# Image Database Architectures

Avinash Akella Silas Kati Jason Phadnis Raj Vaibhav Gude

Computer Science Dept. Computer Science Dept. Computer Science Dept. Computer Science Dept.

University of Minnesota University of Minnesota University of Minnesota University of Minnesota

Minneapolis, USA Minneapolis, USA Minneapolis, USA Minneapolis, USA

akell011@umn.edu kati0012@umn.edu phadn004@umn.edu gude0024@umn.edu

# I. Introduction

It is estimated that we generated about 2.5 quintillion data bytes daily in 2020 [5] and a good percentage of this is image data. An exponential rise in data volume has necessitated innovations in Data to facilitate efficient storage and retrieval of images. There are many reasons to use these images, such as websites or presentations. All of these images convey some information or message. However, no one has forever to look through all the options and find something that fits their needs.

With so many options, finding the right image(s) for one’s purpose could take forever without a way to quickly search them based on what’s in the image.

The design of the database we use to hold images can affect how quickly we get the images we’re looking for. This project aims to find what designs are best for what sorts of searches. If there is a “best” design to use then we want to know what it is so that we have only the highest performance. This paper then serves to help guide anyone making an image based database.

# II. Data

We used the Microsoft COCO dataset which is a massive collection of images each attached with additional identifying information. There are 12 supercategoreis of items to be found in these images: outdoor, food, indoor, appliance, sports, person, animal, vehicle, furniture, accessory, electronic, and kitchen. Then there are 80 more specific categories such as car, tennis, keyboard, and clock. In the dataset, each image is associated with whatever categories are relevant for it. If there is a car in the picture, then it has the ‘car’ category. You can search for images based on one or more categories. Even the number of instances of each category is associated with the image. Thus one can search for images with exactly 3 boats in it and get images with that criteria.

The dataset has a wide variety of images for any particular category as well. For instance, the cat category identifies images with many different kinds of cats and with many different angles. A cat can be identified even if only its head is sticking out. Images can even have categories that a human might miss in initial inspection. For example, a picture with the bird category might have a bird perched far in the background of the image’s focus.

We have about 123,267 images in our data, and the annotations for all of them combined go up to 850,999. Between the massive collection of images and the categories to identify each image by, the COCO set is perfect for our purposes.

# III. Method

We created 3 different database designs, as seen in figures A to C, to test how different queries perform in different databases. Through this we aim to determine a best query if one exists or find the conditions where one would want to use any particular design.

