Data Visualization of the EchoQuery System

Ying Su

ying_su@brown.edu

Dec. 2016

1. Abstract

EchoQuery is a proof-of-concept voice-based database querying system initially implemented by Lyons, et al.[1] The original EchoQuery system allows the use of natural language based version of SQL command to query the relational database, and eventually it returns a voice-based result to the user. To make the interface more user friendly, in addition to the voice response returned from Alexa Voice Service (AVS), visual results (e.g. tables and charts) as well as a history navigation tree containing all the requests in current session are also provided to the user on screen. Users can navigate to any previous query node in the navigation tree and raise further requests based on the existing visual results.

2. Introduction

The current EchoQuery system has two frontends to the user: one is the Echo, a hands-free speaker connected to the AVS; the other is the screen, which displays the visualization of the query results.

Figure 1 shows the workflow of the EchoQuery system. The user's voice request is first converted to audio request by the Echo and then sent to the AVS, which makes the speech recognition and maps the audio request to different intent requests based on the intent schema and spoken input data (i.e. sample utterance and custom slots). The intent request is sent to the AWS Lambda, where the code of the EchoQuery application has been uploaded to. If the intent is recognized as a query intent, it will be translated into SQL query and delivered to the database stored at Amazon Relational Database Service (RDS) for execution; otherwise, if it is an intent that does not need to fetch any new data from the database, the request will be handled by the appropriate handler without reaching the RDS. After the response from RDS is returned to the EchoQuery application, on the one hand, the data in the response is visualized and displayed to the user on screen; on the other hand, the response is turned into a speechlet response recognizable to AVS, and AVS would further convert it to an audio response and eventually the Echo broadcasts the voice response to the user.

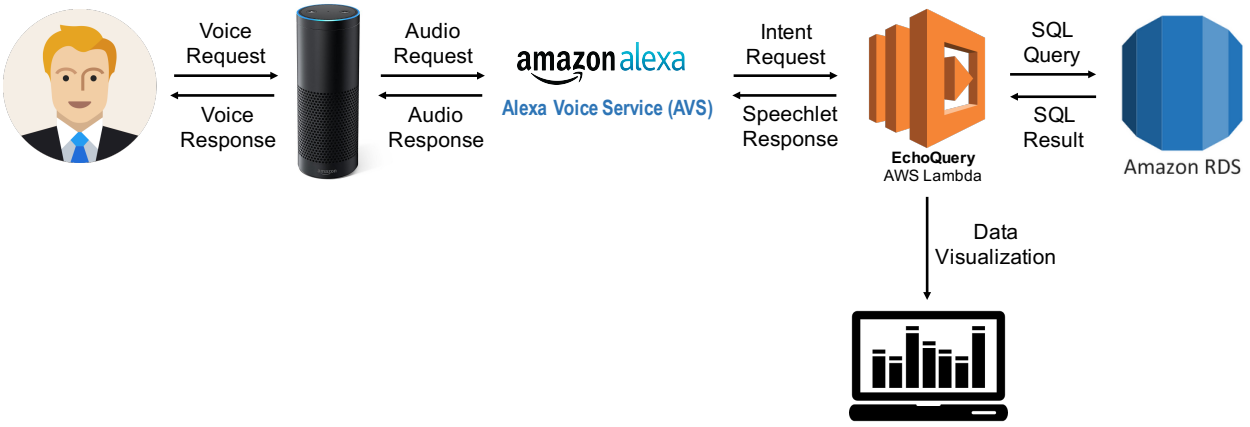


Figure 1. Workflow of the EchoQuery system

3. Implementation

3.1 Backend Implementation

The backend of the application is built using Java, and the package diagram is shown in Figure 2. The main functions of the backend include:

1) It recognizes the category of the intent request obtained from the AVS and invokes the corresponding intent handler to deal with the intent. Currently there are 12 categories of intents with 12 corresponding intent handlers, e.g. QueryHandler, RefineHandler, ClarifyHandler, PlotHandler, SliceHandler, etc.

