UNIVERSITY OF CAPE TOWN

DEPARTMENT OF COMPUTER SCIENCE

COMPUTER SCIENCE HONOURS FINAL PAPER 2015

Title: Data Management of an Exploratory Search System

Author: Dylan Henderson

Project Abbreviation: Travsrch

Supervisor: Maria Keet

Category	**Min**	**Max**	**Chosen**
Requirement Analysis and Design	0	20	20
Theoretical Analysis	0	25	0
Experiment Design and Execution	0	20	0
System Development and Implementation	0	15	15
Results, Findings and Conclusion	10	20	15
Aim Formulation and Background Work	10	15	10
Quality of Paper Writing and Presentation	10		10
Adherence to Project Proposal and Quality of Deliverables	10		10
Overall General Project Evaluation (*this section allowed only with motivation letter from supervisor*)	0	10	0
Total marks		**80**	**80**

Data Management of an Exploratory Search System

Dylan Henderson
Department of Computer Science
University of Cape Town
Rondebosch, 7701, South Africa
Dylan9h@gmail.com

ABSTRACT

When users search for flights for holiday destinations, they can be given the option to make keyword based exploratory searches when they are not entirely sure of what they are looking for. The system required to allow for such queries would need a data management layer to manage initialization, insertion, retrieval and updating of destination based data. Flight data is currently not available in a form required to make keyword based queries. Destination data needs to be linked to keywords and flights to allow for these exploratory searches. Keywords must be defined by their relevance to destinations. One of the problems is that there are conflicts between stored categories and user queries, this is solved by allowing keywords to be built up from user input. A system was developed to allow for keyword based searches, this paper focuses on a solution to the data-management of the system. The development methodology followed an agile approach allowing the system to be developed in short iterations. Keyword-destination pairs are stored with a weight indicating the relevance of the paired keyword to the destination. Weightings are updated based on user input with an initial manually created seed of weights. Use case scenarios were used to test the functionality of the system while running. The system allows for storage and retrieval of keyword based results that are user-driven and automatically updated. Future work could improve the system to allow for multi-word keywords and more destination based information.

Keywords--- database, mongoDB, keyword, flights, destination, search, use case scenario

1. INTRODUCTION

Searching for flights online has become a common task for anyone wishing to travel. To access the vast number of available flights, users need a way to quickly query results, be this for a known item such as a flight to a specific location on a specific date or an exploratory item [1] such as users searching for a holiday location while being unsure of where they are going. Current flight websites offer users the ability to search for flights based off of various preset choices that the user must make. These generally include the selection of a departure date, a return date, an origin location and a destination location. Some flight websites such as Travelstart [2] offer some more advanced selection options such as non-stop preferences, variable dates, and ordering of results by price or by the faster flights but do not offer options for natural language based queries such as a search for a keyword. An exploratory system has been developed to allow users to have even more control over the available data. The system allows users to perform keyword based exploratory searches when searching for flights.

System Documentation and system design process information can be found at http://pubs.uct.ac.za/cshonours/2015/Travelstart

The aim of the system is to allow users who do not know exactly what they are looking for to explore available destinations and flights. These types of users are generally users searching for holidays and as such the system focuses on linking destinations with relevant keywords. Users enter a query into a search bar in plain English such as "an exciting romantic location". They enter their origin and also have the option to enter a price range and possible destinations, but these are not required. Users are then presented with a list of flights that match their query.

The entire system uses a multi-step process to retrieve results for the user, involving user interface management, query formulation, data management and results ranking. This paper will focus on the data management subset of the full exploratory search system. Flight data is currently not available in the form required to make keyword based queries. The system solves this problem by linking destinations with relevant keywords and flights and allowing this data to be queried.

The system aims to initialize a base database with a limited amount of keyword-destination pairs and build upon the initialized data dynamically by using user input. The system also aims to link destinations with images from Dbpedia as well as flights. This is important because the system requires the data to be available in a form that can be queried through a single request. Results from this request are required to contain flight data, destination data and keyword relevance data. Flights, destinations and keywords are separated into separate collections. Each destination-keyword pair is stored in its own document with a weighting indicating relevance. The system is tested using unit tests as well as use case scenarios to test run-time functionality.

The system was requested by and developed for the flight company Travelstart [2] who aim to broaden their user-base by adding a unique exploratory option to their website for users searching for holiday destinations.

2. RELATED BACKGROUND

Searching has become a fundamental application in web-based development due to the popularity and growth of querying the web [3]. One form of searching which involves a high level of learning and investigation by the user [3] is exploratory search. Different aspects of setting up an exploratory search system are investigated, including faceted search, the conflict between user queries and stored categories, the storage structure and the difference between the available database technologies.

2.1. Exploratory Search

Due to the webs exploratory nature, people are now using strategies that involve navigation and trial and error to locate information that they are searching for. Exploratory search can be seen as the combination of both querying and browsing [3]. where the target they are querying may be undefined [7]. Search engines tend to work well when the user's information needs are well defined but are inadequate when users lack knowledge to make an educated query [7]. Users tend to begin making investigative queries to gather information on their desired topic and then reformulate their queries multiple times [8] before exploring results [7].

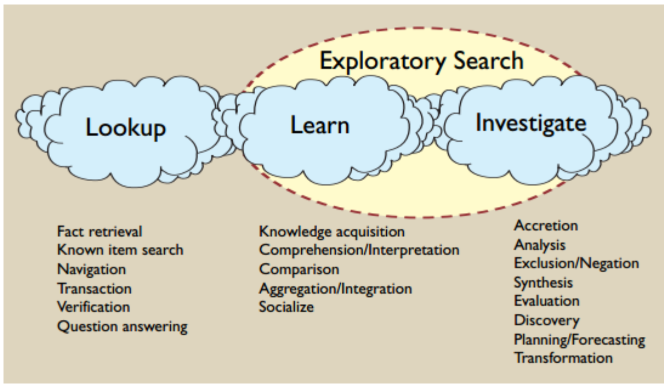


Figure 1: Exploratory Search Activities [6]

Exploratory searches tend to enclose learning and investigation activities as depicted in Fig. 1. Exploratory systems allow users to be active in the processing of their query. The ClearMap framework is one such system which allows users to visually see the processing of their query and manually change the way the underlying data is processed [8].

