Image Content Search Engine

CS 242: Information Retrieval and Web Search
Department of Computer Science and Engineering
Bourns College of Engineering
University of California, Riverside

Karthik Harpanahalli
kharp009@ucr.edu

Varun Sapre
vsapr002@ucr.edu

Hoora Sobhani
hsobh002@ucr.edu

Ameya Padole
apado003@ucr.edu

Sarthak Jain
sjain050@ucr.edu

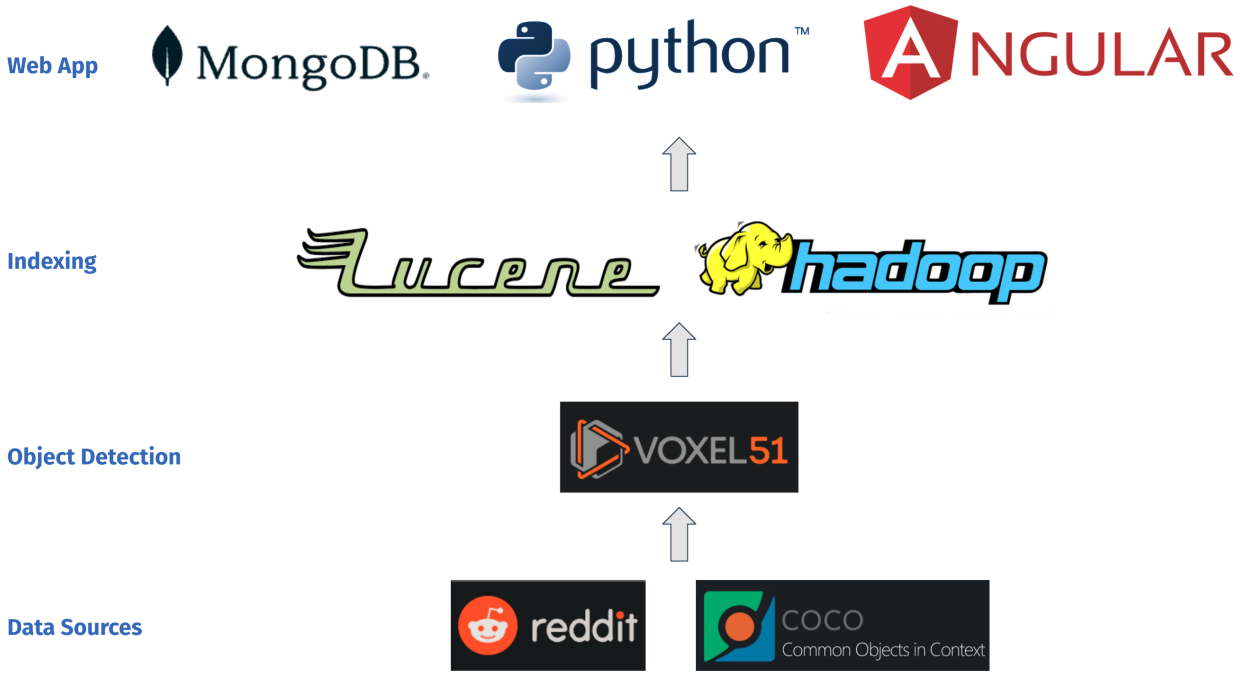
Introduction:

We designed and developed an image search engine using Hadoop Inverted Indexes and Lucene indexing which returns the images pertaining to the input query by the user. We employ FiftyOne API - an object detection software - to detect objects in an image and use the indexed objects called annotations to match with the query. These images are ranked for relevancy using the area of bounding boxes of each annotation and then returned to the frontend to be displayed on the browser.