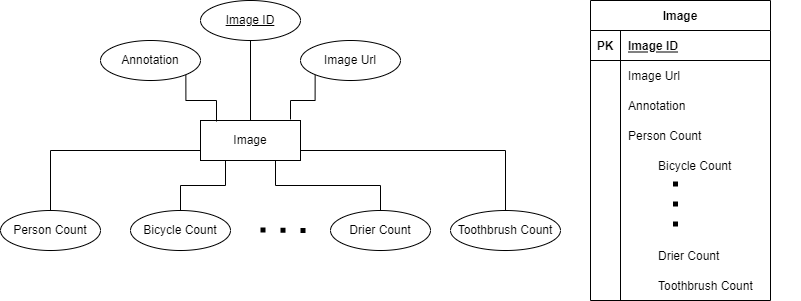


Figure A: Database Design 1 - Image

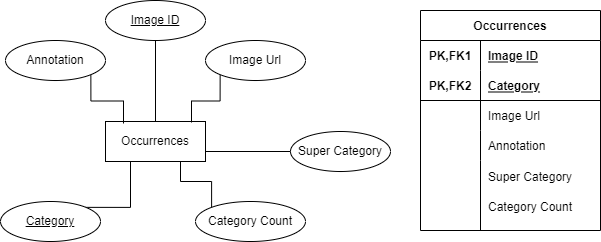


Figure B: Database Design 2 - Occurrences

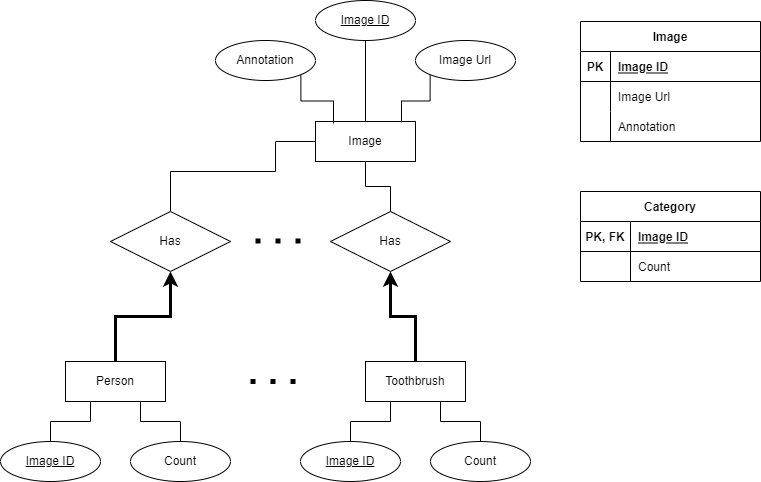


Figure C: Database Design 3 - Normalized

Figure A shows the Image design where each image has one entry in the table. There is one column for each of the 80 specific categories of images. The value in any given row for these columns is the number of those items that appear in the image represented by that row. Take for example you have an image with a single dog and 3 birds (and no other categories). This table will have one entry for this image and its values for the dog and bird columns will be 1 and 3 respectively. All other category columns will have values of 0.

In Figure B, we have the Occurrences design. It’s a single table that can have multiple rows representing any given image. Each row has a single category value and occurrence which is the number of such items in the picture. Thus using the previous example image, there will be one entry for the image where the category is dog and occurrences is 1. There will be a second row for this same image where the category’s value is bird and occurrences is 3.

Finally, Figure C is a normalized version of Occurrences which we name Normalized. Each category is one table that maintains references to any image that matches that category. Our dog and bird photo example will have two relationships. The dog category table will have an entry with a foreign key linking to the corresponding image in the image table of this design. The bird table will also have a foreign key reference to this same image. All other category tables will have no reference to this particular image.

We ran several queries on each of our three designs. For each, we also tested with no index and with 4 different indices: B-tree on category\_name and image\_id of Occurrences, bitmap on category\_name of Occurrences, separate B-tree index on image\_id and category\_name of Occurrences, and bitmap join between image\_id of the Normalized table and id of the Image table. We ran each query 100 times and measured the execution time of each. From this, we can both see the average query execution time and see how the time can fluctuate.

# IV. Results and Analysis

For all the following figures, the vertical axis is time in seconds and the horizontal is the test number (of 100). “Data” in the legends of Figure 1 below refers to database design 1, Image. Furthermore, “Normal-Occurrence” is the join on Normalized and Occurrences tables. Similarly, “Normal-Data” is the join on the Normalized and Image table.

