CSCI 4890: Computer Projects

An Interactive Web UI for Approximate Query Processing

Author: Ryan Lalchand

Advisor: Dr. Feng Yu

What is AQP?

- Approximate Query Processing is a query optimization method for estimating results from databases, especially important in processing Big Data
- This is accomplished by randomly sampling a small percentage of the original dataset, and estimating the ground truth, or actual value, of the data
- As the amount of data that we generate and store grows year after year, this technique becomes increasingly important to cut down on query runtime and associated costs
- While the AQP result may not be the exact value, in many cases the relative error is negligible, but the speedup can be astounding - by orders of magnitude

How do we implement AQP?

- Correlated Sample Synopses (Yu et al. 2013), or CS2, speeds up search query results, gives precise join query estimations, and minimizes storage costs
- We could run the sampling in real-time, however, that is inefficient as it adds calculation time to each query and does not cache the results for future use
    - Especially inefficient on join queries which require several full table scans
- Instead, we sample and calculate estimates of the database tables ahead of time, and store them separately for future reference

The Problem

- Currently, the Data Lab has databases running on the remote Sarah Cloud server
- These can be accessed using the Apache Hadoop + Hive command line interface remotely using SSH (secure shell) and a VPN (virtual private network)
- This is not ideal for anyone outside of the lab, as they would need to install the VPN, get SSH access, and be able to navigate the command line – not exactly user-friendly

Where do I come in?

- There is a way around this complicated setup: a web portal!
- For my capstone project, I created a full-stack web app that enables anyone around the world to be able to run commands that test AQP's power versus traditional queries
- We used the Transaction Protocol Council Benchmark H (TPC-H), a database generator that can scale from a few megabytes up to thousands of gigabytes and produce complex queries with just eight tables, which can contain millions of records

Technologies Used

- I chose Node.js for the server-side simply because I am familiar with it
  - It doesn't hurt that I only need to know JavaScript to run frontend and backend code
  - While only capable of running tasks on a single thread, Node, like JS, has an asynchronous, non-blocking, event-driven loop
  - This means that if you need to make multiple database calls, Node won't sit and wait for the database to answer – it will start calling other databases and doing other important tasks
  - Node also allows you to choose the packages or modules you deem necessary using the Node Package Manager (npm), making it far more lightweight and efficient than something like a Java web app

Technologies Used

- Node.js is fairly limited in its capabilities however, as it is just a command-line interface for JS, which used to live solely in the browser
- This is where Express.js comes in:
  - Express is a minimal unopinionated web app framework or middleware (sits in the middle between the client and server)
  - It facilitates HTTP requests and the flow of data between the databases, the server, and the client
  - It makes life simpler than trying to manually write pure Node to do basic tasks

Technologies Used

- The locally hosted MySQL Community Edition server is populated using Dr. Yu's TPC-H database generator
- This is accessed using the Node.js MySQL module installed via npm

```
1   var mysql = require("mysql");
2   const { credentials } = require("./credentials/credentials.js");
3
4   var connection1GB = mysql.createConnection({
5     host: credentials.host,
6     user: credentials.user,
7     password: credentials.password,
8     database: "1GB",
9     multipleStatements: true, //needed to run multiple queries
10    local_infile: true, //to mitigate error when loading data from local file
11    supportBigNumbers: true, //speaks for itself
12  });
13
14  connection1GB.connect(function (err) {…
21  });
```

Code Review

- When the user hits submit, the frontend makes two separate asynchronous calls using the Fetch() API
- Fetch allows values to be modified dynamically on the webpage without needing to refresh or redirect

```
async function getAQPanswer() {
  try {
    const AQPresponse = await fetch("/AQPanswer", {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify({
        query: document.getElementById("query").value,
        size: document.getElementById("size").value,
      }),
    });

    const AQPanswer = await AQPresponse.json();

    document.getElementById("AQPanswer").innerHTML = JSON.stringify(
      AQPanswer[0]["count(*)"] * 100
    );

    return AQPanswer[0]["count(*)"] * 100;
  } catch (error) {
    console.log(error);
  }
}
```