2) It translates the natural-language-like query into SQL query. Specifically, if the intent is a query intent, the QueryHandler takes the intent object, and constructs a QueryRequest instance by accessing all the relevant slots from the intent to populate the instance. Then the QueryHandler calls Querier in the querier package to execute the request. When Querier executes the QueryRequest, it asks RequestTranslator to build an AST query object from the current QueryRequest, taking care of the selects, aggregations, where clauses, group-bys as well as inferred joins; then SqlFormatter is responsible for formatting the AST query into a valid SQL query string; eventually the SQL query is sent to the database by java.sql.Statement.

3) It maintains the "sessions" database and updates the two tables, "sessions" and "states", in this database. The schemas of the two tables are shown in Figure 3. The "sessions" table stores the results to the current query in this session, and the "states" table stores all the queries and their results in this session. With the "states" table, it is convenient to navigate back and forward through the historical queries.

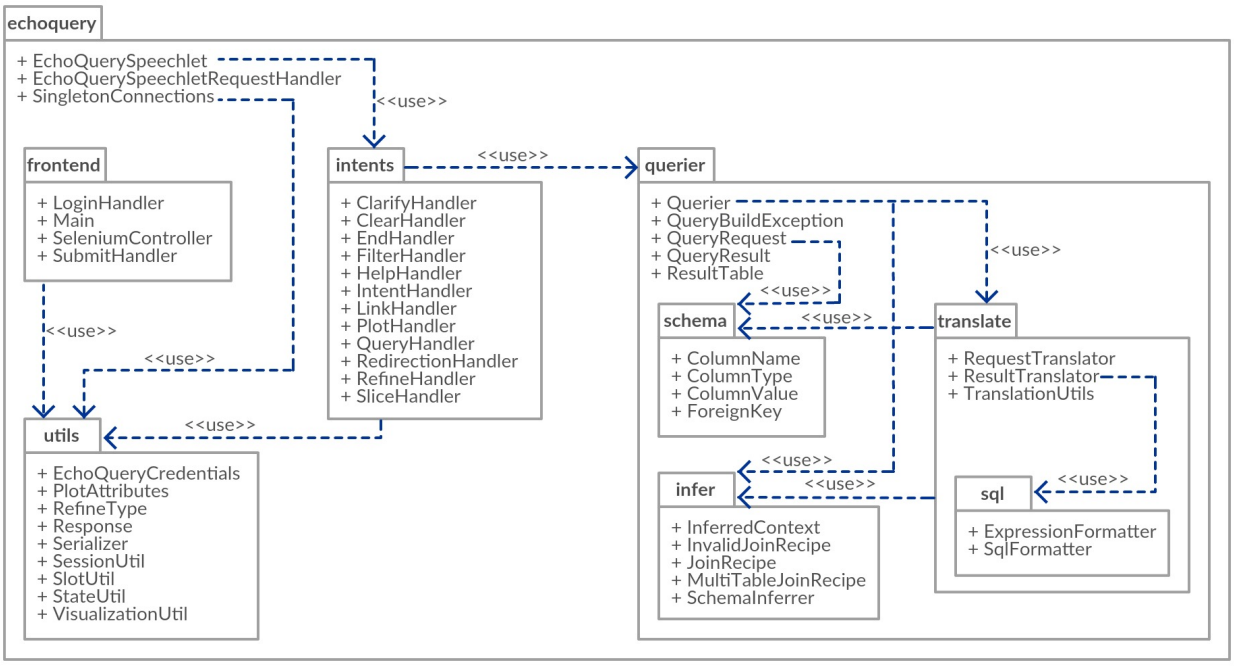


Figure 2. Package diagram of the backend

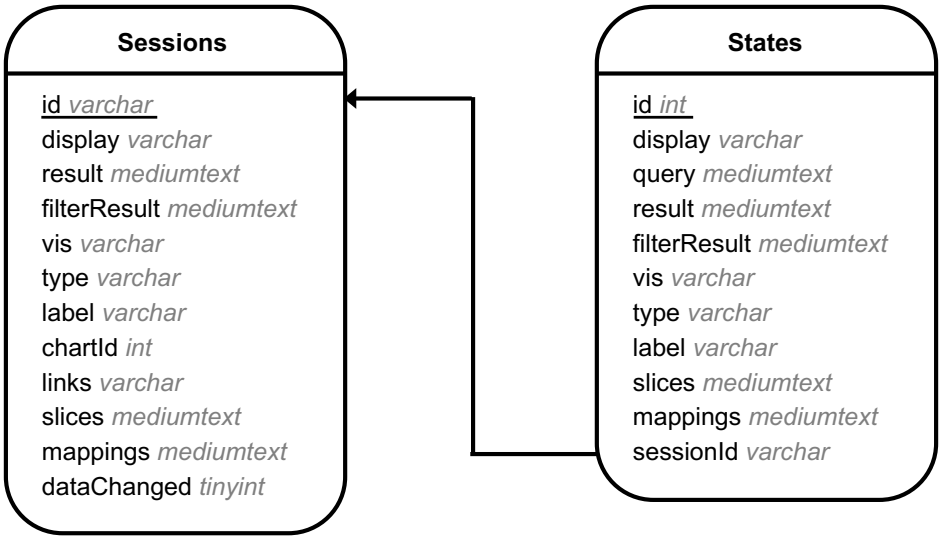


Figure 3. Schemas of table “sessions” and table “states”

3.2 User Interface Implementation

With respect to the user interface (UI), here we focus on the visual interface between the system and the user, as the voice-based interface has been fully discussed in Lyons’s paper[1].

