Database project report

Unfortunately my implementation was a failure. My plan was to create a sqlite3 database management waper designed to allow storage of images in a database using node.js. It is implemented such that as a user i can insert a image file path and keywords into a database table and then at a later time perform queries on those keywords and have the image be found. This would allow a user to build a website and easily query for images off a home server and have them be served to the webpage. But due to time constraints and lack of planning and foresight the implementation does not work.

The plan was to implement C.R.U.D for a “fake” image datatype. Creation was to be done by parsing the creation query and finding the image/IMAGE keyword and attaching a flag to the column that had the image data type associated with it then replacing the image/IMAGE keyword with a varchar that would hold an object with the images information. When it came to reading i came up against several walls that had to do with parsing queries.

I found it incredibly difficult to parse the queries to acquire the relevant information required to find the image object based on the keywords. First i was going to find the relevant table and columns then after finding the column with the image flag. It would be simple to pull the image object and compare the keywords against the query's where clause. But i was unable to do this.

The most developed function is the database image insert function. It takes a query, some image keywords, and file path. Using the image file path and keywords an image object is created and then this object is inserted into the relevant column of the table. But there was a bug that during the addition of a new image that brought development to a standstill. This bug related to the flag used to identify the image column. When i was trying to perform a query to identify and insert the image i was never able to find the correct column. This is due to my own ineptitude and inexperience with the sqlite3 library and i was unable to fix this issue during my time spent on this project.

Below are the algorithms that i was planning on implementing.

```
imageCREATE (query)
        If query is of type create continue
                Let clmIndex = index of column with image datatype
        Replace image from query with varchar 60 (to later have image object inserted)
        Append image key string to cololumn name
        Run create query
```

```
imageINSERT(query, image, keywords)
        If query is of type insert continue
        Find the column with the image key string
        Find imagedata associated with the image column ( this would be the file path)
        Create image object
                imageOBJ.path = path
                imageOBJ.keywords = keywords
        Replace image data from query with image object
        Run insert query
        Save image to db image folder

imageSELECT
        If query is of type select continue
        Select all rows for table specified in query and let R = rows found
        For each row in R
                If imageOBJ.keywords contained keyword or path == path
                Return row

ImageUPDATE
        If query is of type update continue
        Select all rows for table specified in query and let R = rows found
        For each row in R
                If imageOBJ.keywords contained keyword or path == path
                Update imageOBJ to reflect query
                Return row

ImageDELETE
        If query is of type delete continue
        Select all rows for table specified in query and let R = rows found
        For each row in R
                If imageOBJ.keywords contained keyword or path == path
                Get imageOBJ.path and delete image located there
                Run delete row query
```

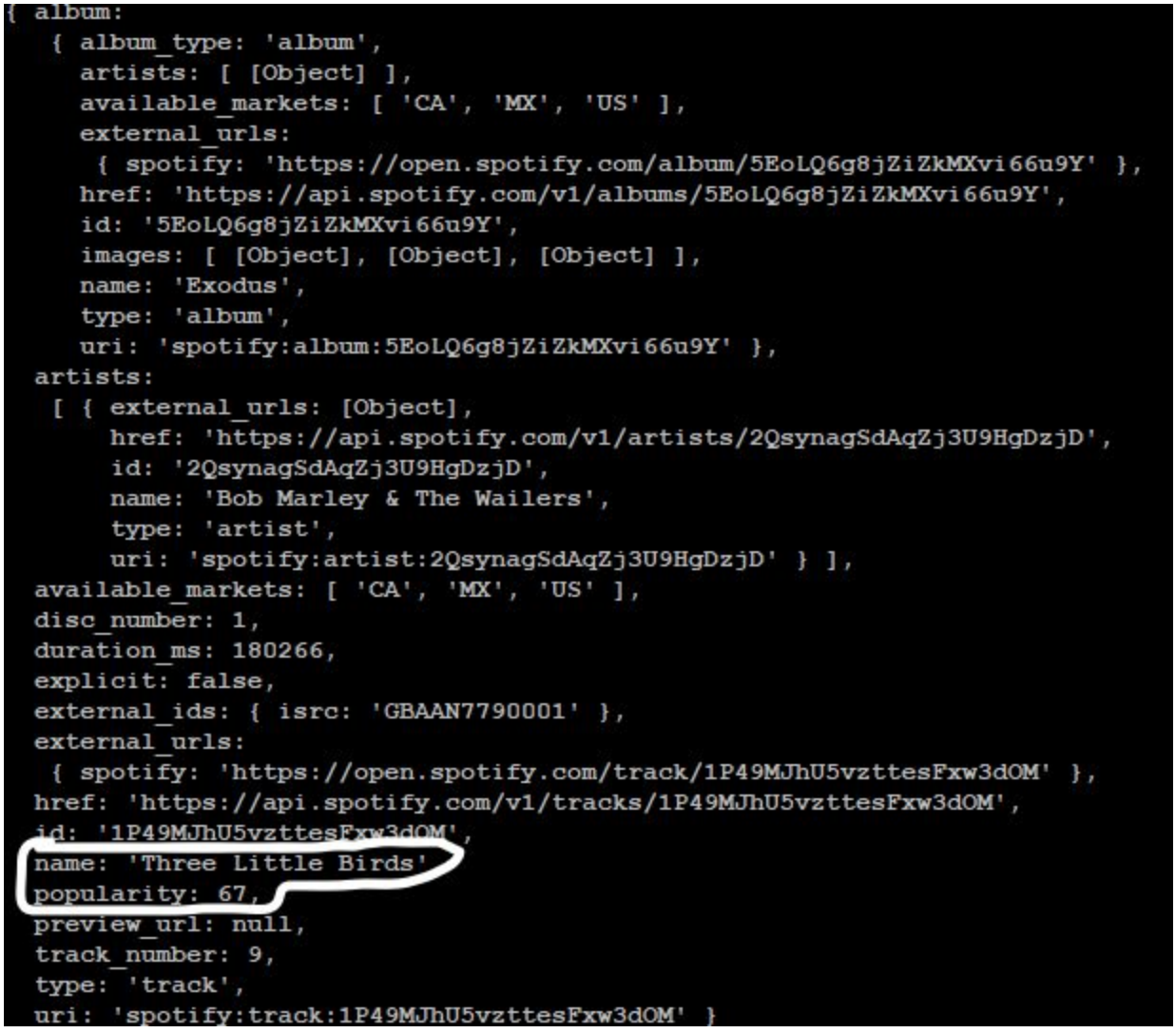
In conclusion In the future i will do more research into the tools i plan to use. I also realized that this was implemented much better by a normal mysql server with a blob, keyword column and, file path column in one row. You could then perform a query for any keyword or file path and receive the blob associated with it. The paper i was using for my implementation described the complete implementation for a new sql db management solution. The scope of a brand new dbmanager was far too large for me to complete in the timespan of the project this was something i only realised after i had started my development.