Collaboration Details:

1. Ameya Padole - Hadoop Ranking algorithm creation
2. Hoora Sobhani - Lucene Indexer and Searcher Implementation
c. Karthik Harpanahalli - Front-End development in AngularJS
d. Sarthak Jain - Hadoop Inverted Index creation
e. Varun Sapre - Back-End development using Python and MongoDB

Architecture:



Hadoop MapReduce:

1. Details of how Hadoop was used:
    - We used MRJob which is a python library that runs Map-Reduce Jobs.
    - We sanitized our input JSON file and made sure to have one record per line to stream it to our mappers.
    - We included annotations as Supercategory, Category_ID and used them as key.
    - We yielded the keys with the image ID as value from the mapper to the reducer.
    - The reducer yielded the key with the list of IDs as output.
    - This output was written to another JSON which was used by the front-end to fetch the relevant images

2. Explain and justify the indexes built by Hadoop

    - The inverted index build using Hadoop MapReduce helps us retrieve all the images for a given annotation. We did not need to stem these annotations as they are already the most relevant words that will be used for matching the search query
    - To make the supercategory searchable, we had to make sure that it was used in the mapping as its own word so that if the search query had a vague supercategory searched, we would still be able to retrieve images.

3. Ranking algorithm for Hadoop Inverted Index results

    a. From the Hadoop inverted index, we know that we have category and supercategory as the keys with a list of the IDs related to them in the index.

    b. When we receive the search query from the user, we use this query to lookup the index for a matching key

    c. Upon finding a match, we retrieve the list of annotation IDs from the index and use these IDs to retrieve the annotations and images.

    d. Each annotation has a pre-calculated field called $areaRatio$ which is calculated by a simple formula:

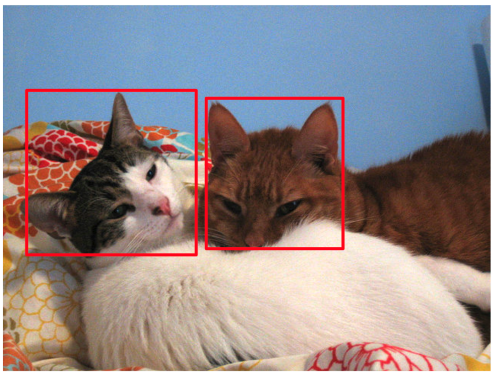
$$areaRatio = (\frac{area\ of\ bounding\ box}{area\ of\ image})$$

    e. The reason we take this $areaRatio$ into account is because if an annotation area is very small compared the area of the entire image, we can conclude that it is not a significant object in the image. On the other hand, if the $areaRatio$ is large, we know that the object is a large portion of the image and is thus more relevant. The images below show the difference.

    f. For each image, we calculate the summation of the $areaRatio$ of the annotations that match the search query. We do this because having multiple objects that match the search query are more relevant and should be ranked higher.

    g. After ranking, we return the ranked list of images with the object in it.

