Virtual Gallery (Powered by NGA)

By DataOmni – L.X., X.L., Y.Z., Xianjie (Iris) Ma

Introduction

Our virtual art gallery web-application aims to provide users an enriched browsing experience with plenty of amazing 2D-artworks exhibited/hosted by National Gallery of Art (NGA), meanwhile, opens a new dimension to the traditional gallery with several data analysis features. This virtual gallery browsing experience helps people enjoy these artworks at home instead of traveling to visit them in person. This web-application serves an educational and research purpose, as it provides an introduction to some of the finest artworks by a group of prominent artists around the world. For instance, our searching functionality allows users to browse artworks by artists, title of artworks, style, nationality, etc. We also performed analysis of these artworks based on their keywords/styles/historical elements/geographical elements, etc. These analytical results provide more insights about the rationales and motivations behind the creation of these artworks. Additionally, our website could serve as a source of aesthetic inspiration for art creators. We achieved this by tailoring the aesthetic designs of our webpages with the visual/color components of the artworks being displayed. There are also several interesting functionalities such as "to give the user an idea about the actual dimensions of artworks by asking the height of the user, and display some artworks with regard to this height". Lastly, we hope our virtual site experience could encourage people to make physical tours to NGA, which will definitely complement them with a more holistic perceptual experience of these artworks.

System Architecture

We designed our full stack system as a three-tier-architecture: 1) database, 2) backend web-server, and 3) frontend client. For instance, a completed cycle of application runtime is like the following: upon launching the application, the frontend unit takes in the user's input to send it as a request to the backend web-server. The backend web-server serves as a relay station between the front-end and database, in which it establishes the connection with the database-server, and converts front-end user's inputs into database queries, which are sent to the database server to fetch query results. The query results are returned to the web-server and get placed on a specific port as the response to the front-end's request. The frontend knows this port and comes to fetch the response to format and display them back to the user.

Data Source and Technologies Used

Our data sources were completely provided by the NGA official Open Data project (see Appendix B for detail) . Exploratory data analysis (EDA) was performed on the downloaded raw datasets to check data availability (presence or absence of null values), usefulness of data, units, and in general to summarize the data characteristics. Subsequently, only relevant data columns, identified to be useful to the project goal, were extracted from the full datasets. Python was used at this stage due to its easy-to-use libraries, including pandas and numpy, which are widely used in data science applications. The processed datasets were hosted on AWS RDS using MySQL.

As for the front-end side, the web application was developed using React, a user interface framework developed by Facebook. Using the Node package manager (NPM), a boilerplate application was set-up via the Node package create-react-app. Specifically, we used "React components Class" to capture user's actions (front -> back) & fetch query results (back -> front) to update to user; we used "React Hook" to hook data for world map display in HomePage and for static collections of artworks in AnalysisPage; we used "Ant design Charts" for the World Map in HomePage and "Ant design Plots" for the Word Cloud in AnalysisPage. In addition, we utilized "React-animated-slider" in our Home-Page welcoming-section, "react-color-extractor" to display dynamically generated color palette in artwork detailed display page, and "React-responsive-masonry" in Search-Page result-presenting-section.

On the backend we used Express.js and Node.js to write handlers for requests and to connect the web application with the database, so as to return results from user queries. The Node.js "util" module was used to enable promisify() function (synchronicity) and realize sequential execution of multiple query requests.

Web App Description

Home Page: Our home page displays an overview for our artwork collection. Summary statistics of our artwork collection and links to other pages are embedded in the image slider, background of which is from some of the most famous artworks of NGA. A responsive world map reflects the most updated data instance of our database that shows the geographical origins of all artworks in our collection. The size of each origin group is visualized by the size of the bubbles on the map.

Search Page: Our search page gives users a variety of options to search for artworks and to view artworks in a waterfall style.

- Users can search by the suggested filters: classification, style, nationality of the artist, and/or time period.
- Users can type in the artist's name and/or artwork's title for search.
- To add more fun - users can type in their height or birth year to view artworks with the same height/created in the same year!

All results are populated from our database and updated based on the search method. 6 results are shown initially, while "load more" and pagination allow users to view as many results as they like.

Artwork Page: Our artwork page includes all detailed info related to an artwork: image, title, artist, medium, dimensions, classification, keywords, etc. All data is dynamically fetched from our current database instance. A section of color palette is automatically generated based on each artwork for aesthetic inspirations.

At the bottom of the object page, 4 similar artworks are recommended to users. The algorithm utilizes the facts of shared artists/exhibition/style/theme/etc. Users can continue exploring similar artworks by a simple click.

Analysis Page: Our analysis page presents the results of 4 different analyses over the collection, empowered by SQL queries.