The UI is implemented using ReactJS together with the Gulp + Browserify + Babelify to help realize the modularity and maintain the chain of dependencies between components[2,3]. It also adopts the Alt library to manage the dataflow within the JavaScript applications. The package structure of the JavaScript application is shown in Figure 4. As can be seen, there are mainly three packages in the JavaScript application: actions, components and stores.

```
js
|--actions
|   |--AudioActions.js
|   |--DisplayActions.js
|   |--SessionActions.js
|   |--WindowActions.js
|
|--components
|   |--index.jsx
|   |--App.js
|   |--DataView.js
|   |--TreeView.js
|   |--. . .
|
|--stores
|   |--AudioStore.js
|   |--SessionStore.js
|   |--WindowStore.js
|
|--utils
    |--SessionUtils.js
```

Figure 4. Package structure of the js folder

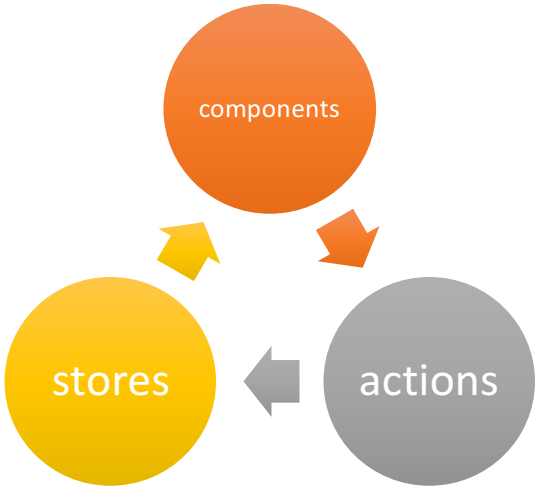


Figure 5. Dataflow direction among packages

As illustrated in Figure 5, in Alt architecture, the data flows from stores to components, then from components to actions, and back to stores to complete a cycle. Specifically, when a *component* in the components package mounts, it fetches data from stores, sets up its initial state and visualizes the data if necessary. The *component* has listeners listening to the *store* change events. Once it hears changes from *stores*, the *component* re-fetches data from *stores* and updates its own state and visualizations. An *action* method is usually invoked by a *component*, with data that needs to be saved into *stores* passed from *component* to the *action*. A *store* listens for events from *actions*. Upon hearing an *action* event, the *store* updates its internal data using the data passed from *action*, and emits a change event that some *components* are listening to.