2.2. Development Technologies

MongoDB is an open source NoSQL database and the leader in the market [10]. It is a non-relational database that was created to handle unstructured data. While relational databases have been around since the 1960's, new types of NoSQL databases are being developed and are being used by an increasing number of developers [13]. The performance of NoSQL databases has been compared to that of SQL databases [11]. Without predefined Schema, NoSQL databases like MongoDB can allow developers to make changes in real time without affecting the end-users experience. Yogesh Punia and Rinkle Aggarwal [11] compare the insert time of records in both an SQL and NoSQL database. MongoDB performs considerably better for larger amounts of records [11]. It can however be seen that when considering update and lookup intensive databases, the speed overhead of SQL is more due to the implementation of its ACID properties [12].

Node is a JavaScript back-end programming language. Node is event-driven and non-blocking making it highly efficient [14]. Some of the major node packages available include Mongoskin [15] for database communication and Sparql-Client [17] for Dbpedia queries.

2.3. Faceted Search

Faceted search can be seen as a combination of direct search and navigational search that attempts to solve the problem of user understanding [6]. Direct search offers users a text box to enter data into and performs a lookup on the exact words but offers no options for refinement. Navigational search provides a hierarchy of content but is unable to classify content that does not easily fit into that hierarchy. Categories can be displayed as separate facets and these facets can even be displayed to the user in the form of results [5]. Each facet can have multiple values and values can fit under more than one facet [16]. An example search with the keywords "Sunny" and "Romantic" would find results under the categories <Romantic> and <Sunny>. Faceted search systems can be combined with query-based systems to allow for searching with the expressiveness of queries [9] but with the simple storage of data under categories.

2.4. Conflict between user queries and stored categories

With regards to the vocabulary problem, users often do not understand the way the information has been stored in the facets created by the developers [5,6]. As an example, a user might make the search: "Warm Romantic Getaways". The system will find no facets that exactly match this search but might break the search down into the following facets: <Romantic> and <Warm> and provide options from those facets instead of the intersection between those facets. A simpler and more common error would be a search for "Seasonal Romantic Getaways" where the system has no facet for <Seasonal> even though the user would probably be perfectly happy with the facet <Warm>.

As seen, the items that the end user wants to see may not be what the developers have under their facets due to different underlying data storage assumptions. One suggestion to this problem is to approach the problem from a user-centric point of view where facets are found for the specific users and then these facet categories are mapped to the underlying ontologies [5]. An experiment was done to test this theory using the card sorting method [4]. Volunteers are given cards with the names of single items on them. Users group the cards into piles based on similarity, these piles are then used as the basic facet groupings to map to the systems underlying ontology. This card sorting method displays that there are multiple right answers for sorting results into different categories and that different users will sort the same items differently. As an example in the experiment [4], two participants grouped different given words under categories. Participant A grouped the words smoking, drugs and tobacco under Intoxicants while participant B grouped these words under Stimulants. Both categories are correct but had the system been designed around participant A's choice, participant B would not be able to find for example "smoking" by searching stimulants. As such it can be noted that the system cannot be designed around a single person's categorization of an object.

This problem is also addressed through dynamic category sets [6]. These dynamic category sets are aimed to address 3 problems. Firstly they must be data driven: they do not provide empty results;

if a search results in two facets where the intersection is null then they must be omitted. For example a search for “Romantic family getaways” where the facets <Romantic> and <Family> exist but there are no destinations containing both those facet-types. Secondly the dynamic sets must address matching semantics, either matching all query terms to facets or if that is impossible, matching partial queries with facets. Lastly dynamic category sets must not provide more specific results than necessary. As an example a user searching “Warm getaway” should only match the facet <Warm> and not include the facet <Romantic>.

2.5. Document Structure: Embedded and Referencing

Data is stored as JSON objects in MongoDB called documents. Referenced data stores relationships between documents by including links to each other [20] while embedded data stores all the information in a single document. Embedded data makes for faster queries as each query only has to make one call to the database. Referenced data on the other hand can save space if the referenced data is used many times.

3. SOFTWARE METHODOLOGY

An agile approach was used in the development of the software as depicted in Fig. 2. A large focus of the approach was the development of working software. Each iteration of the data management system worked, with changes and additions being made each iteration. Every 2 weeks, the separate components of the system were integrated and validated to be working. Unit testing was done on each function of the data management system throughout the development process.

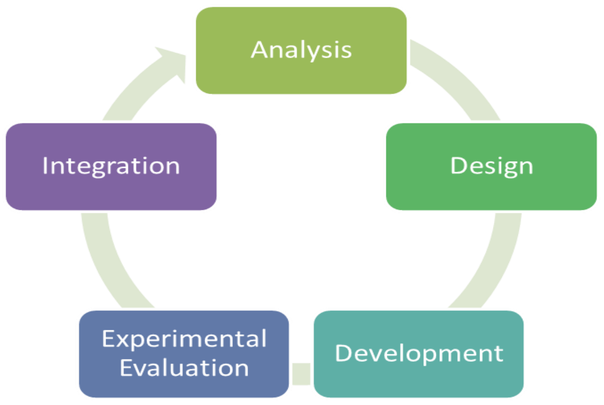


Figure 2: Agile development approach

3.1. Requirements Analysis

The development of the query based search system needed to meet multiple requirements given from the company Travelstart, with the main focus on keyword management and keyword based queries. Additional requirements were added in early iterations to add to the practicality of the system.