Highest Rank



Lowest Rank

4. Implementation of MapReduce Functions:

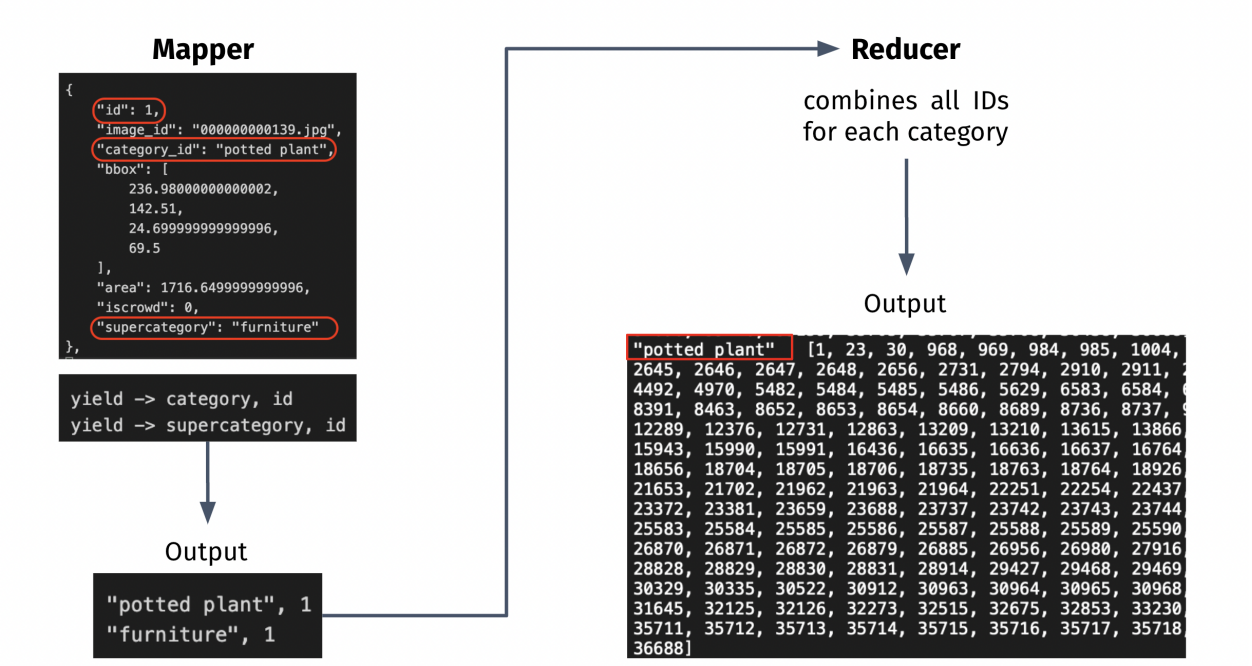
    a. MRJob :

    `mrjob` is a python library that lets developers implement Map-Reduce. It enabled us to write multi-step MapReduce in Python3 and test the same on our local machine.

```
from mrjob.job import MRJob
from mrjob.protocol import JSONProtocol
```

    b. Map and Reduce functions

```
class Invertedindex(MRJob):
    def mapper(self, _, record):
        # each record is a json object
        data = json.loads(record)

        # retrieve relevant fields from data
        id = data['id']
        category = data['category_id']
        supercategory = data['supercategory']

        yield(category,id)
        yield(supercategory,id)

    def reducer(self, annotation, ids):
        yield(annotation, list(ids))
```



Lucene Indexing and Searching:

As mentioned in part A, our Lucene implementation is based on the `org.apache.lucene` Java library which is robust and well-documented. In this report, we briefly overview our Lucene indexer and we mainly focus on our Lucene searcher and how it returns the results to the front-end.

Lucene Indexer:

The input of our indexer is the JSON file which is the output of our crawler with multiple modifications to be easy to be parsed. To parse the inputs, we used `org.json.simple` Java library. The main implementation of our indexer is in Indexer.java. We considered each image annotation as a document and stored the id, image_id, category_id, and super-category. We exploited the standard analyzer of the Lucene and by using `text-field`, we enabled prefix matching and token matching. The indexed data is in the `data` directory which is in a parallel directory with `src`.

Lucene Searcher:

Although we had a search test in part A, we reimplemented our Lucene search as a separate class to be functional on the multi-field search and also to be easily compatible with our front-end. The main implementation of our searcher is in Searcher.java. You

can find the `searcher` directory in parallel with the `indexer` directory inside the `src`. We will elaborate on each part of the implementation of class Searcher in the following section.

- **Searcher Setup:**

We set up our Lucene searcher to read the data from our index inside the `data` directory. We designed our search engine such that users can search one “category” and “super-category” names. Since we have possible searches on multiple fields, we exploited the `MultiFieldQueryParser` class to analyze our query.

```
Directory directory;
DirectoryReader indexReader;
IndexSearcher indexSearcher;
MultiFieldQueryParser queryParser;
Query query;

private static  String[] fields = {"category_id", "supercategory"};
private static int topHitCount = 250;


public Searcher() throws IOException{
    this.directory = FSDirectory.open((new File("../../data/")).toPath());
    this.indexReader = DirectoryReader.open(directory);
    this.indexSearcher = new IndexSearcher(indexReader);
    this.queryParser = new MultiFieldQueryParser(fields, new StandardAnalyzer());
}
```