The visual UI is started by typing "gulp" under the root package of the EchoQuery system, which will execute the gulpfile.js to initially invoke the App component. Through SessionUtils, session data is fetched from server using API calls, and the setDisplayData method in DisplayAction is invoked. The SessionStore hears the action event and receives data passed from the DisplayAction, then it updates its own data in the store. The DataView and TreeView components hear the change event from SessionStore, and they re-fetch the data from the SessionStore and update their visualizations, respectively. The data is updated from the server every 1000 ms. The workflow of above activities is sketched in Figure 6.

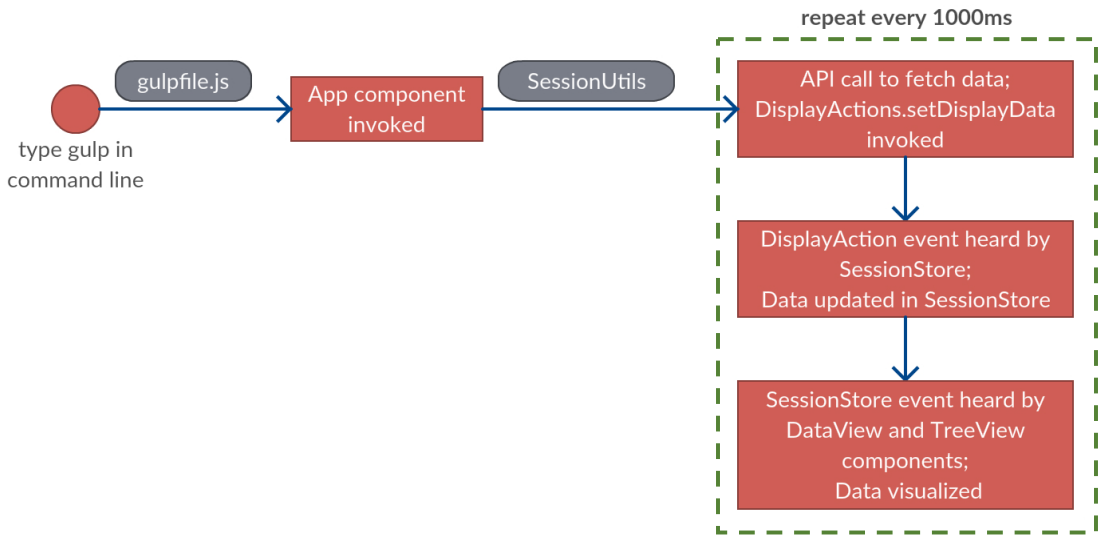


Figure 6. Workflow of the user interface activities

4. Contribution

My work in this project mainly focuses on the visual UI development and improvement, which involves programming in both backend and frontend.

4.1 Using Visualization Elements in Query

One of the new features in the UI is that users can use the visualization elements that they see on the screen in their future queries. These queries are classified as Slice Intents at AVS and are handled by the SliceHandler in the application. In this current version, EchoQuery supports selecting single or multiple (consecutive or inconsecutive) visualization elements (e.g. selecting slices in a pie chart by color or selecting bars in a histogram by the bar's position), then the selected data is visualized in its original format (i.e. if it was a pie chart in the previous query, then the selected slices will still be shown in the pie chart format in the new result).

The request for selecting a single element is of the following form:

{Select} (the) {MappingKeyOne} [part|slice|bar]

Here the {Select} slot includes verbs "select", "choose", "show", "get" and "list", and the {MappingKeyOne} refers to the color name for the pie chart (e.g. "aqua", "darkblue", etc.) and bar position (e.g. "first", "second", etc.) for the histogram.