Code Review

- When the fetch request is sent from the frontend, the backend parses the body for the requested query and database size
- The chosen query is sent to the database of choice and once returned is sent as a response to the client

```
94  app.post("/AQPanswer", function (request, response) {
95    let query = request.body.query;
96    let queryFile = __dirname + "/queries/" + query + "AQP.sql";
97    let size = request.body.size;
98
99    console.log({ query, queryFile, size });
100
101   if (query == "") {
102     response.send(__dirname + "/public/index.html");
103   } else {
104     let queryString = fs.readFileSync(queryFile).toString();
105     if (size == "1GB") {
106       connection1GB.query(queryString, function (err, result) {
107         if (err) {
108           console.log(err);
109           return;
110         }
111         console.log("result: " + JSON.stringify(result));
112
113         var resultJSON = Object.assign({}, result);
114
115         response.send(resultJSON);
116       });
```

Security Measures

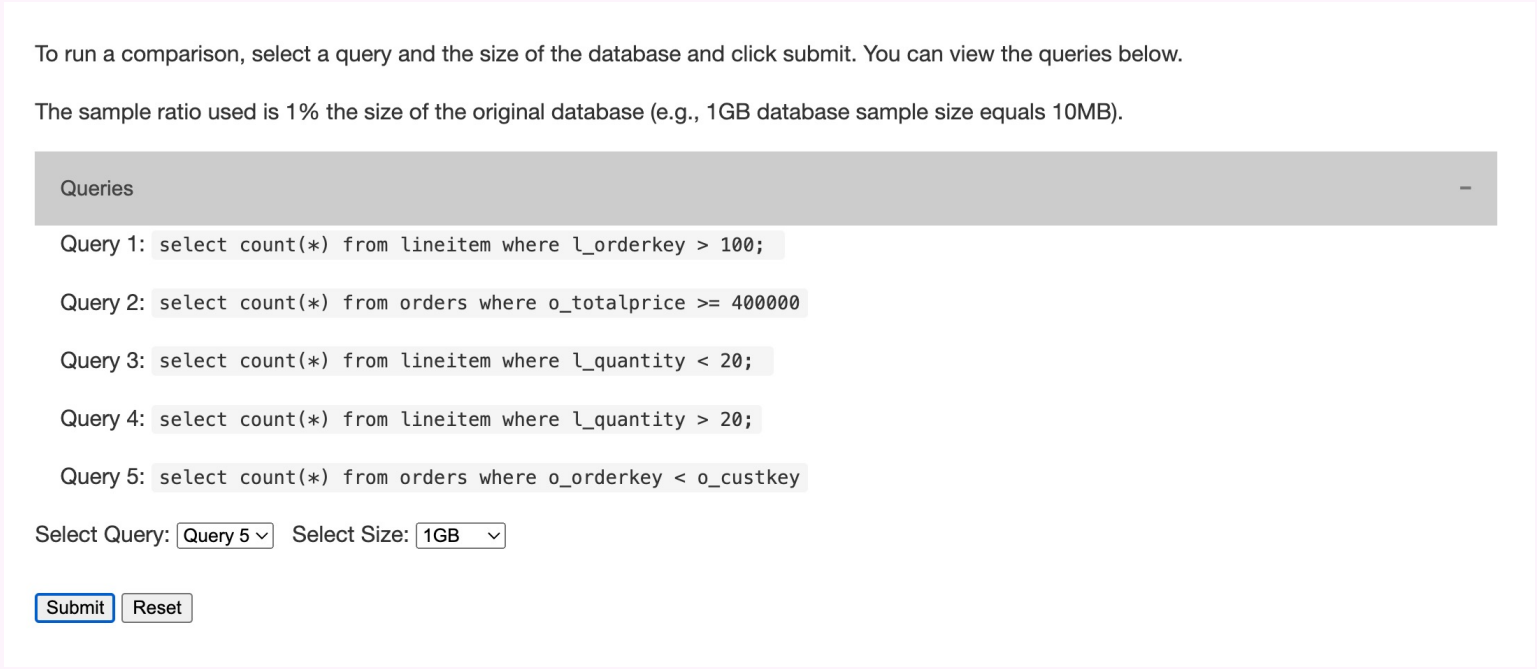
- In addition to not allowing query manipulation on the frontend, the server is also hosted securely via https
- The web authentication key and certificate are generated manually using command line tool mkcert for local hosting

```
const https = require("https");

https
  .createServer(
    {
      key: fs.readFileSync("./credentials/key.pem"),
      cert: fs.readFileSync("./credentials/cert.pem"),
    },
    app
  )
  .listen(3000, function () {
    console.log("Go to https://localhost:3000/");
  });
```