- **Searcher Close:**

As a requirement, we need to close the index reader and directory object we opened.

```
public void close() throws IOException{
    this.indexReader.close();
    this.directory.close();
}
```

- **Searcher Search:**

In our search function, we receive the user query as a String and then parse it with our `MultiFieldQueryParser`. We used the default ranking of Lucene search which is based on TF-IDF. However, in the Hadoop section, you can see our image processing ranking algorithm returns much proper images rather than TF-IDF of image annotations. Since it searches on multiple fields, each image-id may repeat in our `results` arrayList. Therefore, we used a `LinkedHashSet` to return unique image_ids while preserving the rankings.

```
public Document[] search( String searchQuery) throws IOException, ParseException {

    long startTime = System.currentTimeMillis();
    //System.out.println("#Search in Documnets: Elapsed time (ms)");

    this.query = queryParser.parse(searchQuery);
    //System.out.println(query.toString());

    ScoreDoc[] hits = indexSearcher.search(query, topHitCount).scoreDocs;
    Document[] results = new Document[topHitCount];
ArrayList<String> image_ids = new ArrayList<String>();

    //Iterate through the results:
    for (int rank = 0; rank < hits.length; ++rank) {
        Document hitDoc = indexSearcher.doc(hits[rank].doc);
        //System.out.println((rank + 1) + " (score:" + hits[rank].score + ")--> " + hitDoc.get("id") + ":" + hitDoc.get("image_id"));
        results[rank] = hitDoc;
    image_ids.add(hitDoc.get("image_id"));
    }

LinkedHashSet<String> unique_image_ids = new LinkedHashSet<String>(image_ids);
export_data_into_json(searchQuery, unique_image_ids);

    long currentTime = System.currentTimeMillis();
    //System.out.println((currentTime - startTime) +"(ms)");

    return results;
}
```

- **Searcher Export Data:**

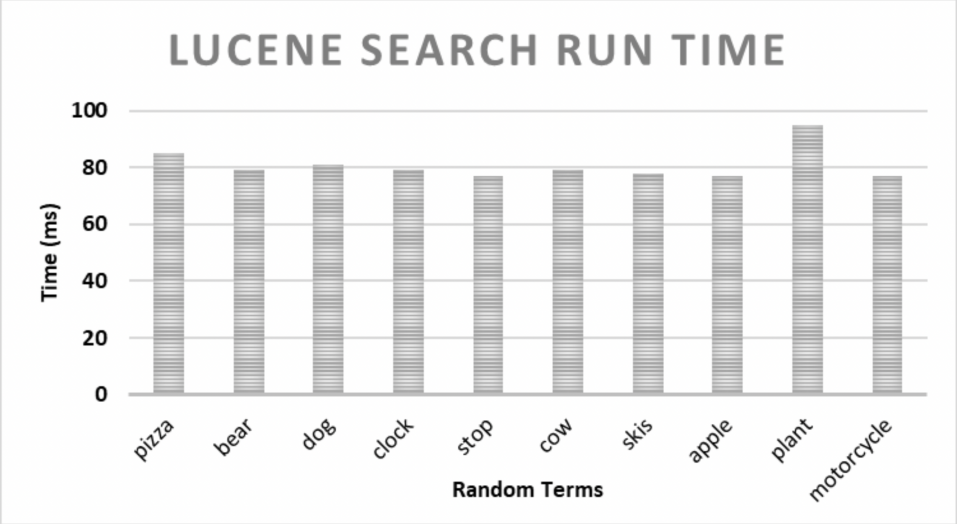
We finally export our search results in a JSON file so that it can be easily used in our web application. We capture a limited number of unique image-ids in a file with the same name as the user query inside the `results` directory. Then, our web application processes this result for each search request. To be fair in timing analysis, we measured the run-time of our searcher class and not considered the processing time of the web application.

```
public void export_data_into_json(String keyword, LinkedHashSet<String> image_ids) throws IOException {
BufferedWriter output = new BufferedWriter(new FileWriter("results/" + keyword + ".json"));
    Iterator it = image_ids.iterator();
int count = 50;
while(it.hasNext() && count > 0){
    output.write(it.next().toString());
    output.newLine();
    count--;
}
output.close();
}
```