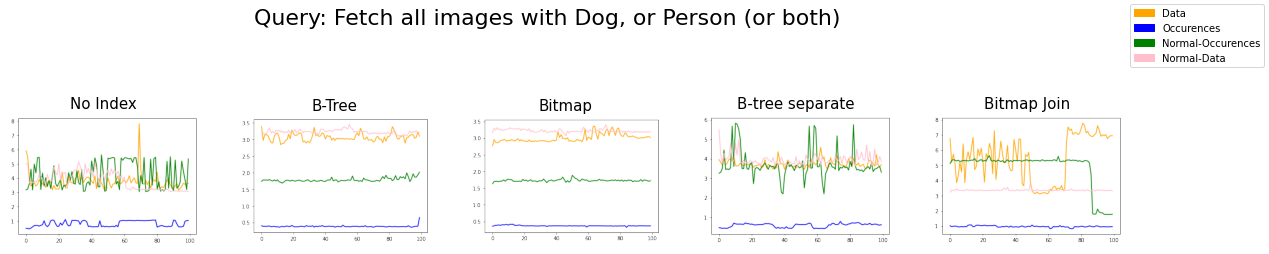
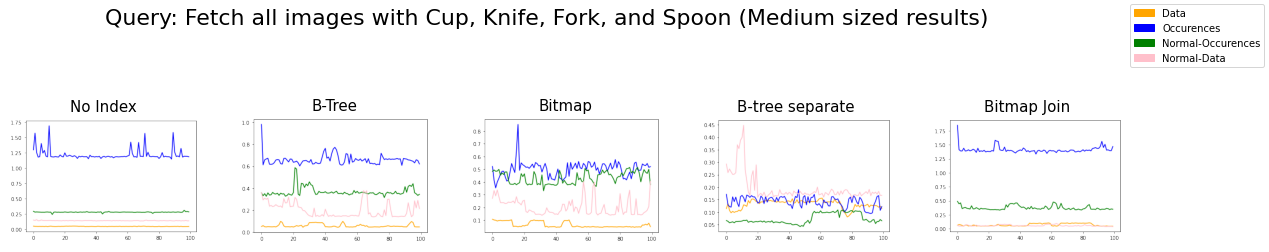
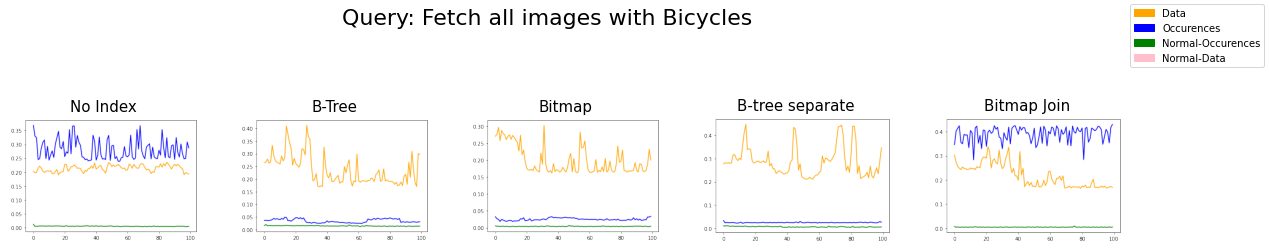
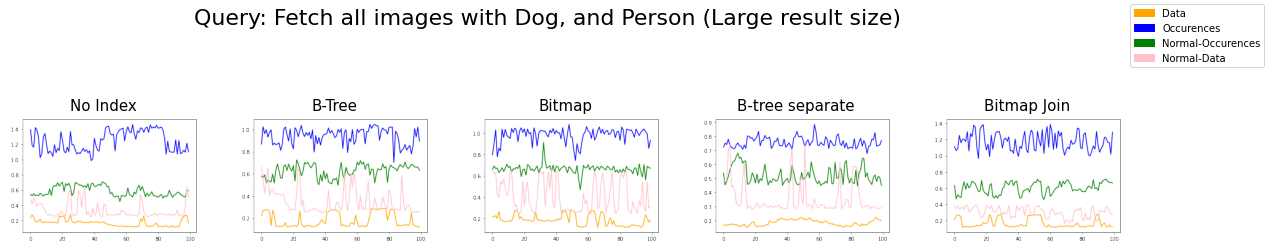
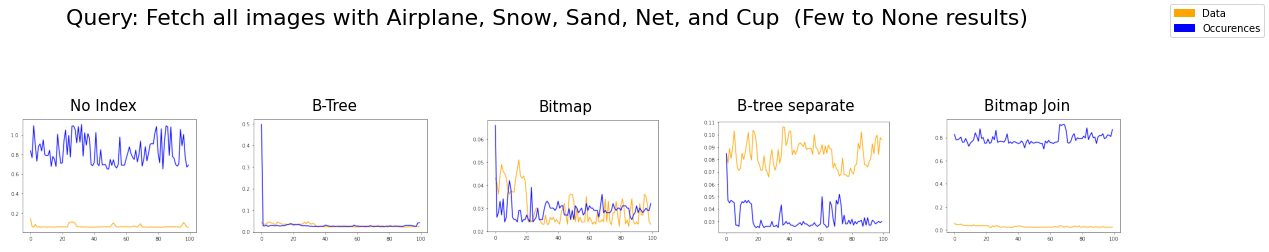


Figure 1: Five different queries have been shown here numbered from 1-5, from top to down.

