

# Inf 551 Spring 2020 Final Report

Jordan Minatogawa  
USC ID - 5084638775

## Project Overview

This system is a database querying application that takes an input query from a user and displays a response for that query based on relevance. A user is able to select one of three databases in which they would like to search. Their query is then applied to all of the tables in the selected database. Results are sorted based on the number of appearances of tokens in the user input query. The higher number of token appearances in a record, results in the record appearing higher in the list. Ties are broken by choosing the record that has the lesser number of different columns containing the search query tokens. If everything is equal, one of the records is chosen arbitrarily to appear higher in the list. Users are also able to navigate the database by clicking on query result hyperlinks on record columns that have foreign key associations.

## Databases

The three databases that the user will be able to query are the classicmodels, sakila, and world databases.

### Classicmodels database

The classicmodels database represents a store selling scale models of classic cars. It contains typical business data such as customers, products, sales orders, sales order line items, etc. There are eight tables with foreign key associations throughout.

### Sakila database

The sakila database is a schema modeling a DVD rental store. It contains 23 tables that highlight information about films, actors, rental information, etc. The tables have foreign key associations throughout.

### World database

The world database represents data about geographic units throughout the world. It has three tables that contain information about countries, cities, and country languages throughout the world. The tables have foreign key associations throughout.

# Components

## Landing page

# Database Query

classicmodels

sakila

world



Enter Query

The landing page of the application displays a minimalist design which allows the user to select the database which they would like to query and also allows the user to enter a query into a text box. The design is a play on the Google search web application and mimics the color scheme that the search giant uses.

## Query results page

Querying Database

classicmodels

sakila

world



Enter Query

Table Name	description	film_id	language_id	last_update	length	original_language_id	rating	release_year	rental_duration	rental_rate	replacement_cost	special_features	title
film	A Fateful Reflection of a Waitress And a Boat who must Discover a Sumo Wrestler in Ancient China	999	1	2006-02-15 05:03:42	101	None	R	2006	5	2.99	28.99	{Trailers', 'Deleted Scenes'}	ZOOLANDER FICTION

Table Name	description	film_id	language_id	last_update	length	original_language_id	rating	release_year	rental_duration	rental_rate	replacement_cost	special_features	title
film	A Astounding Epistle of a Database Administrator And a Explorer who must Find a Car in Ancient China	2	1	2006-02-15 05:03:42	48	None	G	2006	3	4.99	12.99	{Trailers', 'Deleted Scenes'}	ACE GOLDFINGER

Table Name	description	film_id	language_id	last_update	length	original_language_id	rating	release_year	rental_duration	rental_rate	replacement_cost	special_features	title
film	A Intrepid Panorama of a Robot And a Boy who must Escape a Sumo Wrestler in Ancient China	6	1	2006-02-15 05:03:42	169	None	PG	2006	3	2.99	17.99	{Deleted Scenes'}	AGENT TRUMAN

Table Name	description	film_id	language_id	last_update	length	original_language_id	rating	release_year	rental_duration	rental_rate	replacement_cost	special_features	title
film	A Action-Packed Tale of a Man And a Lumberjack who must Reach a Feminist in Ancient China	10	1	2006-02-15 05:03:42	63	None	NC-17	2006	6	4.99	24.99	{Trailers', 'Deleted Scenes'}	ALADDIN CALENDAR

The query results page displays records in the specified database. Each record also includes a hyperlink in each column that has a foreign key association. Clicking on the link follows the foreign key association and produces the record in the associated table.

The top of the page also shows which database is currently being queried and provides a way to further make queries by providing a database selector and an input text field to make a new query.