- Collection keywords (styles, locations, themes, etc.) are shown in a word cloud. Users can see the frequency of each keyword by hovering their mouse over the cloud.
- Top 8 styles in collection are provided for users who want to view artworks for style (eg. Impressionist/Renaissance/Realist/etc.). A single click on the album will bring users to the respective collection.
- Portraits across time loads 6 portraits initially, each from a century between 1400 - 2000. A shuffle button on the right hand side allows users to shuffle and load new portraits from each century. The purpose is for users to get a glimpse of the historical evolution of portrait painting styles over time.
- Collection distribution by type is visualized using a bar chart. Users can select a term type from the menu (eg. style/school/theme/technique/keyword/place), and the chart will return the most common terms within that type.

Performance Evaluation

Performance had been a themed concern throughout the design and implementation of our database and web-application. As such, performance improvements were done at several levels of our system.

3NF Relational Schema Design: Firstly, we were really lucky to find a well-structured dataset to begin with. Upon initial data exploration, we immediately realized that the experts in NGA designed and constructed their datasets directly into a well-normalized 3NF fashion (i.e. for each table, every non-prime attribute is non-transitively dependent on the primary key). So we just strictly followed their design, and extracted the subset of data that served our purpose without breaking the 3NF integrity.

Data Cleaning: Since our web-application is focusing on the display of 2D artworks, we extracted ~80,000 artworks (that were under the classification of "painting", "drawing" and "prints") out of the ~130,000 artworks hosted by NGA. These extracted artworks are then used to extract their associated relationships, entities and attributes. This process helped to achieve a significant size reduction in collective csv file size (125.5MB to 48.7MB, see Appendix B, Table B.1 for detail), which were the source files of our database.

Indexing: Upon initial population of our database, the MySQL InnoDB engine automatically created B-Tree indexes on all the primary keys for all the relations (we also learned that InnoDB only supports B-Tree indexing). Subsequently we analyzed, for each relation, which non-PK attributes got frequently utilized as query predicates (in WHERE clause), so that we could add indexes on these attributes to speed up our query execution time. Please refer to Appendix E, Table E.1, for all the indexes that we manually created for each relation. There were some key rationales for deciding about which attribute(s) to index on for each table. In our functional queries, we frequently needed to lookup artworks within a time span (which were defined by a "beginYear" and an "endYear" attributes for each tuple), and we also needed to sort the artworks by its completion time (which were defined by their "endYear"). Hence we assigned two indexes for "endYear", one by itself, one in a combination with "beginYear". At the same time, in order to ensure a better user experience with the artwork searching functionalities, we indexed on all the relevant attributes that were used for the searching filters: classification, nationality, timespan and style. This really speeded up our searching time by 5 times faster (Appendix E, Table E.2, "filterSearch()" route). As for the "keywordSeach()" function, we did encounter a challenging situation when attempting to create indexes on three of the text-heavy attributes in the "object"-relation, namely the "title", "attribution" and "attributionInverted" attributes. The issue was caused by the declared widths of these 3 attributes exceeding the max 3072 bytes index width restriction for MySQL InnoDB engine. Due to the unpredictable nature of the length of an artwork's title and its attributions (as well as the consideration of any future new data insertions), we did not want to reduce the width of these 3 attributes. Instead, we gave up on adding indexes on these 3 attributes, and tested the query performance with all the other allowable indexes. It turned out that our query performance was still improved to be 1 time faster, which was practically acceptable. Lastly, for all the analysis related queries, the addition of index on "termType" attributes drastically boosted up the query speed, with 7 times faster query performance, we were able to bring down the loading speed of our analysis page from 4.25s to 1.6s, which transformed a laggy user experience to a much smoother feeling. As shown in Appendix E, most query execution timing were below 1s, two queries were above 1s, with the longest being 1.5s.

Caching: As for caching, we just relied on the built-in caching mechanism of MySQL InnoDB. When query data directly from a MySQL console (i.e. Datagrip), we were able to feel the improvement of query returning speed upon repeated execution of the same "cacheable" queries. This did help increase the loading speed of some functions containing static queries, but overall, due to the loading bottleneck coming from front end REACT.DOM components (since our website was visually enriched with digital artwork images and dynamic aesthetic design), the help from query caching was negligible.

Technical Challenges

Database Technical Challenge: As mentioned previously, we were pretty lucky for the data sources exploration and data cleaning process. So the first challenge we faced was at the stage of database schema creation and data population. Due to the nature of artistic work, our collection contained artworks that originated from different languages, which made our data content containing a variety of alphabets or character sets. So we first investigated our data source clarification about the potential range of language types (i.e. character sets) that our artworks include. Luckily, the NGA official data source provided clear instructions for this, which specified that these data are defined as UTF-8 (4 bytes for each character) coding. Through further familiarization with the MySQL database system design, we discovered that MySQL default table schema was with InnoDB engine having latin1 (1 byte for each character) coding. So we first need to adjust our table schema declaration to be UTF-8 coding. After the creation of relation schemas, we started to upload our data sources csv files into the database. This was when we discovered that having foreign key(FK) constraint(s) in a relation really slowed