The request for selecting multiple elements can be categorized into two basic types:

1) selecting consecutive elements: the request falls into one of the following formats:

{Select} (from) (the) {MappingKeyStart} to (the) {MappingKeyEnd} [part|slice|bar] (e.g. "select from the aqua to the darkblue slice");

{Select} (the) {MappingKeyRangeFromStart} [parts|slices|bars] (e.g. "select the first four bars" or "select the leftmost five bars");

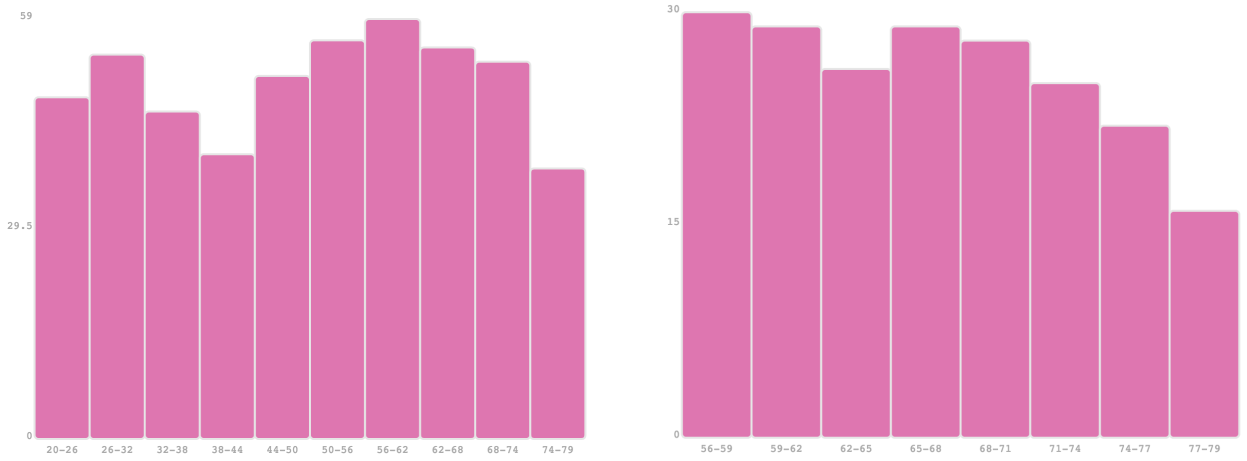
{Select} (the) {MappingKeyRangeToEnd} [parts|slices|bars] (e.g. "select the last three bars" or "select the rightmost four bars");

2) selecting inconsecutive elements: currently it is allowed to select at most 6 inconsecutive elements, and the format could be:

{Select} (the) {MappingKeyOne} (and) {MappingKeyTwo} [parts|slices|bars] (e.g. "select the aqua and bluedark slices" or "select the first and third bars");

{Select} (the) {MappingKeyOne} {MappingKeyTwo} {MappingKeyThree} {MappingKeyFour} {MappingKeyFive} (and) {MappingKeySix} [parts|slices|bars] (e.g. "select the aqua, darkblue, darkgreen, purple, hotpink, and navy slices");

The specific choices for each slot (expression inside the curly braces) can be referred to in the IntentSchema.json file and the custom-slots folder under speech-assets package. Figure 7 shows an example of the visualization comparison before and after the query selecting a series of consecutive bars of a histogram. All the visualization examples used in this document are based on the sample data in the mimic database, the main schemas of which is shown in Figure 8.



(a) (b)

Figure 7. (a) is the histogram showing the age distribution (20–79) of all patients in the mimic database; (b) is the histogram on age distribution (56–79), which is the visualization result after the query "select the rightmost four bars" is executed based on the data shown in (a).