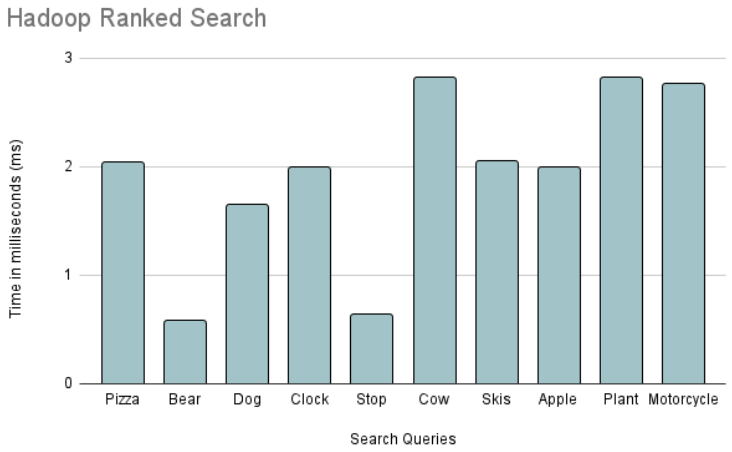
- **Searcher Main:**
  Our back-end wrapper feeds our Searcher class with the user query.

```
public static void main(String[] args) throws IOException, ParseException {

String inputQuery = args[0];
//System.out.println(inputQuery);

Searcher luceneSearcher = new Searcher();
luceneSearcher.search(inputQuery);
luceneSearcher.close();
}
```

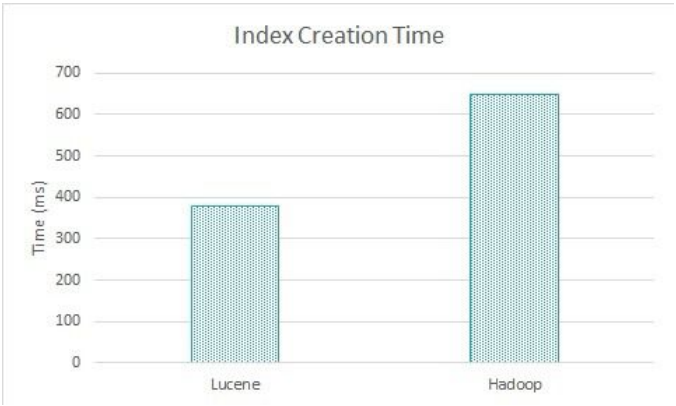
Lucene Search run time:



Hadoop with Ranking Search run time:

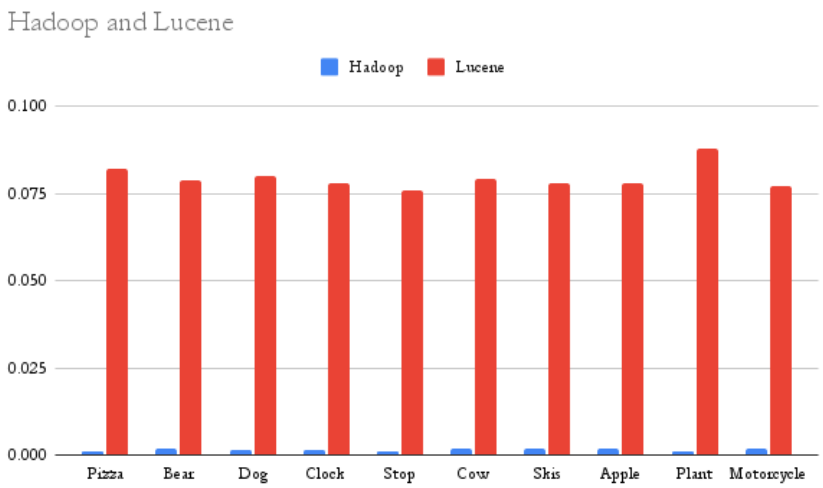


Index creation time comparison between Hadoop and Lucene :



The difference in index creation time we see between lucene and hadoop is likely because of the small overheads of the mapreduce dataflow and also because of the python MRJob pacakge that was used for this.