## Link page

Querying Database

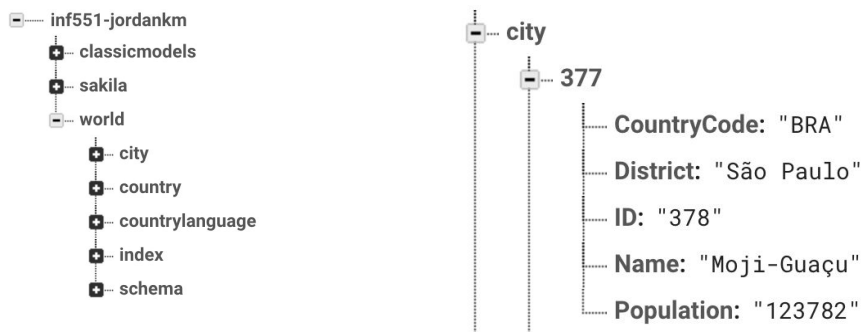
classicmodels sakila world

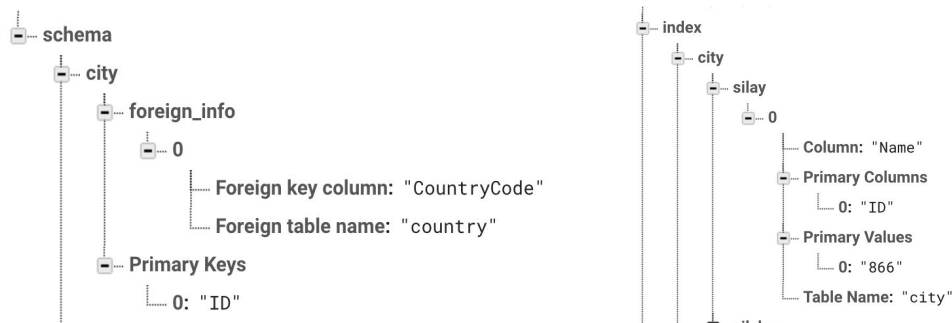
Enter Query

Table Name	addressLine1	addressLine2	city	contactFirstName	contactLastName	country	creditLimit	customerName	customerNumber	phone	postalCode	salesRepEmployeeNumber	state
customers	8489 Strong St.	None	Las Vegas	Jean	King	USA	71800.0	Signal Gift Stores	112	7025551838	83030	<a href="#">1166</a>	NV

The link page displays a single record that is reached from clicking a hyperlink on a response record. The general user interface is the same as the query results page and displays a record as well as a way to further query the databases at the top of the page. Users can continue to click on hyperlinks to follow their associations.

## Firebase





The databases are structured such that each database has its own node. Inside the database node contains the table nodes, which in turn contain all the records in that table. Each database also has an index node which contains an inverted index. There is also a schema node inside each database node which contains information about foreign keys and primary keys for each table.

## Implementation Details

### Import script

The import script works by taking the input database from the command line and querying the different tables within that database. For each table, the different columns, primary keys, and foreign keys are gathered by issuing a SQL query using the information\_schema for the database. This information is used to create the "schema" node for each database table in Firebase, which will later be used to create the foreign key hyperlinks. This primary key information is also used to update the database rules. Each primary key gets an index in Firebase so that it can be queried later by the Rails server during user interaction.

While going through each record, each column's value is tokenized and an inverted index is created for all entries. This allows us to easily query tokenized user input later. This inverted index is then stored as its own node in Firebase.

### User interface overview

The user interface is implemented using the Ruby on Rails web application framework. The Ruby on Rails framework may have provided more features than necessary for this project, but I chose this framework because I didn't want to implement any routing of requests myself, and I had some previous experience with the framework.

The overall architecture is a model-view-controller framework in which the model data is queried from the Firebase database. The Rails server ingests requests from the client and then issues its own requests to the Firebase database to get the data which is then packaged in a way that can be represented on the client.

## Landing page

To display the landing page, the client makes a request to the Rails server, which in turn makes a request to Firebase to get the different top-level nodes (or MySQL databases in this case). The client then sends back HTML/CSS to display on the web page.

## User Query

When a user inputs a query into the textbox of the landing page, the query string along with the selected database name is sent to the Rails server. The rails server then makes a series of requests to the Firebase database.

First the Rails server makes a request to Firebase to get all the database names that are stored. This is used to display the query input at the top left of the query results page shown in red below.

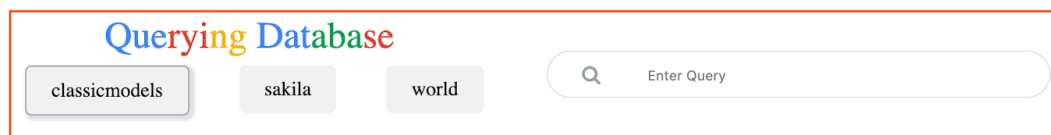
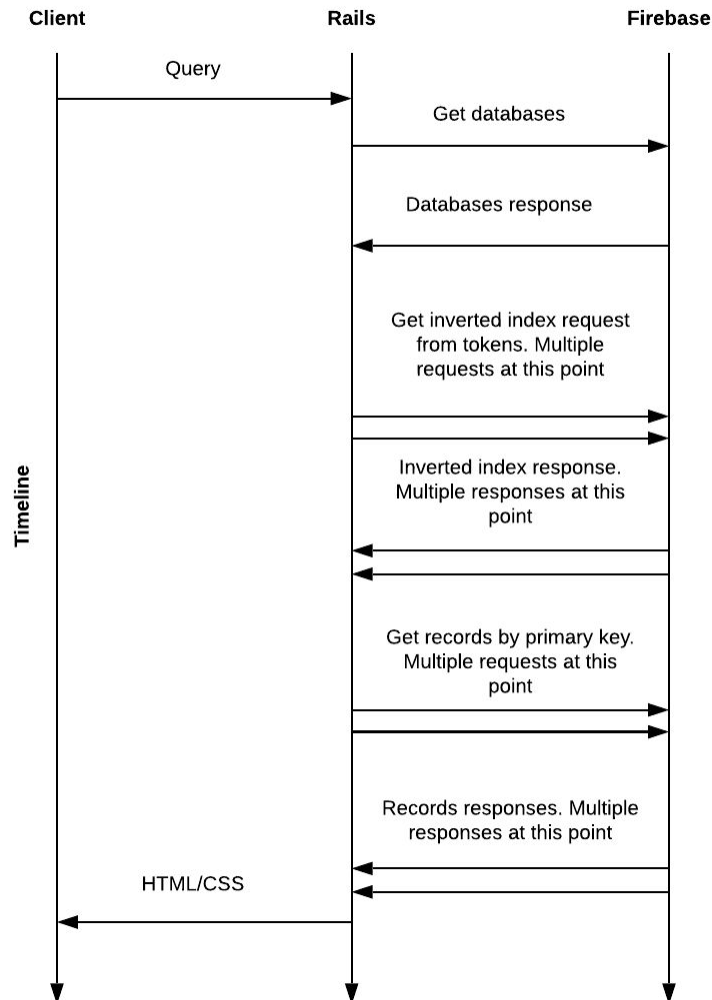