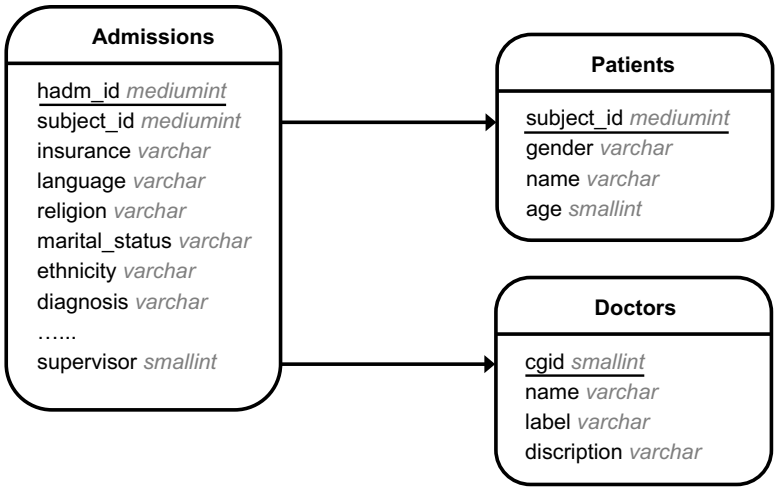


Figure 8. Main schemas of the mimic database

4.2 Improvement on the History Navigation Tree

The history navigation tree shows all the queries in the current session, with each query represented as a node in the tree. With the navigation tree, it is easy to determine the relationship among queries, e.g. the parent-child relationship indicates that the child query is a refine query based on the results of the parent query, while the sibling relationship indicates that the two sibling queries use the same set of data.

All the nodes in the navigation tree are properly numbered and labeled, so users can easily use the node number to navigate back and forward through the nodes. The navigation can be divided into two categories:

1) simply navigating to the last or next (if available) node: the request is of the following format:

{NavigateBack} (e.g. “back”, “go back”, and “back to last chart”);

{NavigateForward} (e.g. “next”, “next chart”, “forward”, and “go forward”);

With these requests, users can navigate one step back or forward at a time until they reach the first or the last node in the tree.

2) navigating to any designated node in the tree: the request is of the following format:

(go back|go|back|return) to [table|chart|graph] {ChartIndex} (e.g. “go to chart two”, “back to table one”, etc);

Here the {ChartIndex} should be a number, with which the node has already been existing in the navigation tree. With this function, not only the data visualization will be switched to the

designated node, but also the result data in the sessions table will be replaced by the designated query result data, which means that users can raise further requests based on the data set that is newly navigated to.

In addition to the navigation function, the navigation tree also provides thumbnails for each query node, which can conveniently remind the user of any previous query results without the necessity of really navigating to that node. The thumbnail sketch will display by its node when the mouse hovers over the node. An example of the thumbnail function is shown in Figure 9.