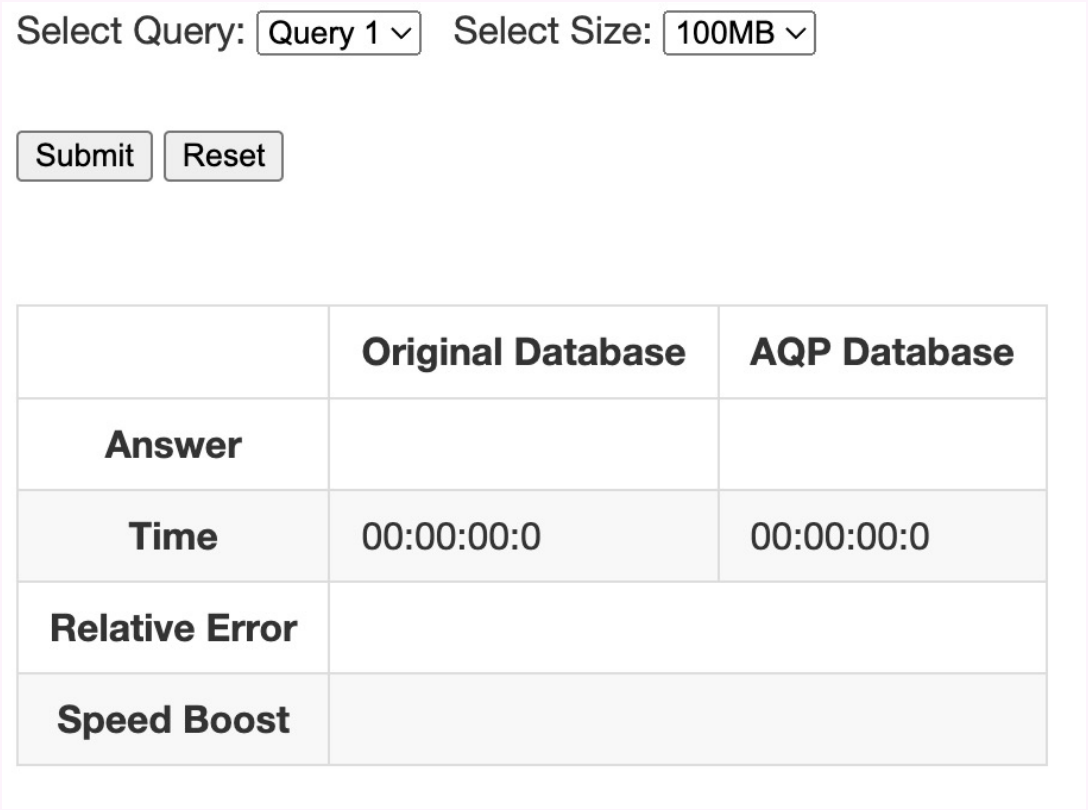
The UI

- To maximize space, queries are displayed within an accordion dropdown
- Users are currently able to select from 5 queries and 2 database sizes (100MB and 1GB)



