CIS 550 Course Project: 1033 & 2020

TABLE OF CONTENTS

INTRODUCTION

1. Team Members

Name	Email	GitHub
Robby Ballard	rdbb@seas.upenn.edu	P0bbn
Yolanda Shao	yoshao@seas.upenn.edu	yoshao
Keith Svendsen	ksven@seas.upenn.edu	ksve
Puneet Uppal	puppal@seas.upenn.edu	p-uppal

2. Guest Credentials

Host	cis550-proj.cl8t7peslmhx.us-east-1.rds.amazonaws.com
Port	3306
User	guest
Password	pT2&gS3@By
Primary Schema	final_project

Secondary Schema (For Testing)	final_project2

3. Application Description

We are looking to foremost compare conflict and protest data this year in the US with police data over the last two decades. Through the Department of Defense's Law Enforcement Support Office (LESO), local and state law enforcement agencies can request surplus military equipment through LESO’s 1033 Program. This equipment can range anywhere from popcorn machines to parkas, and from grenade launchers to mine-resistant armored transportation. Use of this equipment has been cited as a leading indicator, and perhaps cause, of an increasingly militaristic mindset amongst America's law enforcement agencies. This “warrior’s” mindset was on full display during the Summer of 2020 when overwhelmingly peaceful protests in response to the murder of George Floyd were met with often violent responses from the nation’s myriad law enforcement agencies. Our group’s project seeks to determine whether or not a correlation exists between the use of 1033 equipment in a given county and an increase in the use of force against protesters in the same county. Users can search for the correlation between these two factors, as well as demographic information for every county, such as racial makeup, income, unemployment, and education, for every county in the United States.

ARCHITECTURE

1. Technologies

MySQL for database management
Amazon AWS RDS for database hosting
Python/Jupyter Notebook for data cleansing and processing
MySQL Workbench for data ingestion, schema configuration, and testing
React.js and Chart.js for front-end development
Node.js and Express for back-end development

2. Application Pages

Home/Country: The landing page with some country data, protest facts, and links to get to the other pages.
State Search/Lookup: The page where you can look up a state, which will call up the data, charts, and graphs for that state.
Similar Counties: This is a page where you can search for a county, and it will populate with the N most similar counties based on population size.
Demographics: This is a page where you can see demographic information about the protest and police data in the US.
Income Inequality: This is a page where you can see socioeconomic information about the protest and police data in the US.

3. Application Features

- **Menu bar** for navigation
- **Drop-down menu** for users to select a state
- **Drop-down submenu** populated with all of the counties in the user-selected state
- **Selecting a given county** will take users to a page that displays demographic, 1033 request, and protest data for said county, as well as for the county's home state and the nation as a whole, for comparison's sake. A list of the 10 counties with the most similar population will also be displayed.
- **Graphs/charts** showing nationwide averages for 1033, demographic, and protest data
- **Dynamic charts** that change to match new inputs
- **Tables** showing different 1033, state, and protest data

DATA

1. American Community Survey (ACS)

- https://www.kaggle.com/muonneutrino/us-census-demographic-data
- The ACS is an ongoing demographics survey conducted by the U.S. Census Bureau. It gathers information annually from 52 U.S. states and territories. The data here is taken from the DP03 and DP05 tables of the 2017 ACS 5-year estimates.

- 3220 rows, 37 columns; 0.6MB; 2017; statistics: 3220 counties, 324473330 people

2. General Services Administration (GSA) 1033 Program

- https://data.world/datasets/1033-program
- This CSV file represents $1.5B of ~$4.8B distributed by the GSA's 1033 Program, for years 2006-2014. All manner of equipment is here from helicopters to rifles, with prices, nomenclature, and NSNs. The 1033 Program allows local and state law enforcement agencies to receive surplus military equipment from the Department of Defense at little or no cost. Note that the police departments do not pay for the value of the gear, just the shipping and storage en route.
- 243492 rows, 12 columns; 22MB; 2006–2014

3. US Crisis Monitor

- https://acleddata.com/special-projects/us-crisis-monitor/
- “ACLED systematically collects the dates, actors, locations, fatalities, and types of all political violence and demonstration events in the US using a methodology consistent with our global coverage of conflict and disorder around the world.”
- 17675 rows, 28 columns; 3MB; 2020/05/24–2020/11/21; event breakdown: 16331 protests, 753 riots, 539 strategic developments, 42 violence against civilians, 5 battles

DATABASE

1. Conversion Process

Data from the three input .csv files was cleaned and processed into four .csv output files. Each .csv file was ingested using the MySQL Workbench table data import wizard. An overview of the process for each table is shown here:

- ‘acs2017_county_data.csv’ county data → ‘county.csv’ → ‘**county**’ table with 3220 instances