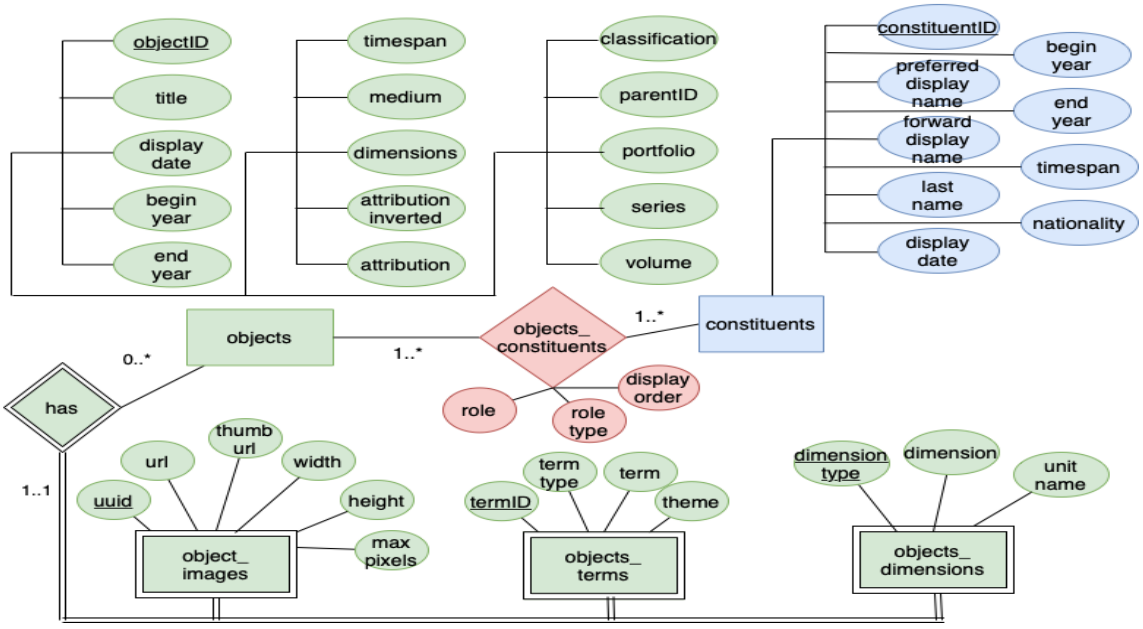
down the massive data-population process. Since for every tuple being uploaded, the database needed to perform a check of the FK's existence in the table being referenced (which we have already verified in our data cleaning process). Instead, we removed the FK constraint declaration, populated the database with cleaned up source data, and then re-added back the FK constraints when all the data was loaded on.

Back-End Technical Challenge: When we were building the logic for our back end functionality, we realized that due to the language feature of javascript and nature of website application, our route handlers were all "async" functions. However, for some of our designed website features, we needed to fetch more than one queries sequentially, because the later queries were dynamically dependent on the result returned from the previous queries. For instance, our recommendation of similar artworks feature (i.e. similarArtworks() route) needed to look for similarities based on many optional criteria. As the back-end, we only asked the front-end to return back the ID of an artwork, and we will use this ID to look for its associated attributes, and then search for artworks that share the same values in these attributes. For this, we learned to use the ".promisify()" function in the Node.js' util module, which enabled our "connection" instance to the MySQL server to execute queries sequentially as needed. So that we could allow the execution of multiple inter-dependent queries in an one-after-another fashion. Another challenge was from the implementation of searching by filter functionality (i.e. filterSearch()). Due to the 3NF design which minimizes redundancy of data, we had to perform joining of several tables to bring back all the associated attributes and relationships regarding an artwork object. As we learned throughout the ***** course that big joinings could slow down the query performance significantly. Hence, we wrote conditional checks logics to divide the filter status into different modes. We analyzed that the "nationality" filter required the joining with 2 additional tables, and the "style" filter required 1 additional joining. So we case splitted the corresponding query scripts into 4 types (please refer to our "routes.js" code file for detail). This logical optimization allowed our database to avoid unnecessary joinings whenever possible.

Front-End Technical Challenges: As to achieve a pleasant user experience, we initially had some challenges with displaying multiple artworks as the search results (on the search page). As we know the aspect ratio for artworks could differ greatly. It was not aesthetically pleasing if we fixed the width/height/both when presenting them all together. We overcame this issue by utilizing a waterfall-style React component (i.e. React-responsive-masonry). However a subsequent issue immediately emerged: it did not come with built-in pagination functionality. Hence, we customized a "load more" function on our own. This function was able to load additional search results under different scenarios. Search results are displayed under 3 scenarios - search by keywords, search by filter, and search by collection (i.e. when re-routed from the "Top 8 styles in collection" of the analysis page). This requires the "load more" button to only load the additional artworks under a certain scenario. For instance, when users perform the search based on the keyword "flamingo", the search results consist of flamingo-related artworks, and "load more" should only add flamingo-related artworks. If users are searching for French artworks using filters, load more should only add French artworks. We successfully implemented the "load more" function by adding a "mode" field in the class. When rendering, the mode field will be checked first, and then different results will be rendered according to the current mode.