Comparison between Hadoop and Lucene for index retrieval:



The big difference seen in this graph is mainly because the look up time for hadoop inverted index is constant time for lookup. The timing for hadoop with ranking is different and is mentioned below.

Instructions on how to deploy the system:

- MongoDB Setup:
  - Install MongoDB on system.
  - Uploading Inverted Index:
    - create a database - `use inverted_index;`
    - create collection - `db.createCollection('index');`
    - make sure `inverted-index-whole-words` is present in the directory.
    - run `upload_index_to_mongo.py`
  - Uploading Annotations:
    - use previously created database - `use inverted_index;`
    - create collection - `db.createCollection('annotations');`
    - make sure `export_annotations.json` is present in the directory
    - run `upload_annotations_to_mongo.py`

    - Uploading Lucene Index:
        - use previously created database - `use inverted_index;`
        - create collection - `db.createCollection('lucene_index');`
        - make sure you are in the folder `lucene_index_results`
        - run `upload_lucene_index_to_mongo.py`
- Backend Server:
    - Package requirements:
        - python3
        - flask, flask_cors, pymongo
    - run `python3 app.py`
    - NOTE: IP address that is displayed in the output will be used in the next steps
- Frontend:
    - The zipped file contains the output in `dist` folder.
    - For easier setup, running the code in build environment is recommended.
    - Get the IP address from the backend.
    - Change the `baseUrl` in `app.component.ts` to the current IP of the backend server.
    - Before running install required node files by running `npm install`.
    - Open a terminal in the project folder and run `ng serve` command.
    - An app instance should run on the default browser.

Limitations of System:

1. Lucene Search :
   Lucene's search and ranking are based on TF-IDF, however for our application which is based on images and annotations in that image, using TF-IDF for ranking shows slightly less relevant images. When we compare this to the $areaRatio$ ranking algorithm, we see a big difference in the relevancy of the search results. This is highlighted in the Screenshots section.

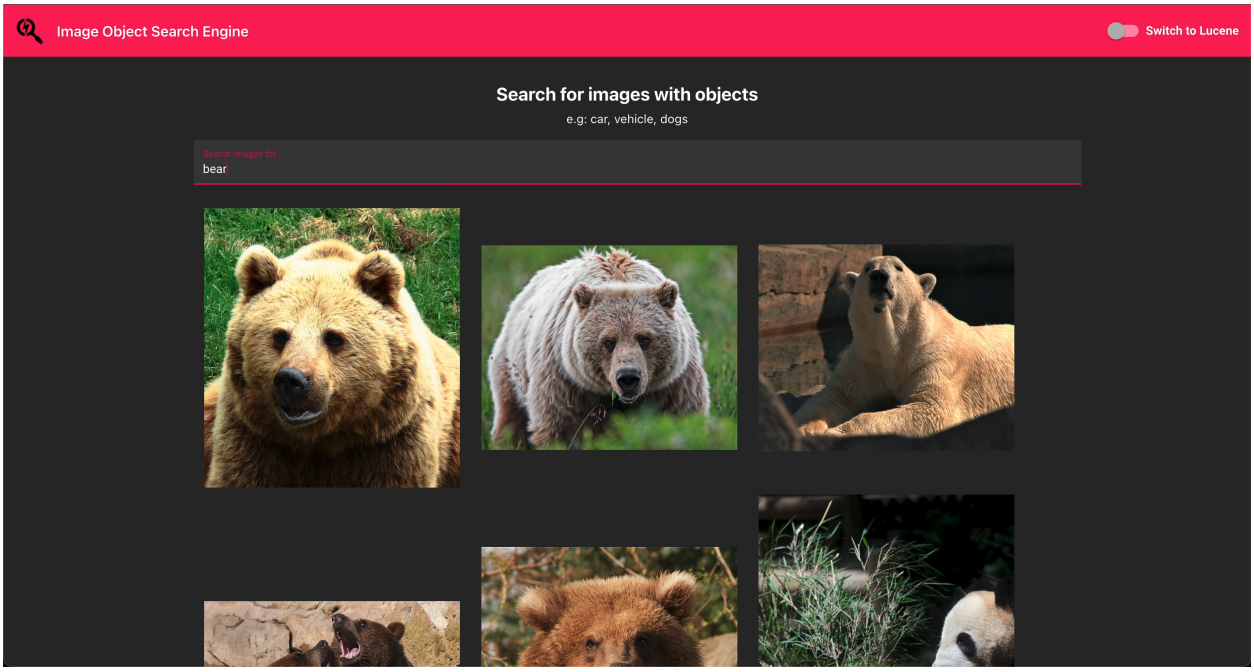
2. Slow speed of ranking Hadoop results:
   Ranking using the Hadoop Inverted Index is much slower than the Lucene Search. This is because we multiple steps and database lookups for retrieve and rank the $areaRatio$ values of the images. This can be improved more by using more indexes that allow us to speed up the time to retrieve the $areaRatio$ values.