Equivalent versions of five different queries have been run on the 3 database designs described earlier, along with different index options. The runtime results are shown for each query in Figure 1, where each query is numbered from 1 to 5 in top-down order.

*A. Query 1*

First, we tested getting images based on a single category.

All the indices significantly improve performance. If we're querying for a single category, then it's best to use the normalized table.

*B. Query 2*

Next, we wanted to test a query with multiple categories involved and relatively few results.

The Images table usually performs the best with the separate B-tree index being the main exception. The B-tree and Bitmap indices improve performance, and the runtimes for both designs become comparable. However, the separate B-tree index significantly boosts performance with the Occurrences table outperforming the Images table.

*C. Query 3:*

Now we see how a multi-category query with a medium-sized pool of results fairs.

The Image table performs consistently well, although other tables catch up with the help of an index. With the separate B-tree index, the query between normalized and occurrences tables (green) outperforms all of the rest. And the query on occurrences is comparable to that on data.

With the bitmap join index, the query between normalized and Image tables (pink) is as good as, or better, than the query on just the Image table.

*D. Query 4*

When the result size is large and the number of categories fetched is small, the Images table outperforms all of the rest. Although, the separate B-tree index does show an improved runtime.

*E. Query 5*

This is a surprising result, in that the occurrences table now consistently outperforms all the rest.

This possibly is because the occurrences table is smaller compared to the data, and all the other queries involve a join.

*F. Join Queries*

Finally, we have more experiments with indexes on join columns while performing table join. Fig.2.1 and Fig.2.2 have BTree indexes created on the Occurrences table. The green lines are joined with no indices, the blue is an index on image ID, and pink is an index on the category name and image ID:

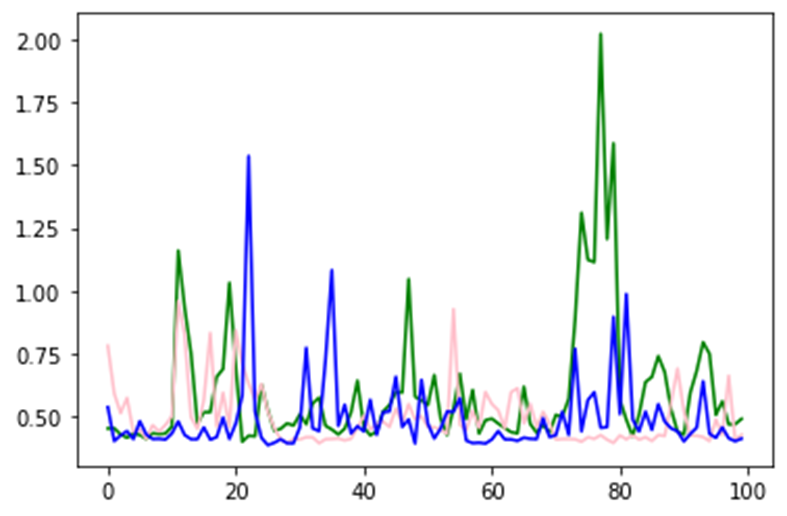


Figure 2.1: Fetching Cup, knife, fork, spoon

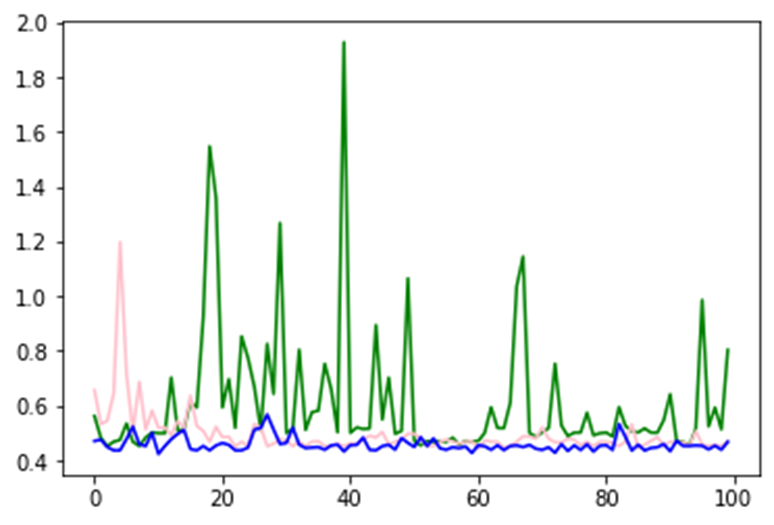


Figure 2.2: Fetching Spoon

In the above two figures, it is observed that having an index definitely helps in reducing the time taken to join. Also, the joins with an index on a single column and having a composite index on the two columns have a similar performance.

Fig. 2.3 captures the results of having indexes on both the tables participating in the join. The green line is the join with no indices, the blue is the index on the Occurrence table, and pink is the index on Occurrence and Normalized tables. The blue and pink lines almost overlap, showing not much difference in execution times.

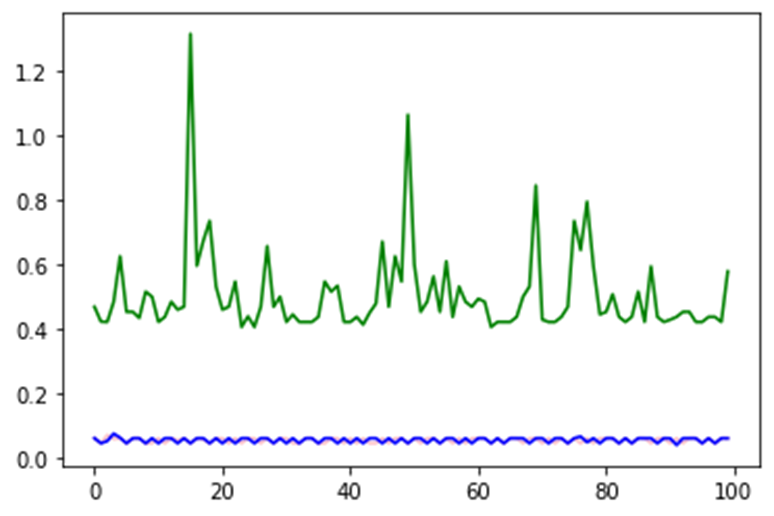


Figure 2.3: Fetching Cup, knife, fork, spoon

Next, we compare the joins with Bitmap and B-Tree indices shown in Fig.2.4. The green line is the join with no indices, the blue is the B-Tree index on the Occurrences table, and pink is the bitmap index on Occurrences. Here again, the blue and pink lines are very close to each other with Bitmap performing slightly better.