Lastly, on the homepage, we wanted to use a marked world map to showcase the summary statistics of the countries of origins of our artworks. And we wanted this summary data to be dynamically reflecting the most updated database instance. After some searching on implementation methods, we identified that the utilization of React-Hook framework along with a world map chart would make this happen. We used "useState/useEffect" functions to fetch the summary data, then converted it to match geo.json location. Initially when rendering the graph, we were confused about why hook would return the maps twice: first was the plain world map outline, and second was the marked. After researching about Hook's useState() calls, we realized that the issue was about the initial default state and updated state, and we wanted to display the updated state only (which was the marked map). We then customized a useEffect() conditional checking block that only rendered the graph after the initial call, so that artwork countries data were mounted onto the world map, which was displayed to the user.

Appendix A.1： Entity Relationship Diagram



Appendix A.2-Relation Schema:

objects (objectID, title, displayDate, beginYear, endYear, timeSpan, medium, dimensions, attributionInverted, attribution, classification, parentID, portfolio, series, volume)
constituents (constituentID, preferredDisplayName, forwardDisplayName, lastName, displayDate, beginYear, endYear, visualBrowserTimeSpan, visualBrowserNationality)
objects_constituents (objectID, constituentID, displayOrder, roleType, role)
objectID FOREIGN KEY REFERENCES objects(objectID)
constituentID FOREIGN KEY REFERENCES constituent(constituentID)
objects_images (uuid, objectID, URL, thumbURL, width, height, maxpixels)
objectID FOREIGN KEY REFERENCES objects(objectID)
objects_dimensions (objectID, dimensionType, dimension, unitName)
objectID FOREIGN KEY REFERENCES objects(objectID)
objects_terms (termID, objectID, termType, term, visualBrowserTheme)
objectID FOREIGN KEY REFERENCES objects(objectID)

Appendix B： Data Sources

Data Source: National Gallery of Art (NGA) Open Data

- https://www.nga.gov/open-access-images/open-data.html
- https://github.com/NationalGalleryOfArt/opendata

These data were already placed in well-formatted CSV tables. We just needed to perform some exploratory data analysis and extract the subset of columns and rows that are relevant to our website's focus.

CSV Table Name	**Description**	**Original Size** (cols x rows)	**Extracted Size** (cols x rows)
objects	Artwork objects are the core entity of our dataset. Objects are physical or logical constructs of art..	136929 x 29 (52.5MB)	81843 x 15 (17.4 MB)
constituents	A constituent is a single person or a group of persons. A constituent may be an artist, an artist	18282 x 13 (2.3MB)	9941 x 9 (1.1 MB)

	after, or any means of being the creator of the artwork. A constituent can be the creator of one or more objects.		
objects_constituents	This table maps the object-constituent relationships to describe the association between an object and its constituent(s).	467716 x 12 (9.5 MB)	103004 x 5 (2.9 MB)
objects_images	Digital images of artwork objects and their detailed informations that have been released by NGA to for public access under a Creative Commons 0 (CC0) license	103047 x 12 (28.7 MB)	81912 x 7 (17.2 MB)
objects_dimensions	records various dimensions for artwork objects	206598 x 5 (14.7 MB)	113682 x 4 (3.5 MB)
objects_terms	Record the descriptive or analytical terms of artwork objects.	386242 x 6 (17.7 MB)	187066 x 5 (6.6 MB)

Table B.1: Summary Statistics of the 6 CSV Tables that are used to populate our database

Appendix C： API Specification

Route#1 – galleryOverview(req, res)
Description: query and return for gallery's summary statistics
Request Path: `GET /home` **Route Parameter(s)**: n/a **Query Parameter(s)**: n/a
Return Type: JSON **Return Parameters**: `{ msg: (string of welcoming message), results: [ {classification: "painting", artworkCounts: (int) }, {classification: "drawing", artworkCounts: (int) }, {classification: "print", artworkCounts: (int) } ], ArtworkOrigin: {regionName (string): artworkCounts (int), . . . (cardinality: 0..*) . . . } }`

Route#2 – artworkInfo(req, res)
Description: given the objectID of an artwork, this function will query and return all the necessary/detailed information (results are broken down into 3 parts) about a given artwork
Request Path: `GET /artwork` **Route Parameter(s)**: n/a **Query Parameter(s)**: `objectID` (int)
Return Type: JSON **Return Parameters**: `{ results_P1: [{ title: (string), attribution: (string), medium: (string, nullable), dimensions: (string, nullable), classification: (string), series: (string, nullable), portfolio: (string, nullable), volume: (string, nullable), URL: (string) }], results_P2:[{ preferredDisplayName: (string), displayOrder: (int), displayDate: (string), visualBrowserNationality: (string)}, {element2}, . . . (artist cardinality: 1 .. *) . .], results_P3:[{ termType: (string), term: (string) }, . . .(cardinality: 0 .. 6) . .] }`