Table Name	addressLine1	addressLine2	city	contactFirstName	contactLastName	country	creditLimit	customerName
customers	Bank of China Tower	1 Garden Road	Central Hong Kong	Mike	Gao	Hong Kong	58600.0	King Kong Coll

The user query is then tokenized by white space and underscore. A Firebase query is then issued for each token using the inverted index node. Firebase responds with all the occurrences of the tokens. The rails server then groups the inverted index responses by primary key so that we can sort the results. At this point, the data that is received from Firebase is not complete, as the data stored in the inverted index does not contain the entire record. Thus to get the entire row/record we need to again issue a Firebase request using the records primary key to get the entire record information to display to the user. This primary key request is issued once for every distinct primary key that contains one of the tokens. A diagram for this request flow is shown below.



The double arrow lines indicate that this request is made multiple times.

## Navigating hyperlinks

Clicking through hyperlinks has a similar flow as the user query. The hyperlink is created in such a way that the url contains enough information for the controller to just plug into a Firebase request. For example, the hyperlink URL constructed by the user queries controller contains the database name, table name, primary keys, and primary columns that need to be displayed. Thus the controller just needs to construct a Firebase rest request and display the HTML/CSS to the user.

# Performance analysis

## Import script

The import script makes 4 SQL queries per table (get table columns, get all rows in table, get foreign keys, get primary keys). It then makes batch requests to send the table data to Firebase. This batch request process sends the table in batches of about 100KB. This was required initially because I was using a very large database that was causing issues when attempting to send to Firebase due to the size and thus I tried a batch approach. However, this is probably not needed for the databases I ended up using. The inverted index is also sent to Firebase in the batch approach due to the size reasons specified above.

## Landing page

Displaying the landing page is simple and just requires one Rails request and one Firebase request and thus is very fast.

## User query

The user query flow sends one Firebase request to get the databases to display in the query section in the top left hand corner. It also sends one request to Firebase to get the table names and one request per table to get the table schema. It then issues one Firebase request per table, per token. Thus if there is 4 tokens and 5 tables, it will issue 20 Firebase requests. Once the inverted index responses are received, the Rails server then makes one Firebase request per primary key that needs to be displayed to the user. Thus if there are 20 results to display to the user, then there will be 20 requests to Firebase to get the entire record for each result.

The total number of Firebase requests is thus  $2 + \text{number tables} + (\text{number tokens} * \text{number tables}) + \text{number of results}$ .

## Navigating hyperlinks

The hyperlink query flow sends one request to get the databases to display in the top left hand corner, just like the user query flow. It then issues a Firebase request to get the table schema for the result and also one request to get the actual record.

The total number of Firebase requests is 3.

## Improvements

There are a few improvements that I would like to make but don't have the time to implement. I will mention those improvements here for completeness.

### Import script

I would improve the import script by getting rid of the batch sends to Firebase which is probably not needed anymore and was just included due to legacy reasons described above. This would greatly reduce the number of Firebase requests as well as improve the speed of the script, since each separate connection adds latency.

Another improvement would also store the entire record in the inverted index. This way, we can remove one round trip per result during the user query flow since there is no need to query Firebase for the entire record. This would greatly reduce the number of Firebase requests and overall latency, but would increase the amount of storage required in Firebase. Overall I think this tradeoff would be worth it, as latency is often highly prioritized from a user perspective.

### Rails server

One improvement I would make to the Rails server is to batch requests to Firebase in a way that I could specify all the different primary keys for a table in a single request. This would cut down the amount of Firebase requests from "number of results to display" to just a single request. This would greatly reduce the latency. I don't think this behavior currently exists for the Firebase rest api, as you can only do an `limitToFirst`, `limitToLast`, `startAt`, `endAt`, `equalTo`. This would require the `equalTo` param to correctly handle a list of values. This feature could reduce the latency of my application greatly if available.

## Group work

All work was done by myself.