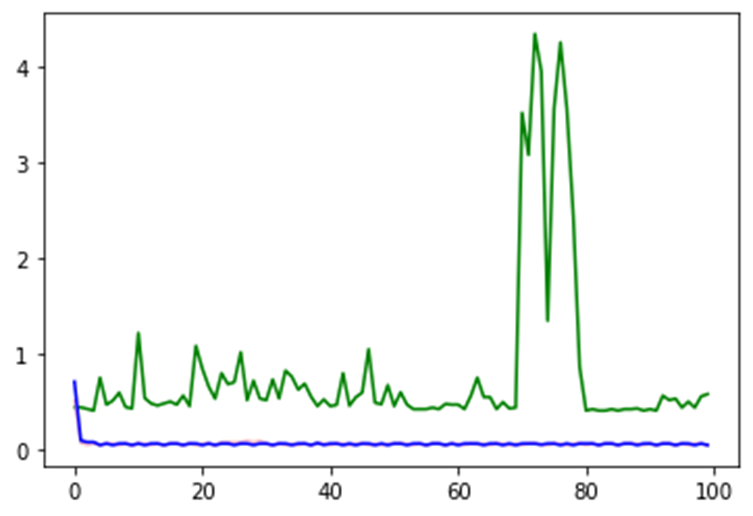


Figure 2.4: Fetching Cup, knife, fork, spoon

Finally, the comparison of AND and OR for B-Tree join and Bitmap join indices is shown in Fig.2.5 and Fig.2.6. The orange lines are joins with no indices, blue is the bitmap index on both tables, and green is a B-Tree index on both tables.

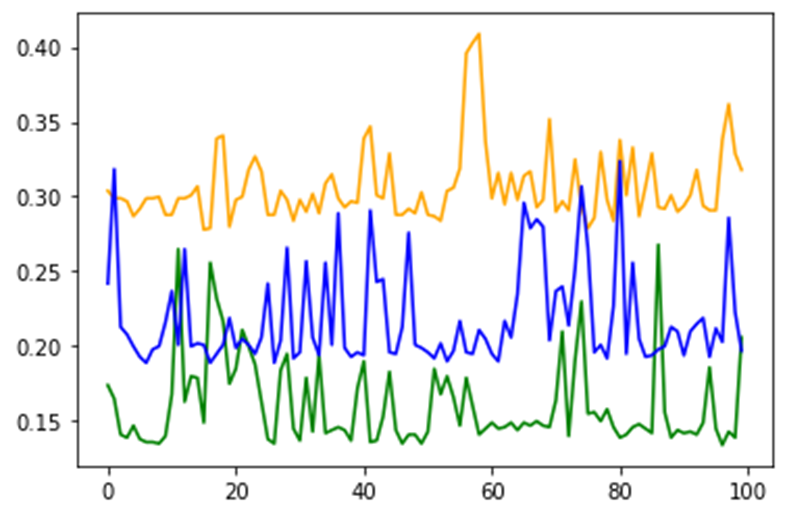


Figure 2.5: Dog AND Person

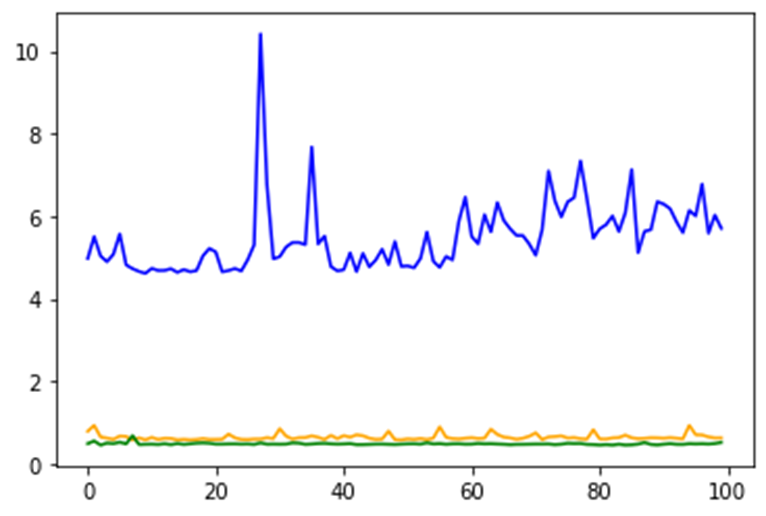


Figure 2.6: Dog OR Person

It is seen that B-Tree performs well in both cases while Bitmap join index performs considerably in the case of AND but performs very poorly in the case of OR.

# V. Conclusion

*A. Based on Query:*

The following is for a single category. When we know there's no overlap of categories, and we're interested in fetching the details of only one of them, a normalized version of the tables is the best choice.