Route#3 – similarArtworks(req, res)
Description: recommend similar artwork by primary (i.e. results_P1) and secondary (i.e. results_P2) similarities
Request path: `GET /artwork/similarArtworks`
Route Parameter(s): n/a **Query Parameter(s)**: `objectID` (int)
Return Type: JSON **Return Parameters**: `{ results_P1: [{ title: (string), attribution: (string), objectID: (int), thumbURL: (string), series: (string, nullable), portfolio: (string, nullable), volume: (string, nullable) },. . .(cardinality: 0..4) . .], results_P2:[{ title: (string), attribution: (string), objectID: (int), thumbURL: (string), termType: (string), series: (string, nullable), portfolio: (string, nullable), volume: (string, nullable) } . . . (cardinality:: 0 .. 4) . .] }`

Route#4 – filterSearch(req, res)
Description: search relavent artworks by applying a variety of filtering conditions. Result will be returned by the following ordering: endYear >> title >> attribution.
Request Path: `GET /search/byFilter`
Route Parameter(s): n/a **Query Parameter(s)**: `nationality` (string), `style` (string), `classification` (string), `beginYear` (int), `endYear`(int), `page` (int, default: 1), `pagesize` (int, default: 6)
Return Type: JSON **Return Parameters**: `{ results: [ { title: (string), attribution: (string), endYear: (int), objectID: (int), thumbURL: (string) }, . . . (cardinality: 0..*) . . ] }`

Route#5 – keywordSearch(req, res)
Description: search relavent artworks by artwork's title OR/AND artist's name (**in case-insensitive manner**)
Request Path: `GET /search/byKeyword`

Route Parameter(s): n/a **Query Parameter(s)**: `artworkTitle` (string), `artistName` (string), `page` (int, default: 1), `pagesize` (int, default: 6)
Return Type: JSON **Return Parameters**: `{ results: [ { title: (string), attribution: (string), endYear: (int), objectID: (int), thumbURL: (string) } . . . (cardinality: 0..*). . ] }`

Route#6 – naughtySearchHeight(req, res)
Description: naughty search "painting" artworks by matching user's height (cm) with artwork's height (cm), return a list of artworks in the order of least height-deviation to most height-deviation
Request Path: `GET /search/naughtySearchByHeight`
Route Parameter(s): n/a **Query Parameter(s)**: `height` (int or float, default: 170), `page` (int, default: 1), `pagesize` (int, default: 10)
Return Type: JSON **Return Parameters**: `{ results: [{ title: (string), attribution: (string), objectID: (int), thumbURL: (string), dimension: (float), deviation: (float) }, . . .(cardinality: 0..10) . . ] }`

Route#7 – naughtySeachBirthYear(req, res)
Description: naughty search artworks by matching with user's birthYear, return the artwork (of all kinds) produced in/around the birthYear, and then order the results in height descending order (tall --> short)
Request Path: `GET /search/naughtySearchByBirthYear`
Route Parameter(s): n/a **Query Parameter(s)**: `birthYear` (int), `page` (int, default: 1), `pagesize` (int, default: 10)
Return Type: JSON **Return Parameters**: `{ results: [{ title: (string), attribution: (string), objectID: (int), endYear: (int), deviation: (int), thumbURL: (string), deviation: (float) }. . . (cardinality: 0..10). . . ] }`

Route#8 – analysisOverview(req, res)
Description: query and return the top 80 popular terms (and the counts of associated artworks) for each of the 5 big analysis categories (Style, School, Technique, Theme, Keyword, Place Executed)
Request Path: `GET /analysis/analysisOverview` **Route Parameter(s)**: n/a **Query Parameter(s)**: n/a
Return Type: JSON **Return Parameters**: `{Style:[{term: (string), StyleCounts: (int)}, . . (cardinality: 80) . .], School:[{term: (string), SchoolCounts: (int)}, . . . (cardinality: 80) . .], Theme:[{term: (string), ThemeCounts: (int)}, . . .(cardinality: 80) . .], Technique:[{term: (string), TechniqueCounts: (int)}, . . (cardinality: 80) . . ., {element5}], Keyword:[{term: (string), KeywordCounts: (int)}, . . .(cardinality: 80) . ., {element5}],PlaceExecuted:[{term: (string), PlaceExecutedCounts: (int)}, . . (cardinality: 80) . .]}`