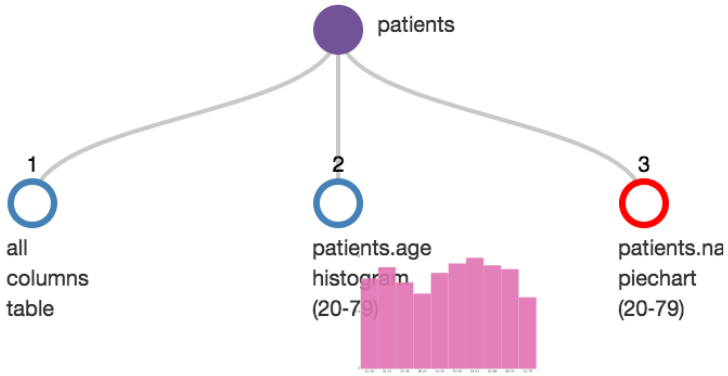


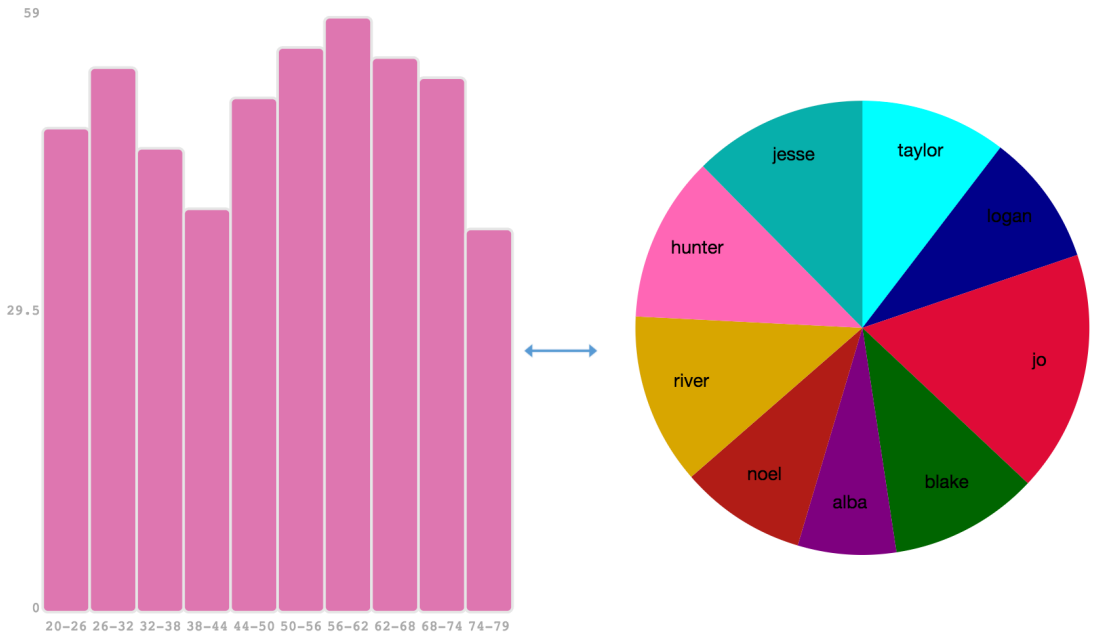
Figure 9. History navigation tree with thumbnail shown by the node. The red circle indicates that node 3 is the current node, and the thumbnail of node 2 shows up when the mouse hovers over node 2.

4.3 Using One Graph as Filter for Another Linked Graph

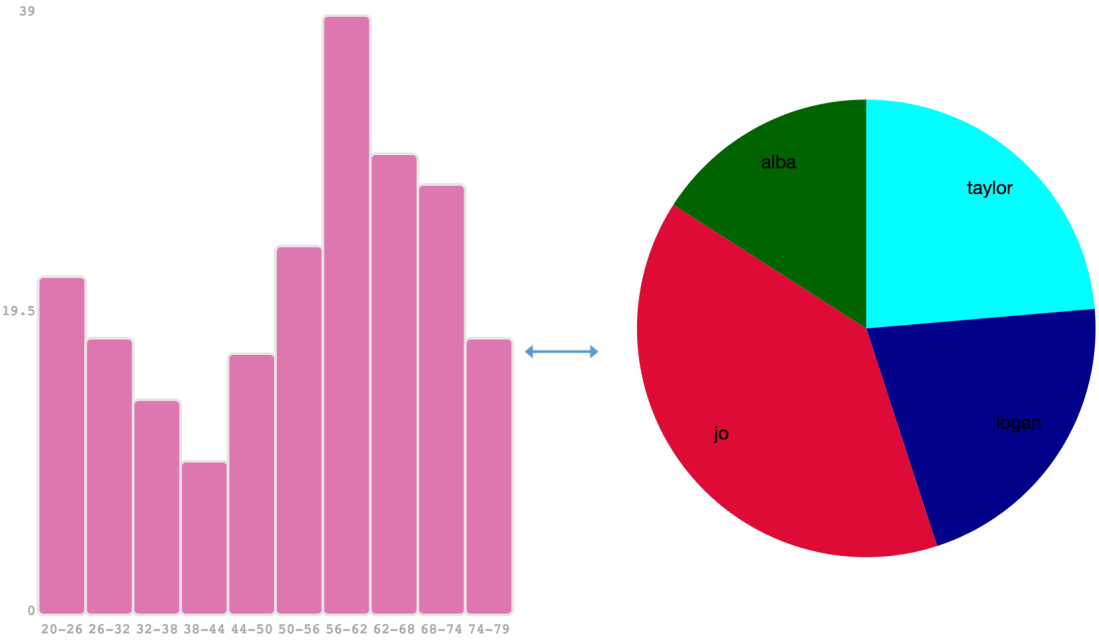
In the current version of EchoQuery, two visualizations can be linked with each other and displayed together using the Link Intent in the following format:

{Link} [chart|graph] {PlotOneID} [to|with|and] (chart|graph) {PlotTwoID} (e.g. "link chart two with chart three");

With two graphs linked together, when users select specific visualization elements in one graph, the other graph will also use this filter to alter the appearance of its own visualization. However, the prerequisite to use this function is that the two linked graphs must use the same set of data, i.e. the two graphs are siblings in the navigation tree, otherwise the filter for one graph might not be applicable for the other one. Figure 10 shows an example of using this function. As can be seen, the query not only performs the selection on the slices of the pie chart, it also updates the histogram using the filtered data.



(a)



(b)

Figure 10. (a) shows the linkage between the histogram on age distribution of all patients and the pie chart on name distribution of all patients; (b) shows the visualization result after the query "select the aqua, darkblue, crimson, and purple slices" is executed based on the data result shown in (a).

5. Conclusions

Besides the voice-based interface, the visual UI is added to the EchoQuery system, which makes it possible for the user to see the data visualization, raise further requests using the visualization elements, and navigate back and forward among the already existing query results.

6. Acknowledgement

I wish to thank my advisor Professor Ugur Cetintemel for his excellent guidance on this project. I also would like to thank Professor Carsten Binning and Xiaocheng Wang for all their helpful suggestions to my work.

7. References

[1] G. Lyons et al. Making the Case for Query-by-Voice with EchoQuery. Proceedings of the 2016 International Conference on Management of Data: 2129–2132, 2016.

[2] http://chris.house/blog/grunt-configuration-for-react-browserify-babelify

[3] https://tylermcginnis.com

Appendix

Here are some instructions on how to get the EchoQuery system work:

1. Install mysql to your local machine. To browse the current database, run **bin/connect-to-db** under the root package to get access to the database. The password for the database is in the file EchoQuery/src/main/java/echoquery/utils/EchoQueryCredentials.java.

2. Install node.js. To build the sample utterances, run **node build-utterances.js** under the EchoQuery/speech-assets package.

3. Install gulp. You will need gulp to start the frontend.

4. Make sure to install all the libraries listed under the "dependencies" in the EchoQuery/package.json file to the root package; install all the libraries listed under the "dependencies" in the EchoQuery/src/main/resources/js/package.json file to the EchoQuery/src/main/resources/js package.

5. Set up AWS Lambda function to write a skill for the Amazon Echo using the Alexa SDK. You can learn to set it up by following any one of the samples in this link: https://github.com/amzn/alexa-skills-kit-java/tree/master/samples/src/main/java

After you set up a sample Lambda function following the instruction, set up your own EchoQuery Lambda function. To create a jar file for the EchoQuery project, you can directly run **bin/assemble-jar**, and you can find the jar file (echoquery-1.1-jar-with-dependencies.jar) in the EchoQuery/target directory.

6. After step 5 is finished, run a test on the Alexa skill page, e.g. enter "get patients" in the text box, and click "Ask EchoQuery", you will find your userId in the Lambda Request box. Copy everything after the string "amzn1.ask.account." of this userId, and insert it into the id column in the sessions table in sessions database. Note: please do not include "amzn1.ask.account." when you save your id into the database!

7. To start the frontend, type **gulp** in the command line under the root project directory. Open Chrome browser and enter "localhost:4567/user/" + the string after "amzn1.ask.account." of your userId into the address box.

8. In current version of EchoQuery, a sequence of sample queries could be:

   1) get patients

   2) plot a piechart on name

   3) plot a histogram on age

   4) select the rightmost three bars

   5) back to chart two

   6) link chart two and chart three

   7) select aqua and darkblue parts