3.1.1 Travelstart requirements

1: Keyword based queries: A user may make a query to the system that includes keywords. The system should allow the user to enter 0 or more keywords and search each of the keywords and return each destination code associated with each keyword by a weighting value as well as available flights for each destination. Priority = high.

2: Additional destination information query: A user may query the system for additional information on a list of given destination codes. The system will search for each destination code and return full information on each destination. This includes: country, location name and an image url. Priority = medium.

3: Basic queries: A user may make a query based on a destination, date or price range. The system will return results that meet at least one of the entered requirements along with flights. Priority = medium.

4: Fetch flights: The system should be able to link to the Travelstart API to request flights for selected destinations. Priority = medium

3.1.2 Additional requirements

1: Initialization of destination information: The system will populate a pre-made database with destination information including the country, name and an image url queried from Dbpedia. Priority = medium.

2: Update keyword relevance: The system should be able to increase the keyword relevance for any destination when that destination is selected by the user or searched by the user along with a keyword. Priority = high.

3: Add new keywords: The system should be able to add new keywords to the database for a specific destination through user input. This occurs when a user enters a keyword for a selected destination. Priority = high.

4: Keyword weighting normalization: The system will normalize and dampen the growth of keywords when keywords deviate too far from the mean. Priority = high.

3.2. Design

The entire system is designed with a model view controller architecture where the view consists of the User interface, the model includes the formulation, retrieval and ranking functions and the controller consists of a routing setup for integration. The overall system design is discussed but the main focus is on the data management system used for initialization, insertion, retrieval and maintenance of flight, keyword and destination data.

3.2.1 Overall System Design

Users are able to enter their query in a search bar along with dates, a price range as well as an origin location.

Once the query has been entered, the system uses a 3 step process to retrieve relevant results, this is depicted in the system architecture diagram (see Fig. 3) . The initial step, is the query formulation of the user entered query. The query is broken down, stop words are removed and entered keywords are checked against a dictionary of words as well as their synonyms and alternative forms. The resulting query consists of a list of keywords and an origin location as well as an optional list of locations, price range and dates.

The second step involves the retrieval of flight and destination data from the data management layer. Destinations are matched against keywords and each destination-keyword pair is given a weighting indicating the relevance of the keyword to the location. The query that has now been formulated queries the database for any matching keywords, price ranges and locations and sends through detailed destination and flight information through to step 3 where ranking takes place.

Destinations are ranked based off of keyword relevance, amount of keywords entered matching a location, flight prices and user entered destinations. These ordered results are then displayed to the user.

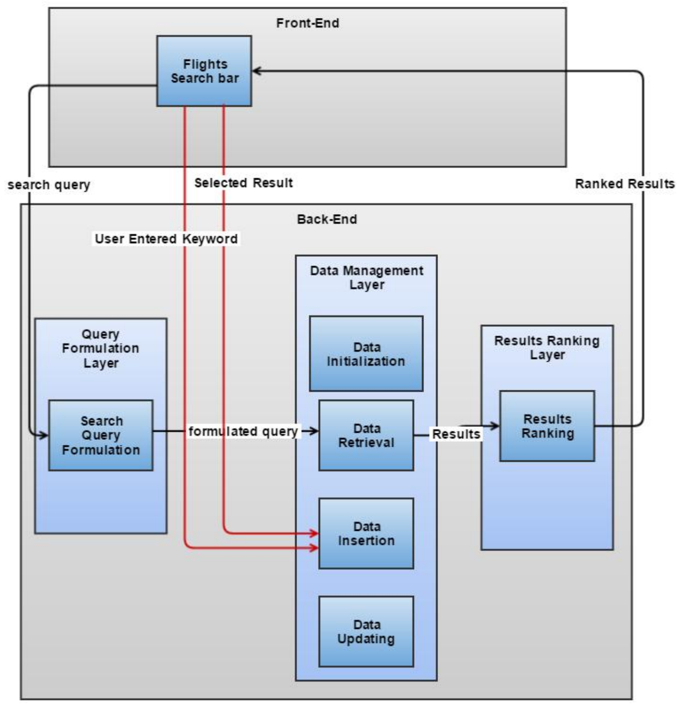


Figure 3: System Architecture Diagram

3.2.2 Data Management System

The system is designed with 4 main features in mind: initialization, retrieval, insertion and updating of location and flight data. Each system feature serves as a list of functions that can be utilized to access and change the stored data. The data management system aims to automatically learn new keywords from current users that can later be explored by future users as opposed to ClearMap [8] which allows each user to define the way in which they search. The system also aims to follow in the footsteps of query-based faceted search systems which allow for the expressiveness of queries [9] with the simple storage of data under categories. The system was built using Node JavaScript due to it being event driven and incredibly efficient. Unit tests for the system were done using Mocha [18] and Chai [19]. Mocha as the test framework and Chai as an assertion library.

A use case diagram shown in Fig. 4 is used to display the user interaction with the system. Users are able to enter queries, of which they can enter a string with keywords, a price range and a location, only one of which is required to make up a valid query to the system. Users can view flights as well as select a flight to view more information. Users can enter a relevant keyword for a destination they have selected. The system needs to be capable of handling these different requests from the user.

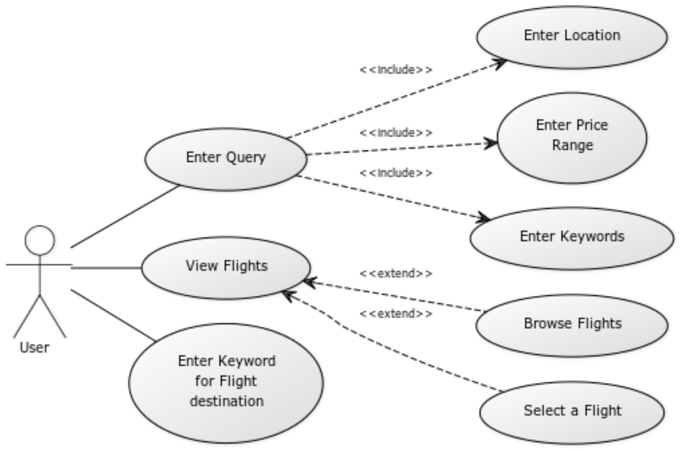


Figure 4: System Use Case Diagram

3.2.2 Use Case Scenarios

It is important to capture the dynamic behavior of the system and as such use case scenarios are used to capture system behavior while it is running. These scenarios attempt to mimic possible system behavior as if the system were live.