Route #9 – analysisByType(req, res)
Description: Front-end will prompt user to specify which type of analysis he/she wants to check. This function will return, in descending order, most popular terms under the analysis category. (analysis category: Style, School, Theme, Technique, Keyword, Place Executed)
Request Path: `GET /analysis/analysisByType/:analysisType`
Route Parameter(s): `analysisType` (string)
Query Parameter(s): `page` (int), `pagesize` (int, default: 10)
Return Type: JSON **Return Parameters**: `{results : [{term:(string), termCounts:(int)}. . .( cardinality: 1 .. *) . . ] }`

Route #10 – portraitsAcrossTime (req, res)
Description: Within the given time range, find and return artworks that have their theme of contents been defined as portraits. Front-end will fetch for 5 different time-spans:15th (before 1500), 16th (1500~1599), 17th (1600~1699), 18th(1700~1799), 19th (1800~1899), 20th (1900~1999) centuries
Request Path: `GET /analysis/portraitsAcrossTime/:artworkClass`
Route Parameter(s): `artworkClass` (string) **Query Parameter(s)**: `beginYear` (int, default: 1500), `endYear`(int, default: 1599), `page` (int), `pagesize` (int, default: 5)
Return Type: JSON **Return Parameters**: `{ results: [ { title: (string), attribution: (string), classification: (string), objectID: (string), thumbURL: (string a URL), endYear (int) }, ... (artwork cardinality: 1 .. *) ... ] }`

Appendix D： MySQL Queries

Route#1 – galleryOverview(req, res)

1.1) counts number of artworks under each classification

```
SELECT classification, count(*) as artworkCounts FROM objects GROUP BY classification;
```

1.2) get all the known geographical origin of artwork (i.e. by artist's nationality)

```
SELECT visualBrowserNationality AS nationality, COUNT(DISTINCT O.objectID) AS artworkCounts
FROM objects O JOIN objects_constituents OC JOIN constituents C
        ON O.objectID = OC.objectID AND OC.constituentID = C.constituentID
GROUP BY C.visualBrowserNationality ORDER BY COUNT(*) DESC ;
```

Route#2 – artworkInfo(req, res)

2.1) fetch the basic information about the artwork itself

```
SELECT O.title, O.attribution, O.medium, O.dimensions, O.classification, O.series, O.portfolio, O.volume, OI.URL
FROM objects O JOIN objects_images OI ON O.objectID = OI.objectID
WHERE O.objectID = ${objectID};
```

2.2) fetch the information about the artist(s) of the artwork

```
SELECT C.preferredDisplayName,OC.displayOrder, C.displayDate, C.visualBrowserNationality
FROM objects_constituents OC JOIN constituents C ON OC.constituentID = C.constituentID WHERE OC.objectID = ${objectID}
ORDER BY displayOrder;
```

2.3) fetch information about the analytical content of the artwork

```
SELECT OT.termType, OT.term  FROM objects_terms OT WHERE OT.objectID = ${objectID} ORDER BY termType;
```

Route#3 – similarArtworks(req, res) {MOST COMPLEX FUNCTION}

3.1) Primary Recommendation: find similar artworks done by the same artist, or in the same series/volume/portfolio

```
SELECT DISTINCT O.title, O.attribution, O.objectID, OI.thumbURL, O.series, O.portfolio, O.volume
FROM objects O JOIN objects_constituents OC  JOIN constituents C  JOIN objects_images OI
ON O.objectID = OC.objectID AND OC.constituentID = C.constituentID AND O.objectID = OI.objectID
WHERE (O.objectID <> ${objectID} ) AND (O.classification = '${artworkClass}') AND (
        (O.portfolio LIKE '${portfolio}') OR (O.series LIKE '${series}') OR
        (O.volume LIKE '${volume}') OR  (C.constituentID = ${artistID}) )
ORDER BY O.series, O.portfolio, O.volume, O.attribution LIMIT 4;
```

3.2) Secondary Recommendation: find similar artworks based on their analytical contents

```
SELECT DISTINCT O.title, O.attribution, O.objectID, OI.thumbURL, OT.termType, O.series, O.portfolio, O.volume
FROM objects O JOIN objects_images OI JOIN objects_terms OT
ON O.objectID = OI.objectID AND O.objectID = OT.objectID
WHERE (O.objectID <> ${objectID}) AND (O.classification = '${artworkClass}') AND (
          (OT.termType = 'Style' AND OT.term = '${style}') OR (OT.termType = 'School' AND OT.term = '${school}') OR
          (OT.termType = 'Keyword' AND OT.term = '${keyword}') OR (OT.termType = 'Theme' AND OT.term = '${theme}') )
ORDER BY termType LIMIT 4;
```

Route#4 – filterSearch(req, res) {MOST COMPLEX FUNCTION}

4) search relevant artworks by a variety of filters:

4.1) CASE 1: having "style" filer AND "nationality" filter

```
SELECT DISTINCT O.title, O.attribution, O.endYear, O.objectID, OI.thumbURL
FROM objects O JOIN objects_constituents OC JOIN constituents C JOIN objects_images OI JOIN objects_terms OT
           ON O.objectID = OC.objectID AND OC.constituentID = C.constituentID AND
               O.objectID =OI.objectID AND O.objectID =OT.objectID
WHERE (LOWER(C.visualBrowserNationality) LIKE LOWER('%${nationality}%')) AND
        (LOWER(OT.term) LIKE LOWER('%${style}%') AND OT.termType = 'Style') AND
        (O.beginYear >= ${beginYear} AND O.endYear <= ${endYear}) AND
        (LOWER(O.classification) LIKE LOWER('%${classification}%'))
ORDER BY O.endYear, O.title, O.attribution, C.lastName LIMIT ${offset}, ${limit};
```

4.2) CASE 2: having "style" filter, missing "nationality" filter,

```
SELECT DISTINCT O.title, O.attribution, O.endYear, O.objectID, OI.thumbURL
FROM objects O JOIN objects_images OI JOIN objects_terms OT
                   ON O.objectID =OI.objectID AND O.objectID =OT.objectID
WHERE (LOWER(OT.term) LIKE LOWER('%${style}%') AND OT.termType = 'Style') AND
        (O.beginYear >= ${beginYear} AND O.endYear <= ${endYear}) AND
        (LOWER(O.classification) LIKE LOWER('%${classification}%'))
ORDER BY O.endYear, O.title, O.attribution LIMIT ${offset}, ${limit};
```

4.3) CASE 3: having "nationality" filter, missing "style" filter

```
SELECT DISTINCT O.title, O.attribution, O.endYear, O.objectID, OI.thumbURL
FROM objects O JOIN objects_constituents OC JOIN constituents C JOIN objects_images OI
        ON O.objectID = OC.objectID AND OC.constituentID = C.constituentID AND O.objectID =OI.objectID
WHERE (LOWER(C.visualBrowserNationality) LIKE LOWER('%${nationality}%')) AND
        (O.beginYear >= ${beginYear} AND O.endYear <= ${endYear}) AND
        (LOWER(O.classification) LIKE LOWER('%${classification}%'))
ORDER BY O.endYear, O.title, O.attribution, C.lastName LIMIT ${offset}, ${limit};
```

4.4) CASE 4: missing "nationality" filter, missing "style" filter

```
SELECT DISTINCT O.title, O.attribution, O.endYear, O.objectID, OI.thumbURL
FROM objects O JOIN objects_images OI ON O.objectID =OI.objectID
WHERE (O.beginYear >= ${beginYear} AND O.endYear <= ${endYear}) AND
        (LOWER(O.classification) LIKE LOWER('%${classification}%'))
ORDER BY O.endYear, O.title, O.attribution LIMIT ${offset}, ${limit};
```

Route#5 – keywordSearch(req, res)

5) search relevant artworks by artwork's title OR/AND artist's name (in case-insensitive manner)

```
SELECT DISTINCT O.title, O.attribution, O.endYear, O.objectID, OI.thumbURL
FROM objects O JOIN objects_constituents OC JOIN constituents C JOIN objects_images OI
       ON O.objectID = OC.objectID AND OC.constituentID = C.constituentID AND O.objectID =OI.objectID
WHERE (LOWER(O.title) LIKE LOWER('%${artworkTitle}%')) AND (
        LOWER(O.attribution) LIKE LOWER('%${artistName}%') OR
        LOWER(O.attributionInverted) LIKE LOWER('%${artistName}%') OR
           LOWER(C.lastName) LIKE LOWER('%${artistName}%') OR
           LOWER(C.forwardDisplayName) LIKE LOWER('%${artistName}%') OR
           LOWER(C.preferredDisplayName) LIKE LOWER('%${artistName}%') )
ORDER BY O.title, O.attribution, C.preferredDisplayName, O.endYear LIMIT ${offset}, ${limit};
```

Route#6 – naughtySearchHeight(req, res)

6) find "painting" artworks by matching user's height (cm) with artwork's height (cm)

```
SELECT O.title, O.attribution, O.objectID, OI.thumbURL, OD.dimension, ABS(${height}-OD.dimension) AS deviation
FROM objects O JOIN objects_images OI JOIN objects_dimensions OD ON O.objectID =OI.objectID AND O.objectID = OD.objectID
WHERE O.classification = 'painting' AND OD.dimensionType = 'height' AND OD.unitName = 'centimeters'
ORDER BY ABS(${height}-OD.dimension), O.title LIMIT ${offset}, ${limit};
```

Route#7 – naughtySeachBirthYear(req, res)

7) find artworks completed in user's birthYear

```
SELECT O.title, O.attribution, O.objectID, O.endYear, ABS(${birthYear}-O.endYear) AS deviation, OI.thumbURL,
OD.dimension
FROM objects O JOIN objects_images OI JOIN objects_dimensions OD ON O.objectID = OI.objectID AND O.objectID =
OD.objectID
WHERE O.endYear IS NOT NULL AND OD.dimensionType = 'height'
ORDER BY ABS(${birthYear}-O.endYear), OD.dimension DESC  LIMIT ${offset}, ${limit};
```