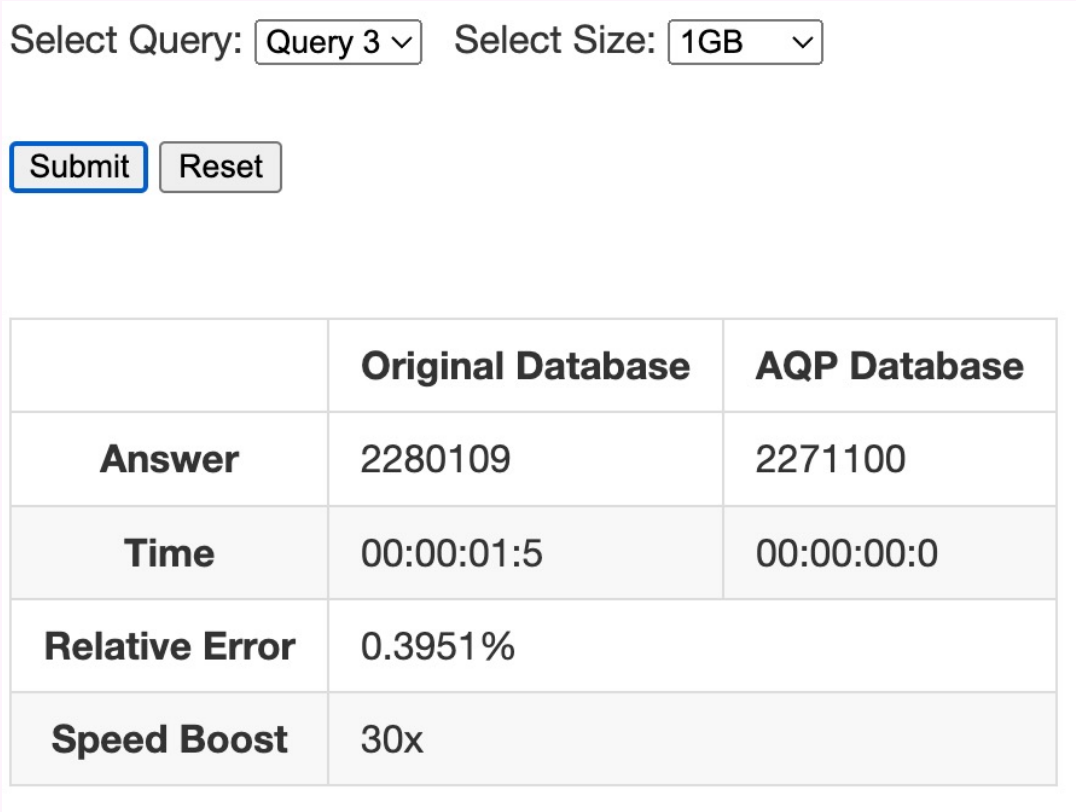
The UI

- Once a query and database size have been selected and the user hits submit, the timers begin
- When the queries return, each table cell is populated with the associated value



The UI

- Time is measured in tenths of a second, but for AQP, you'll notice that the majority of the time, the result is calculated quicker than that!
- MySQL command line estimates the average time for AQP queries to be about .05 seconds, thus the speed boost is calculated simply by multiplying the original time by 2



	Original Database	AQP Database
Answer	2280109	2271100
Time	00:00:01:5	00:00:00:0
Relative Error	0.3951%	
Speed Boost	30x	

Float like a butterfly, sting like a bee

- Node and AQP go hand-in-hand, both extremely quick, efficient, and lightweight alternatives to older methods
- Node.js is the perfect backend, combining a low-profile single-threaded event loop with its async nature means that this server can run quickly, efficiently, and without the need for multicore processing for many simultaneous tasks (extremely useful for virtualization)
- AQP demonstrates tremendous efficiency gains over traditional querying with low relative error

Challenges

- Setting up MySQL databases locally proved to be more difficult than anticipated
  - My setup script mysteriously returned no errors, yet some tables were left unpopulated
- Creating a Single Page App (SPA) that required a data manipulation/exchange relationship, without a JavaScript framework, was certainly a poor choice on my behalf
- The EasyTimer.js library's precision is limited to 1/10 of a second, so exact time calculations are hindered

Future Work

- Refactor/rebuild in React.js
- Redesign using a CSS framework like Bootstrap
- Allow custom query entry, ensuring security against SQL injection attacks
- Write custom install script
- Containerize to run on the Sarah Cloud

Acknowledgements

- To paraphrase Isaac Newton, I stand on the shoulders of giants
- This project would not have been possible if not for the work that Dr. Yu, the YSU Data Lab, and all the students that came before me have done
  - Most notably, I'd like to acknowledge Dr. Yu's TPC-H database generator and David Wilson's Master's thesis, "Correlated Sample Synopsis on Big Data"
- I would be remiss to not mention the amazing open-source software community that made my job infinitely easier, namely the MySQL Community Edition server, the OpenJS Foundation (Node/Express), GitHub, npm, Visual Studio Code, and countless others

Thank you!

- Source code available on my GitHub
- Information about the Data Lab is available at datalab.ysu.edu
- Any questions?