Scenario 1: A user makes a search query with the keyword <Romantic>. The user receives the results of the romantic locations with the highest weights which include flights to the locations: Paris, Athens and Rome.

Scenario 2: The user selects Paris as one of the results returned by the system when searching with the keyword: ‘romantic’, incrementing the weight value of Romantic-Paris by 1 point.

Scenario 3: The user enters a new keyword: “beautiful” along with the location Paris in their query. The system adds the new keyword to the database with a weight of 2.

3.2.2 Long-term Scenarios

While capturing the dynamic behavior of the system for single use case scenarios is important, it is also important to analyze the systems expected changes with multiple insertions and retrievals. These scenarios aim to show how the system changes in the long-term.

Scenario 1: Multiple users enter keywords for the location: Paris. Most users choose to enter the keyword <Romantic> while some users choose to enter the keywords <Fun> and <Exciting>. Even fewer users choose to enter the keyword <Dull>. The system updates the weights of each keyword when entered and adds new keywords to the database. The weightings of the keywords are normalized and dampened at the end of the day due to the standard deviation being higher than 50. The common, uncommon and rare keywords are now pushed closer together.

Scenario 2: Users enter a keyword that finds the location Paris or Dubai or search either directly by entering the location. Each time the user enters their own keyword for the location. Paris is more popular that Dubai. The system adds new keywords to the database and updates the weights of existing keyword-destination pairs. The database grows fast at first with the new keywords and slows down as more and more insertions contain already stored keywords.

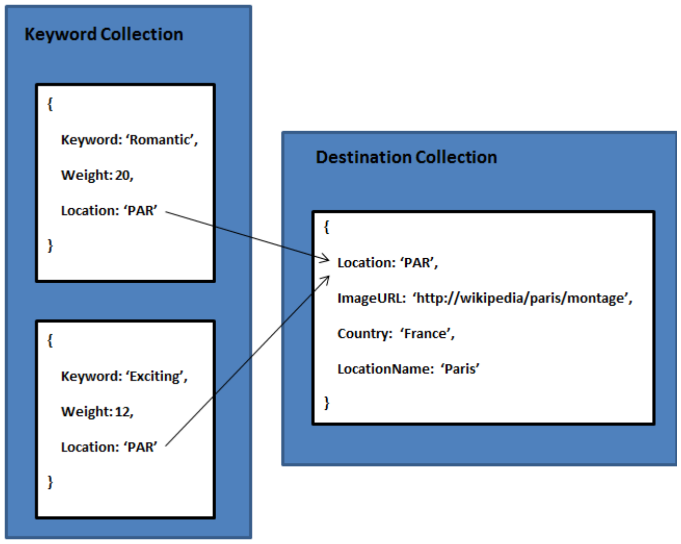
3.2.3 Previous Processing

The current system assumes that query formulation has already been done on the user query and that the request comes in the form of a Json object with a list of keywords, locations, dates and price range. Any of the lists may be empty but at least one is expected to contain information to retrieve relevant results.

Wordnet and a list of common English words was used to define keywords [21]. Only keywords in the word list could be sent through to the system and similar words are mapped to a single word in the wordlist using Wordnet. This creates a limit to the amount of keywords a location can have as well as prevents the system from automatically accepting unwanted keywords such as certain jargon that would bloat the system. It also to a degree prevents storage of multiple keywords that mean the same thing.

3.2.3 Data Storage

All data is stored in MongoDB collections. These include a collection for routes, keyword-destination pairs, destinations and flights. Due to the large amount of repeated destinations in the destination collection, data has been stored using a referencing format where destination data is referred to from its own collection as opposed to embedding the data in the keyword document (Fig. 5).



```
{
  Keyword: 'Romantic',
  Weight: 20,
  Location: 'PAR'
}
{
  Keyword: 'Exciting',
  Weight: 12,
  Location: 'PAR'
}
{
  Location: 'PAR',
  ImageURL: 'http://wikipedia/paris/montage',
  Country: 'France',
  LocationName: 'Paris'
}
```

Figure 5: Data Storage Structure

This is done to prevent the keyword collection from having too much repeated data that would then have to be individually updated with every destination update. This helps keep the data consistent and drastically reduces document size. If embedded data was used, an extra 3 elements would need to be stored with each new keyword (image url, country and location name) as opposed to just once for the destination. We tested the effects on the performance of the two options and even though retrievals need to make extra calls to the database to fetch the destination information when using a referencing format, the results clearly show that query times are not largely affected (see Table 1) and as such we use a referencing format to reduce document size and keep data consistent. The query times were tested by creating a keyword collection with documents and either referencing destinations or embedding them.

Table 1. Query Times

Number of Documents	Referencing (ms)	Embedding data (ms)
10	34	33
100	38	36
1000	50	47
10000	180	152

The structure of stored data can be seen in the Entity Relationship Diagram (see Fig. 6). All data is stored in separate MongoDB collections represented by the entities in the entity relationship diagram. The most used collection being the keyword collection where keyword-destination pairs are stored with a weighting indicating the relevance of the keyword to the destination. Additional destination data is stored in the destination collection including an image url of the location, the country the location belongs to as well as the location's full name and the code it is stored under. These codes are unique and are used to link destinations with keywords as well as flights. Each flight has an origin and a destination. At the moment only 3 origins are used: Johannesburg, Cape Town and Durban. This was done for testing purposes.