The following is for a large number of categories. When we aim to fetch images containing a large number of categories, using normalized tables along with the occurrences table with a separate B-tree index on category\_name and image\_id fields of occurrences gives the best results.

Alternatively, a bitmap join index between a normalized and the Images table also helps. On the other hand, for small categories, the Image table outperforms all of the rest.

For the type of join: AND vs OR, the Occurrences table significantly outperforms all of the rest for OR queries. This is probably because of its compact nature and absence of any joins.

*B. Based on space constraints*

First regarding table space constraints. The Images table can take up a lot of space as it can be quite sparse. The normalized tables can have the overhead of creating and maintaining a new table for very little info, especially when the number of categories is large. The Occurrences table provides a better structure and is a good compromise.

Now for index spaces constraints. Bitmap and Bitmap-join indexes are useful for low-cardinality columns. For columns with high cardinality, they have a lot of sparsity and waste space. In theory, B-trees should perform worse for columns with cardinality less than 200, but in practice, they work just fine in our case.

*C. Based on the type of data*

For multi-domain data, the normalized version is especially useful in such cases as each category can have different metadata. Both the Images and Occurrences tables can be hit by sparsity here.

# VI. Future Work

We could look for more scalable architectures. In a real-world setting, the images would be stored in File Storage Systems distributed across servers. So setting up and comparing the performance of these queries in a distributed setting could be helpful. Also, we could implement and improve an image segmentation or object detection algorithm to get better metadata. Gathering more metadata can also improve the quality of the search.

Also, we could explore methods for performing similarity searches using an image. This is a case where a user would input an image as a query. To tackle problems like this we could generate the hash value using an optimal function and store that information in the Database. Furthermore, an index over this column could speed up subsequent searches for comparison with the input hash value.

# Acknowledgments

For this project, we used Python as the programming language as it has some libraries that make it easier to work with databases. For instance, pandas and NumPy helped with the data cleaning. Sqlalchemy meanwhile helped us translate our data from CSV files to tables. We chose to use Oracle DB as our group has some prior experience with it.

Regarding the division of work, we all discussed the idea and where we wanted to go for this project. Raj wrote up a Python script to help with cleaning and formatting the data to be put into the database. Silas set up the Database and environment for the experiments and analysis. Avinash focused on writing the code to set up the tables and performed experiments. Raj performed additional experiments for the join queries. Jason drew up SQL diagrams based on the designs that the group discussed and created the presentation outline. We also wanted to try the same experimentation on AWS but decided it would end up becoming expensive for a group project. All members worked on the presentation and we transferred much of the information there to this report. All of us thus had a hand in making this report and making sure it was of satisfactory quality.

# VII. References

1: Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, Piotr Dollár. “Microsoft COCO: Common Objects in Context,”arXiv:1405.0312 [cs.CV], May 2014

2: Raghu Ramakrishnan and Johannes Gehrke. 2002. Database Management Systems (3rd. ed.). McGraw-Hill, Inc., USA.

3: Brown MS, Shah SK, Pais RC, Lee YZ, McNitt-Gray MF, Goldin JG, Cardenas AF, Aberle DR. Database design and implementation for quantitative image analysis research. IEEE Trans Inf Technol Biomed. 2005 Mar;9(1):99-108. DOI: 10.1109/titb.2004.837854. PMID: 15787012.

4: Güntzer, Ulrich & Balke, Wolf-Tilo & Kieling, Werner. (2000). Optimizing Multi-Feature Queries for Image Databases. Proc. of the 26th VLDB Conference.

5. Bulao, Jacquelyn. “How Much Data Is Created Every Day in 2021? [You'll Be Shocked!].” *TechJury*, 7-Dec.-2021, https://techjury.net/blog/how-much-data-is-created-every-day/#gref.

6. “Common Objects in Context.” *COCO*, Microsoft, https://cocodataset.org/#overview.