Route#8 – analysisOverview(req, res)

8) find the top 80 popular terms for each of the 5 analysis category (i.e. School, Style, Theme, Technique, Keyword, Place Executed)

```
(SELECT OT.term AS name, COUNT(*) AS value FROM objects_terms OT WHERE OT.termType = 'Style'
 GROUP BY OT.term ORDER BY COUNT(*) DESC LIMIT 80)
UNION (SELECT OT.term AS name, COUNT(*) AS value FROM objects_terms OT WHERE OT.termType = 'School'
 GROUP BY OT.term ORDER BY COUNT(*) DESC LIMIT 80)
UNION (SELECT OT.term AS name, COUNT(*) AS value FROM objects_terms OT WHERE OT.termType = 'Technique'
 GROUP BY OT.term ORDER BY COUNT(*) DESC LIMIT 80)
UNION (SELECT OT.term AS name, COUNT(*) AS value FROM objects_terms OT WHERE OT.termType = 'Theme'
 GROUP BY OT.term ORDER BY COUNT(*) DESC LIMIT 80)
UNION (SELECT OT.term AS name, COUNT(*) AS value FROM objects_terms OT WHERE OT.termType = 'Keyword'
 GROUP BY OT.term ORDER BY COUNT(*) DESC LIMIT 80)
UNION (SELECT OT.term AS name, COUNT(*) AS value FROM objects_terms OT WHERE OT.termType = 'Place Executed'
 GROUP BY OT.term ORDER BY COUNT(*) DESC LIMIT 80);
```

Route #9 – analysisByType(req, res)

9) Search and return, in descending order, most popular terms a given the analysis category (i.e. Style, School, Theme, Technique, Keyword, Place Executed)

```
SELECT OT.term AS name, COUNT(*) AS value
FROM objects O JOIN objects_terms OT ON O.objectID = OT.objectID
WHERE OT.termType = '${analysisType}'
GROUP BY OT.term ORDER BY COUNT(*) DESC LIMIT ${offset}, ${limit};
```

Route #10 – portraitsAcrossTime (req, res)

10.1) within the specified year span, find and return artworks that have their theme of contents been defined as portraits

```
SELECT O.title, O.attribution, O.classification, O.objectID, OI.thumbURL, O.endYear
FROM objects O JOIN objects_images OI JOIN objects_terms OT ON O.objectID =OI.objectID AND O.objectID = OT.objectID
WHERE OT.visualBrowserTheme = 'portrait' AND ( O.endYear <= ${endYear} AND O.endYear >= ${beginYear}) AND
        classification = '${artworkClass}'
ORDER BY O.endYear LIMIT ${offset}, ${limit};
```

Appendix E: Query Optimization Timing

Relation Name	Additional `CREATE INDEXES ON`
objects	(beginYear, EndYear), (endYear), (classification)
constituents	(visualBrowserNationality), (lastName), (forwardDisplayName), (preferredDisplayName)
objects_constituents	none
objects_images	none
objects_dimensions	(dimensionType, unitName), (dimension)
objects_terms	(termType, term), (visualBrowserTheme)

Table E.1: List of manually created indexes of each relation.

routes	query (see Appendix D)	additional INDEX ON	Execution Time (ms) no index	Execution Time (ms) with index	x times faster
galleryoverview()	query 1.1)	(classification)	225	54	4
	query 1.2)	(visualBrowserNationality)	1231	1128	1
similarArtwok()	query 3.1)	(classification)	439	112	4
	query 3.2)	(termtype)	858	203	4
filterSearch()	query 4.1)	(beginYear, EndYear), (endYear), (classification), (visualBrowserNationality), (termType)	218	43	5
keywordSearch()	query 5)	(lastName), (forwardDisplayName), (preferDisplayName)	920	898	1
naughtySearchHeight()	query 6)	(classification), (typeUnit), (dimension)	839	106	8
naughtySearchBirthYear()	query 7)	(beginYear, endYear), (endYear), (classification)	1802	1514	1
analysisOverview()	query 8)	(termtype)	2387	348	7
analysisByType()	query 9)	(termtype)	391	60	7
portraitsAcrossTime()	query 10)	(termtype)	180	42	4
				Average Speed Improvement (x faster) =	**4**

Table E.2: Execution Time (computed by **`EXPLAIN ANALYZE`**) Improvement of queries due to query optimization

Page (Initial Loading of default (no caching))	Pre-Optimization	Post-Optimization
Home	4.25s	1.07s
Search	1.83s	1.48s
Analysis	4.25s	1.61s

Table E.3: Execution Time (computed by Google Chrome, tested on the same machine locally) Improvement of website pages due to query optimization