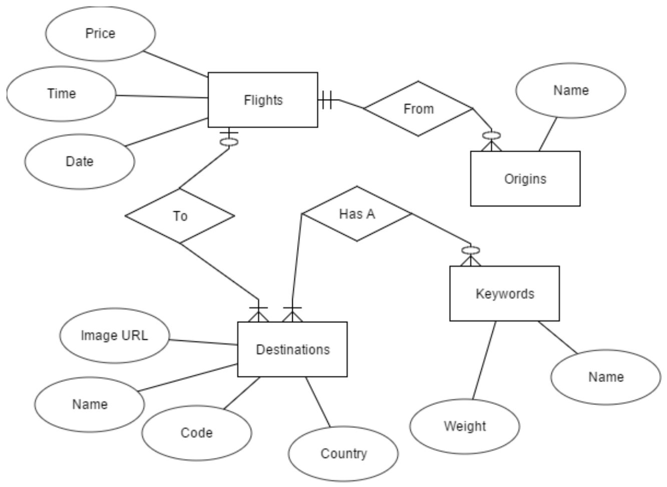


Figure 6: Entity Relationship Diagram

3.3. Development

3.3.1 Data Initialization

A large part of the project involved gathering and mapping the data to Json objects to be stored in the database. When searching for flights, all destinations are identified by a unique 3 letter code. Destination codes were scraped from the Travelstart website and stored with their country and full name. Flight routes were stored in a separate collection.

The keyword collection was created with a small amount of manually created data that is used to initialize the system.

Location names are used to query Dbpedia for urls of image montages of the location through a union of select statements. Dbpedia only accepts 200 select statements in a single query. To gather data on all 600 locations, a recursive function is used that splits the data into sets of 200 select queries and then queries one set at a time, waiting for the previous set to complete before continuing. These depiction urls are stored with the relevant destination in the destination collection. About 70% of destinations queried from Dbpedia contain a relevant image. This is due to an inconsistency with the storage of location depictions on Wikipedia. Most locations have a montage stored as their depiction but some have individual images placed close together as opposed to a single image montage. When this occurs, an image from the page, often the location's flag is stored as the depiction instead.

All initialization functions are only run on the first database population. When an update is needed the relevant databases can be dropped and will re-initialize automatically.

3.3.2 Difficulties encountered and changes in strategy

The system has included a few limitations to allow for rapid development and testing as well as limited by the amount of data available currently from Travelstart.

Difficulties were encountered in the keyword initialization process. The initial plan was to use keywords from Tripadvisor and Wikipedia. It was however noted during system design that the Tripadvisor destinations were too specific (specific locations as opposed to cities) when compared to the city names given by Travelstart. Wikipedia keywords were too general to describe the location and were not well suited to describing holiday destinations. To solve the keyword seeding, the amount of available destinations were limited to: Amsterdam, Paris, Dubai, Rome, Athens and Bangkok. This allowed keywords to be initialized manually for the locations and seed the database to be later built up by user-input.

Due to Travelstart API issues and multiple Travelstart representative changes, working API usage and support was never available, limiting access to real-time flight data. This was solved through the use of storing fake flight data to fetch and adjusting the project focus more to destinations and keywords. Stored flight data is limited to what is required and as such does not store dates as these are not required for the system to operate.

3.3.3 Keyword Relevance

Keywords needed to be defined in a way that described how relevant they were with a paired destination. To achieve this, all keyword-location pairings also include a weight between 0 and 100 that describes how relevant the keyword is to the destination; 100 indicating most relevant. The idea behind the weighting is that higher ranked keyword-location pairs will be shown to the user before lower ranked pairs. This in combination with the number of relevant keywords as well as matching flight dates and prices can be used to rank resulting flights for the user.

3.3.4 Updating keyword weightings

Keyword weightings need to change when users search and interact with the results from the system. This is done by increasing destination-keyword weightings.

There were 4 possible options for changing the weights of keywords, 3 of which can be seen in the use case diagram: entering a location as well as a keyword in a query, selecting a flight destination after entering a keyword and entering a keyword for a destination (see Fig. 4). Firstly, all manually created keywords are inserted into the database with a starting weight of 1. The remaining options are broken up into their methods which can be called directly.

Update weighting method: This is the first use case path when a user searches a keyword and then selects a destination. In this case the weighting of the searched keyword is increased by 1. This is because by selecting one of the destinations, the user confirms that this is a result that should tie to the entered keyword.

Retrieval method: This is when a user enters a location and a keyword in the initial query, we can immediately increase that keywords weighting by 1 to the entered location. This is done because by simultaneously entering a keyword and a location, the user confirms that they believe this keyword and location to be related.

User update weighting method: This is the only method that involves users actively entering a new keyword for a location. It occurs after searching for a destination, a user enters a keyword for that location through a pop up box asking them to do so. In this case the weighting for the location-keyword pair is increased by 2 as opposed to 1 due to the user actively deciding on a keyword that is related to the destination. This last case encourages the underlying facets of the system to be user-centric [5].

Leaving weightings to continually grow would cause greater and greater gaps between the weightings of popular and less popular keywords as well as cause space requirements of weightings to grow until they can no longer be stored in integers. Weightings needed to be dampened and normalized to control the growth of these keywords as well as lessen the gaps between popular and less popular keywords.

The normalization and dampening of the weightings is done automatically and once per day. For each location, the maximum weight is found and stored as well as the mean weight for each location. While the data needs to be normalized if weightings are greater than the maximum weight, data does not always need to be dampened if keywords are already close enough. The standard deviation is calculated for the set of locations and if one location weight deviates by more than 50% of the maximum keyword weight, the keywords are dampened before normalizing.