Because of the failure mentioned above i have created another report on the preprocessing required to generate frequent item sets.

The component being implemented

The component being implemented is a pre-processing program designed to output an array of transactions and a list of frequent one-itemsets to be used while mining frequent itemsets in song titles, popularity, and genre. It was designed and implemented in node.js, a server side javascript framework. The purpose of preprocessing my data was to reduce the amount of space that was required for storage by removing irrelevant information and to allow all of my frequent itemset algorithms to have a consistent and stable dataset to work on. As well as splitting song titles into individual words and removing punctuation.

Before pre-processing one transaction looks like this.
(Figure 1) The information that is relevant to me is circled in white.



As you can see it has a lot of information that is not useful for generating frequent itemsets based on name, popularity, and genre. When querying the spotify api i was searching for tracks based on genre and because a song can have more than one genre this led to duplicate values e.g. if a song was both rock and blues i would get the same track from both my rock and blues searches these duplicates skewed results when doing frequent itemset generation and thus are also removed during preprocessing.

Because popularity is a numerical value (0 - 100) it needed to be quantised. This was done as follows.

most popular (90 - 100)
very popular (80 - 89)
fairly popular (70 - 79)
somewhat popular (60 - 69)
popular (50 - 59)
not very popular (40 - 49)
not popular (30 - 39)
unpopular (>30)

Another aspect of preprocessing relates to the order of items in each transaction. for FPgrowth the items in your transactions should be ordered such that items that are more frequent are stored at the beginning of a transaction. So when building an fp tree, elements that are more frequent are inserted into the tree first reducing branching. First the items from each transaction are counted and inserted into a list if they are above the minimum support threshold. This list is then ordered from greatest to least and then each transaction is recreated based on the order of the list and reinserted into the array of all tracks.

After processing the record in figure 1 would look like this.

{
"name":["three","little","birds"],
"popularity":"somewhat popular (60 - 69)",
"genre":"genre: reggae"
}

Evaluation

My evaluation strategy was to store the unprocessed transactions shown above (figure 1) as well as the preprocessed transactions then compare the resulting file sizes using windows file manager.

When storing 3000 unprocessed records the resulting file was 7,794 kb. On the other hand the processed transactions file was only 288 kb this is a reduction of 2700%. Another benefit of reducing the space required is a reduction in the time required to read a transaction although since the transactions are stored sequentially in text format, the time saved is negligible for any reasonable amount of data.

Conclusion

The preprocessing of data is worth the time investment required as it can save on cost and headaches in the future. Another benefit of doing my preprocessing separately from the frequent itemset generation had to do with the spotify API. The API limits the amount of requests that can be sent to the spotify servers within a short amount of time. This is done to limit DDOS attacks and make the API more fair for low use users. By doing my preprocessing in advance i could do it in batches reducing the strain on spotify's servers.