Obstacles and Solutions:

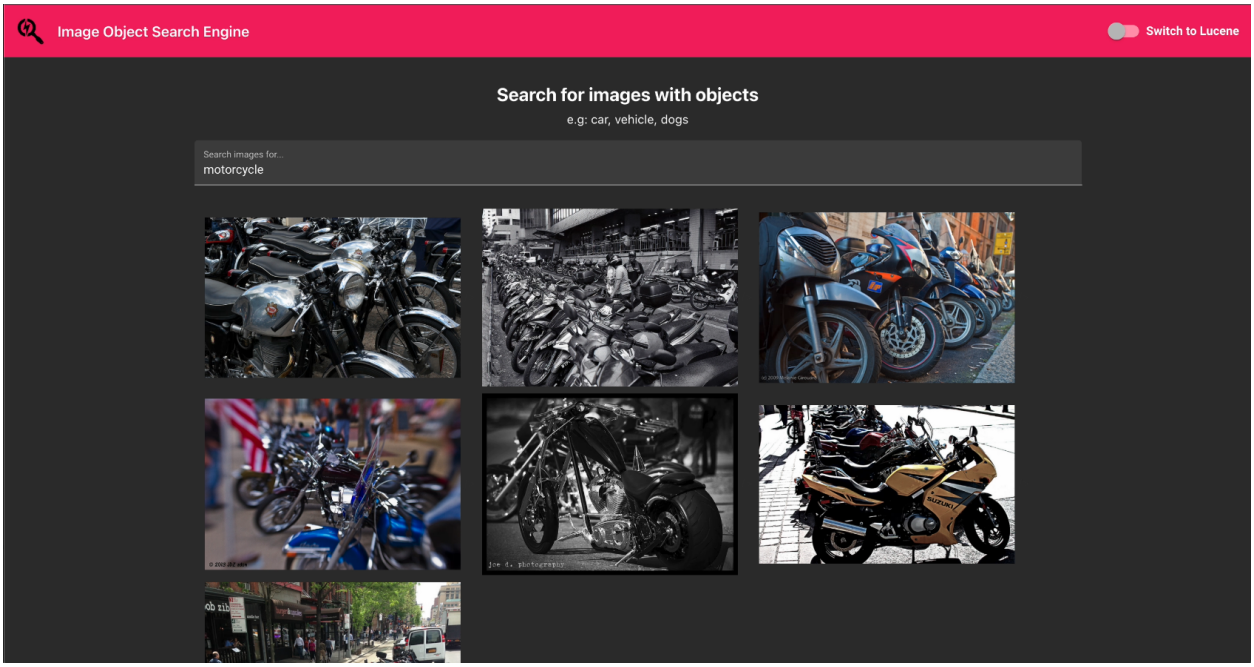
The main challenge in our Lucene implementation was the fact that we implemented and tested our Lucene index and search on the assigned WCH machine. Since we did not have permission to install any web server on that machine, we decided to migrate our setup to the local machine which contains our Hadoop and web server too.