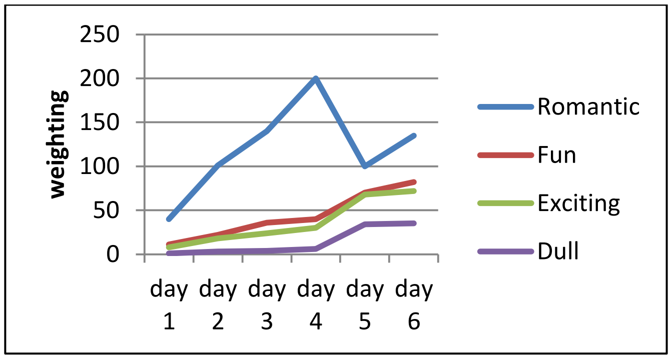
Dampening was included in a later iteration when it was noticed that common keywords would become overly dominating. Dampening is done with a log function. When dampening, the normalization value n is calculated by dividing 100 by the log of the location maximum m. When dampening is not required, just the location maximum is used in the calculation (as opposed to the log of). The log function has a larger effect on smaller weights compared to larger weights. This causes larger weights to be dampened. This can be seen in Graph 1 where a test was done on a sample of keywords, on day 5, the data is dampened using the log function and normalized. The graph shows how the strong keyword "Romantic" is dampened to be closer to the other keywords.

$$n = \frac{100}{\log(m)} \text{ or } n = \frac{100}{m}$$

Each weight w for each keyword for the location is multiplied by this normalization value. Again if dampening is required, the log of this weight is used when multiplying. The new weight W, replaces the old weight.

$$W = \log(w) \cdot n \text{ or } W = w \cdot n$$

After this process, all weights are now between 0 and 100 and the standard deviation of all keywords is lower, popular keywords are still retained, but less common keywords are now more visible.

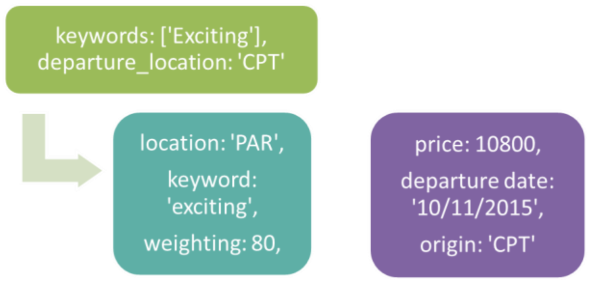


Graph 1: Dampening and normalization of keywords

3.3.1 Database Querying

The system was designed to allow for function calls to perform various queries and insertions to the underlying Mongo database.

The system offers a retrieval method that accepts a JavaScript object as a query parameter. This object is expected to contain at least one of the following: a keywords list, a locations list, a departure date, an arrival date, a minimum price and a maximum price. A departure location is needed for every query. The system will query the keyword collection for keyword-destination pairs matching the query, the destination-collection for matching destinations and the flights collection for a matching price range. The system will return a JavaScript object containing a list of matching keywords-destination pairs, locations and flights. The example seen in Fig. 7 shows the first resulting flight and destination in an array returned for the query containing the keyword: "Exciting".



keywords: ['Exciting'], departure_location: 'CPT'
location: 'PAR', keyword: 'exciting', weighting: 80,
price: 10800, departure date: '10/11/2015', origin: 'CPT'

Figure 7: Keyword Query Result

A method is available to get additional information from keyword-destination pairs. This is due to this pair not containing any information on the destination other than the 3 letter location code. This information is not returned immediately with the pair as it is not needed for any form of ranking and will only need to be queried for results that have been chosen to be displayed to the user as opposed to retrieving this information for each destination that matches the query only slightly. This method accepts a query in the form of a list of destination ID's and retrieves information from the destination collection.

The last two methods involve updating the keyword-destination weightings. Both methods accept a query in the form of a list of keywords and a location. The location has it's corresponding keyword-destination pair updated/created with each keyword. The first method being when a user selects a destination after entering a query, the weighting updates by 1 and the second being when a user enters a keyword on request for a destination, updating the weighting by 2.

3.3. Experimental Evaluation and Discussion

The system is evaluated through the mentioned use-cases to show run-time functionality as well as unit tests to show that functions work and show database consistency.

3.3.1 Unit Testing

Mocha is used to write unit tests for each function to show correctness and reliability. Each function is run using a test database, filled with test data that resembles real-life data but is altered for the purpose of the test. Functions are generally checked to behave correctly when given different forms of relevant data as well as edge cases such as empty queries or incorrect queries.

As an example, one of the unit tests is used to check the method that is used to extract destination data from a text file and return it in the form needed to post to the database. The unit test checks that the method is a function, it returns an array, it can handle empty inputs it returns a correct result and that it can handle errors.

Unit tests are also used to test database queries as well as database consistency. Database methods are checked for retrieving correct results, only adding elements if they do not already exist, not initializing data that is already available and checking that the live database does not contain duplicates.

3.3.1 Use Case Scenario Evaluations

The use case scenarios are also tested using unit tests, in this case the initialized database and queries are very specific to the scenario.

Scenario 1: System retrieve method is called with a query containing only the keyword: "Romantic". System correctly calls the get results method from the retrieval layer and retrieves keyword-destination pairs from the keyword collection. The system then retrieves all flights to each location in the keyword-destination pairs. The flights and pairs are returned to the user.

Scenario 2: The database is initialized with a few keyword-destination pairs, one of which is "Romantic-Paris" with a weight of 1. The update weighting method is called with the query containing "Paris" in the location list and the keyword "Romantic" in the keywords list. The keyword collection update method from the insertion layer is called. The database is checked to see if the weight for the pair has increased to 2.

Scenario 3: The database is initialized with a few keyword-destination pairs, none of which is "Beautiful-Paris". The user update weighting method is called with a query containing "Paris" in the location list and "Beautiful" in the keyword list. The keyword collection update method from the insertion layer is called. The database is checked to see if the destination-keyword pair has been added with a weight of 2.

