data for a given resource. The code is easily alterable to query for multiple teams. An additional design choice we implemented was, where possible, extracting each football team’s colour to use as a colour scheme when displaying data, however as previously mentioned this data is not available across all teams. It was decided that it was best to leave the user the choice of querying DBPedia or Twitter on the homepage. This allows us to keep the design streamlined. The twitter report page implements some novel features such as using regular expressions to workout what the most popular hashtags are and displays them accordingly. Knowing what the top hashtags are

means that journalists can access the trending topics more quickly. We went forward with using the sparql client for node[5] for it's ease of use.

Limitations

A limitation of our interface is not being able to customise the and or selectors on a term by term basis, having this would allow users to perform even more complex searches at the expense of making the system more confusing to the user.
A limitation of the system is that we assume that the DBPedia data returned is correct and there is no semantic evaluation performed of whether a club's 'ground' is actually it's stadium or just where the stadium is located.

1.4. Producing Data for the Web of Data

Issues

An issue that we had with producing data for the web of data was finding where in the dynamically generated HTML to add in the RDFa tags. Due to the complexity of the function generating our HTML, in particular for the player profiles, it was not always simple to add in the addition required tags.

Another issue that we had was testing for this data being presented properly. This was a pain to do with a crawler and so we made use of RDFa's *play* tool [2] to ensure that the data was correctly machine readable and returning the correct information.

Design Choices

The design choice for producing data for the Web of Data was, as described in the assignment description, to link back all data retrieved from the Web of Data to it's source. This was done using RDFa tags in the HTML.

Limitations

Due to the way we currently display the player abstract (as a pop up generated upon click) it is not possible for a crawler to see that information and index it. This could be fixed by including an invisible field in the player profile with the abstract in it.

1.5. Quality of the solution

Issues

There were multiple issues that arose from us not having used several technologies such as AJAX or nodeJS before which caused meant there was a great learning curve, during which many mistakes were made. As we became more proficient in the technologies this issue was mitigated.

Design Choices

Our solution makes use of HTML5 to check the inputs provided by the user, JavaScript and AJAX to give the user a well presented experience without the need to reload pages, JSON to exchange

data between the client and the server, variable routes to allow for more flexible and less repetitive code. We also use the JQUERY library to allow us to easily access and edit the DOM.
Our solution meets the requirements and provides a great experience for end users.

Limitations

It became apparent through testing that it was not irregular for tweets and images of an inappropriate or adult nature to be returned in searches. A limitation of our current implementation is that we do not have a system for dealing with these. A potential addition to remedy this would be the use of a language filter or to filter content by taking into account the 'possibly_sensitive' flag that Twitter includes in metadata.

1.6. Additional features

Issues

The main issue we had when implementing additional features was getting them to fit in with our existing code - quite often implementing a new feature required major changes to existing code, which made it very time consuming.

Design Choices

We decided to integrate Flickr into the solution to provide some pictures about the general search topic, we created a new route for this, '/flickr', and use the 'node-flickr' npm module to get the data. The implementation is fairly simple and just takes any input from the query as a searching tag and returns the top 30 images relating to those tags.
We thought it would be useful for the user to include a simple drop down box for all the premier league teams and their players. We get this data regarding teams and their players from an ajax call to the airstadium api [4]. When using the player drop down box, if you click on a player it will add it as a textbox input so you can add multiple and delete them later on. This allows for more versatile search queries.
We added in a function to filter tweets based on their map marker, or the top users - this required extra attribute tags on each tweets to be able to find them and hide them after the initial load.

Limitations

Flickr can sometimes return no results, which is an issue - we mitigated this by checking if results were returned. If no results are returned then we hide the picture column without the user ever seeing it. Another issue with Flickr is that the API only accepts a maximum of 20 tags, therefore we simply take the first 20 tags provided by the user, this will potentially cause some tags to not be considered if the user searches for too many.
Sometimes you may want to search for a team or a player not in out drop down lists, therefore we included custom text boxes so that you will always be able to search for whatever you like, we simply provide suggestions.

Conclusion

A lesson that we learnt as a team was to pull together out different interpretations of the, sometimes vague, specification. By offering our different ideas about what we thought was expected we managed to create a much more thorough plan of what we wanted the system to be.

Learning to write asynchronous code was also a lesson well learnt as it required a different way of thinking compared to most coding that team members had done previously. It took a bit of time to adjust to the new style of writing but by learning it together we all got the hang of it fairly quickly.

Division of Work

Work for individual team members was decided at meetings scheduled weekly to discuss progress and plans for the following week.
The initial setup of the mode server was completed as a team during the first meeting. All files have author tags in them to show division of work.

Name	**Agreed Percentage of Work**	**Worked On**
Paul MacDonald	33.33%	Database Design and Implementation, queries made to Twitter, flickr, airstadium and the database, the interface and the ajax code for the report.
Ross Millward	33.33%	Implementation of the map, interface design and producing data for the web of data.
Alex Burley	33.33%	The analysis code and queries made to the dbpedia api.

Extra Information

- You must be on the university network for the database to work (or connected via vpn)
- You can run in the root directory using ‘node solution/bin/www’ or from within solution/ using either ‘node bin/www’ or ‘npm run’.

Bibliography

- [1] - http://dbpedia.org/sparql
- [2] - RDFa, Play, http://rdfa.info/play/
- [3] - Twitter Rest Api, https://dev.twitter.com/rest/public
- [4] - Airstadium API, http://www.airstadium.com/api/
- [5] - SPARQL Client https://www.npmjs.com/package/sparql-client