Web Application Screenshots:

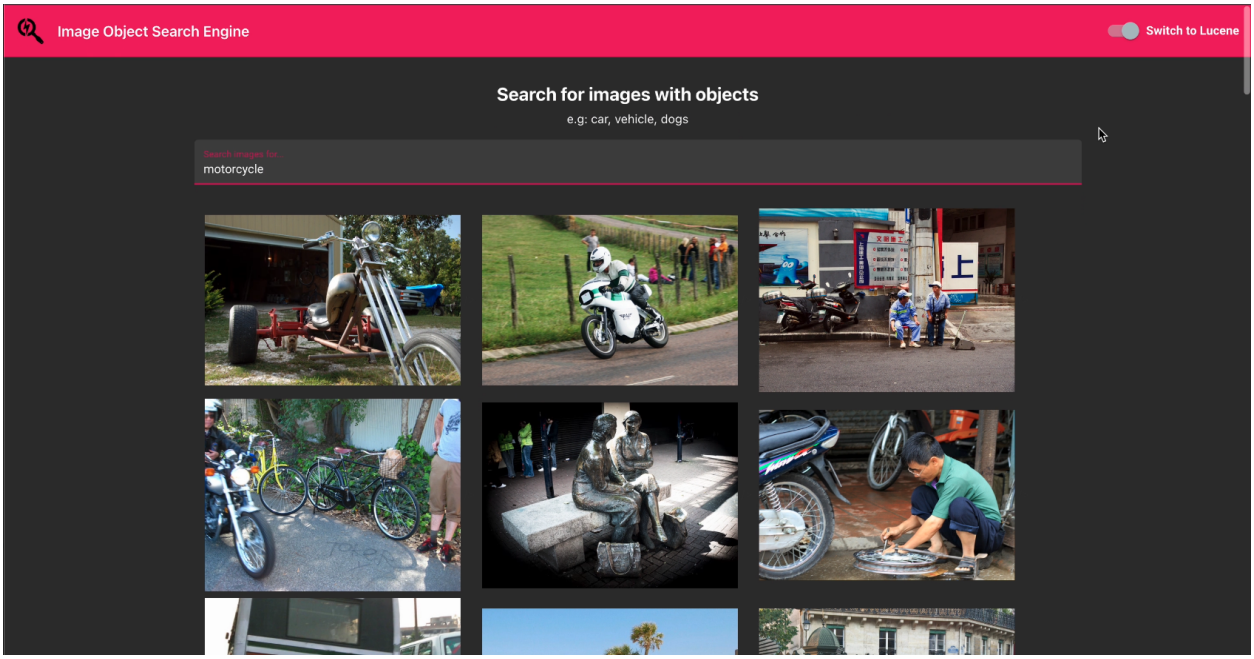
Search query: `Bear` mode: `Hadoop with Area Ranking`



Search query: `Motorcycle` mode: `Hadoop with Area Ranking`



Search query: `Motorcycle` mode: `Lucene with TF-IDF Ranking`



When we compare the results of the Lucene based search of motorcycle versus the Hadoop based search of the same word, we can see our $areaRatio$ ranking with Hadoop is much more stronger than the TF-IDF based ranking of Lucene.

Images Data:

Since the images we scraped are too large to upload, we have made a Google Drive link available to download these images. It is only accessible using RMail account. Please contact if there's any problem with accessing the link.

https://drive.google.com/file/d/1Uw54Ub12zrO_BPdQjBbk10LyZ5wW_0iA