3.3.1 Long Term Scenario Evaluations

Scenario 1: A database is created with the location Paris in the destination collection. The keyword collection is empty. The user update weightings method is called 100 times with the keyword "Romantic", 20 times with the keyword "Fun", 15 times with the keyword "Exciting" and 3 times with the keyword "Dull". It is important to show the change in weighting when the keywords are normalized to prevent additional growth and to prevent overly dominating keywords. The keyword weighting before and after normalization and dampening can be seen in Table 2.

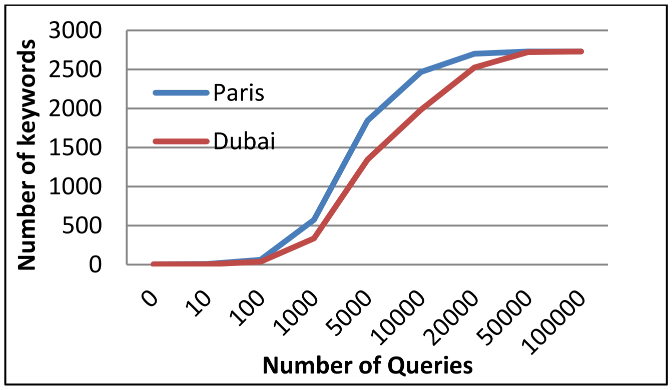
Table 2. Effect of normalization on weighting

Keyword	**Weighting before normalization**	**Weighting after normalization**
Romantic	200	100
Fun	40	70
Exciting	30	68
dull	6	34

The standard deviation was calculated for the set of keywords in Table 2. This shows how much weightings deviate from the mean. The standard deviation before normalization is 88.50 (which would normally cause the automatic normalization and dampening methods to call at the end of the day while live.) After normalization, the standard deviation is 27 which is acceptable. It can be seen that the maximum weighting is once again 100.

In a live test this would be a likely occurrence for a more popular keyword such as "Romantic" to be entered for a location than keyword such as "Dull". It can be seen that the keywords are significantly closer after normalization and that the keywords "Fun" and "Exciting" could compete with "Romantic" more in a form of ranking. This prevents 1 or 2 popular keywords from dominating and preventing diversity. The database could be diluted down to a top percentage of keywords each day to prevent bad keyword choices from polluting the database.

Scenario 2: A limited database is initialized with the destinations Paris and Dubai in the destination collection and no keyword-destination pairs in the keyword collection. The user update weighting method is called 10, 100, 1000 and 10,000 times with a 60% probability of the location being Paris and a random keyword from the dictionary that is used in the query formulation containing 2727 common English words. This long-term scenario models a possible real world scenario where more users search for one location than another and shows the growth of the keywords for these locations, eventually reaching the maximum number of keywords as the dictionary has been exhausted. The growth of the keywords can be seen in Graph 2.



Graph 2: Number of stored keywords with increasing queries

The growth seen can be expected to occur when the system is live. A typical location will grow largely initially and then begin to stabilize as fewer new keywords are inserted. This is seen by running the slightly bias random function with the list of keywords for the two locations. In a real environment, it might take a lot longer for the entire list of available keywords to be exhausted as some of the less common keywords may take a while or never be entered.

3.3.1 Insertion/Retrieval Time Growth

The growth of the insert and retrieval times of a document are checked to test scalability. The original destination collection is used with approximately 600 destinations. The keyword collection is tested for increasing amounts of keyword-destination pairs (see Table 3) created randomly from a word list and available destinations. Both insertion and retrieval times are similar in comparable literature [12]. Retrieval times are manageable up to 100,000 documents where it starts taking around a second to retrieve a document. Insertion times are largely similar to retrieval times. With the limited amount of locations, this growth in number of documents is capped.

Table 3. Insertion/Retrieval times

Number of documents	Retrieval (ms)	Insertion (ms)
10	34	32
100	38	35
1000	50	51
10000	180	168
100000	1293	1363

3.3. Integration

The data management component needed to be integrated each iteration with the other components of the project. These include the user interface, the query formulation and the result ranking. This integration was done through a controller which called the correct methods based on post requests from the front-end. For retrieval, query formulation is called with the initial query data, the keywords are extracted and sent through to database retrieval where flights, destinations and keywords are fetched and sent through to ranking where results are ordered and flight data and destination data are combined. Two additional post requests are included to update keyword weightings when a user enters a keyword for a destination via the UI popup box [22] and when a user selects a destination after previously searching a keyword.

4. DISCUSSION

The experimental evaluation shows that the system is able to support keyword-based queries made by users. This can be seen through the first scenario evaluated as well as the unit tests done on the methods required to make such queries. This solves the problem of exploratory search but is limited in this solution due to the limited amount of initialized keywords as well as the way in which keywords are defined. Only 5 destinations have been initialized manually with a few keywords to seed the growth process. All destinations would need to be initialized with a few set keywords which would require a significant amount of manual research and time or an already set keywords-destinations structure which could be linked with Travelstart's destinations, neither of which were available. Keywords are defined as destination-keyword pairs along with a weighting indicating relevance, this definition does not yet allow for multi-word keywords which limits the user in their exploratory nature to single word keywords. Additional information on retrieved destinations can be successfully queried from the database, meeting the second requirement set out by Travelstart.

All the additional requirements work mostly as expected, these include initialization of destination information, keyword relevance updating, keyword normalization and the addition of new keywords. All are tested through use case scenarios for functionality and through experimental scenario evaluations, both short and long-term. Keywords grow as expected and normalization of keywords successfully eliminates the focus on single strong keywords. The initialization of destination data was unable to gather images for all destinations due to inconsistencies with Dbpedia storage where images were often not stored as a montage but rather as separate images as well as destination names being different on Dbpedia and Travelstart. This lead to approximately 70% of images retrieved being correct, this is an acceptable amount for the project scope.

The system is usable and can be connected to a front-end with query formulation and ranking through routing to allow for a fully working system. The system does not use live flight data due to API issues on Travelstart's side and multiple communication issues. The system would need access to live flight data to provide usable results.

5. CONCLUSIONS

A system was developed that is able to match and update keyword-destination pairs based on user queries and input when searching for flights through an exploratory search system. The system uses only enough destinations, flights and manually created keywords for testing purposes. Unit testing done with Mocha showed that the system functions as expected with both valid and invalid input data.

The system meets all but one of the requirements set out by Travelstart including keyword based queries, additional destination information queries and basic queries. The requirement to present users with real-time flight data was not met due to issues with the available API. All additional requirements were met, the system is able to add and update keyword relevance, as well as normalize stored keyword weightings. It allows for growth of keyword-destination pairs through user input on destinations and is able to control the weightings of these pairs automatically. This shows that there is a plausible exploratory alternative to the way that flights are currently searched for (requiring specific information on flight dates and destinations). The additional destination information requirement was met but with only 70% of locations containing a valid image due to inconsistencies with Dbpedia storage and differences in name conventions between Dbpedia and Travelstart. The retrieval and insertion times for the keyword based queries requirement was tested and are low (less than 1 second when the keyword collection contains 100,000 documents or less).

Use case scenarios were used to test and experimentally evaluate the run-time functionality of the system. Scenarios were also used to observe the change in keyword weightings when undergoing the logarithmic normalization function. This showed that the function was able to bring the standard deviation between weightings down considerably when needed as well as to bring keywords closer together for diversity. Another scenario showed the growth of number of keywords for a destination, showing that more popular destinations would grow faster but both popular and less popular destinations would see their growth in keywords stagnate due to the limited number of keywords in the common English words dictionary.

There are a few areas where this system could grow with future work. When querying Dbpedia for images, one could choose other images if a montage is not available, with possible checks to make sure it is not getting a flag or coat of arms. Additional data from Dbpedia could also be stored, currently only the image url is used but other information such as population, history and description could also be stored. In terms of keywords, currently only single word keywords are supported, but multi-word keywords would be incredibly useful. As an example Paris could be associated with the keyword "Eiffel Tower" or London with "Big Ben". These multiword keywords offer more complex descriptors. A great deal of extra data could also be gathered if the system went live, data on popular keywords and actual growth as opposed to the growth observed by use case scenarios could also be reported and analyzed.

5. REFERENCES

[1] Kules, B. Capra, R. , Banta M. and Sierra T. What do exploratory searchers look at in a faceted search interface? *9th ACM/IEEE-CS joint conference on Digital libraries*, (Austin, USA, 2009), ACM New York, 313-322.

[2] Cheap Flights & Air Tickets from Travelstart South Africa: 2015. https://www.travelstart.co.za/. Accessed: 2015- 10- 19.

[3] Marchionini G. Exploratory search: from finding to understanding. *Communications of the ACM*, Vol.49 (4), 2006. 41-46.

[4] Gordon Rugg and Peter McGeorge. "The sorting techniques: A tutorial paper on card sorts, picture sorts and item sorts", *Expert Systems.* Vol. 14 (2), 1997, 80 -93.

[5] Suominen, O., Viljanen, K. and Hyvänen, E. Franconi, E. User-Centric Faceted Search for Semantic Portals. Kifer, M., May, W. (Eds.) *4th European Semantic Web Conference, ESWC*, (Innsbruck, Austria, 2007), Springer Berlin Heidelberg, 356-370.

[6] Tunkelang, D. Dynamic category sets: An approach for faceted search. *ACM SIGIR Workshop on Faceted Search*, Vol. 6, 2006, 1-5.

[7] White R., Kules, B. and Bederson, B. Exploratory search interfaces: Categorization, clustering and beyond. *ACM SIGIR*, Vol. 39 (2), 2005, 52-56.

[8] Bao, Z. Zeng, W. and Ling T. Exploratory Keyword Search with Interactive Input. *The 2015 ACM SIGMOD International Conference on Management of Data*, (New York, USA, 2015), 871-876

[9] Ferré, S. and Hermann, A. Semantic Search: Reconciling Expressive Querying and Exploratory Search. Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (Eds.) *10th International Semantic Web Conference Part 1*, (Bonn, Germany, 2011), Springer Berlin Heidelberg, 177–192

[10] MongoDB for Giant Ideas. MongoDB: 2015. https://www.mongodb.org/. Accessed: 2015- 10- 18.

[11] Punia, Y. and Aggarwal, R. Implementing Information System Using MongoDB and Redis. *International Journal of Advanced Trends in Computer Science and Engineering*, Vol. 3 (2), 2014, 16 – 20.

[12] Stonebraker, M., SQL databases v. NoSQL databases, *Communications of the ACM*, Vol.53 (4), 2010, 10-11.

[13] Leavitt, N. Will NoSQL Databases Live Up to Their Promise?, *Computer*, Vol.43 (2), 2010, 12-14.

[14] Node.js: 2015. https://nodejs.org/en/. Accessed: 2015- 10- 18.

[15] Node-mongoskin: 2015. https://github.com/kissjs/node-mongoskin. Accessed: 2015- 10- 18.

[16] Koren, J. Yi, Z. and Xue, L. Personalized interactive faceted search. *the 17th international conference on World Wide Web*, (2008), ACM, 477-486.

[17] Sparql-client: 2015. https://github.com/ruby-rdf/sparql-client. Accessed: 2015- 10- 18.

[18] Mocha - the fun, simple, flexible JavaScript test framework: 2015. https://mochajs.org/. Accessed: 2015- 10- 18.

[19] Home - Chai: 2015. http://chaijs.com/. Accessed: 2015- 10- 18.

[20] Anuradha, K. Arpita, G. and Shantanu, K. A study of normalization and embedding in MongoDB, *Advance Computing Conference (IACC), IEEE International*, (2014), 416-421.

[21] Salie, L. Travel search: query formulation and expansion, 2015

[22] Choga, N. Travel search: user interface design and evaluation, 2015