- ‘acs2017_county_data.csv’ demographic data → ‘demographic.csv’ → ‘**demographic**’ table with 3220 instances
- ‘1033prog.csv’ → ‘equipment.csv’ → ‘**equipment**’ table with 241062 instances
- ‘USA_2020_Nov21-1.csv’ → ‘protest.csv’ → ‘**protest**’ table with 17670 instances

2. Data Cleaning

1. **For each dataset, review the following:**
    a. Numerical distributions and data types (check summary statistics and look for abnormalities)
    b. Missing data (look for NULL values and note any column with too many NULLs)
    c. Check values in non-numeric columns (look for any string that may represent NULL)
2. **Since only ACS has county id, extract county id, state and county from ACS as table County**
3. **To improve mapping rate of countries, for county in each of the three datasets (Demographic/County, Protest, and Equipment):**
    a. Capitalize county and state
    b. Remove the word COUNTY, BOROUGH, and CENSUS AREA from county ACS
    c. Correct Spanish letters to English in ACS
    d. Standardize the use of -, ‘, . and the use of space in county name
    e. Standardize the use of ST and SAINT in county name
    f. See Jupyter Notebook for details
4. **Add county id to Protest and Equipment by joining with table County on county and state.** Mapping rates improved to over 99% from less than 50% after taking all the steps in the previous step
5. **For equipment, classify equipment to “weapon” and “other” by examining column Item Name**

3. SQL DDL

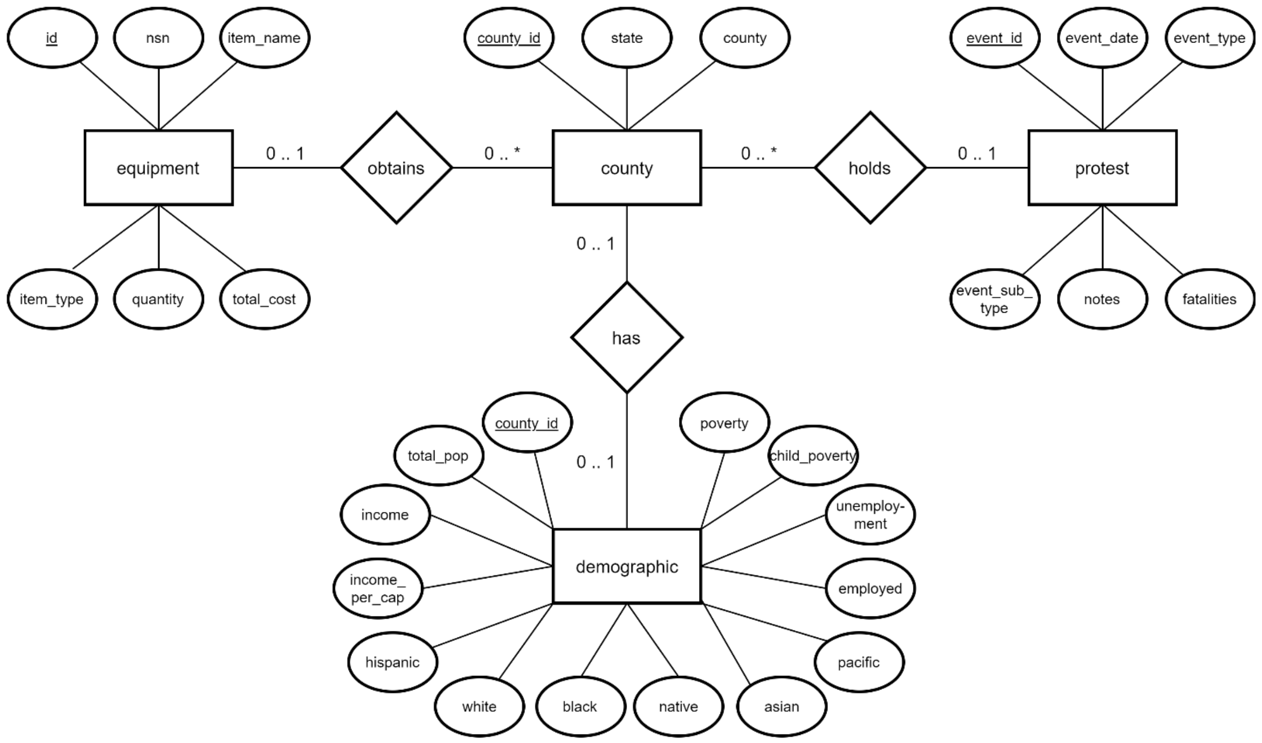
- **CREATE TABLE `county`** (`county_id` int NOT NULL, `state` varchar(20) NOT NULL, `county` varchar(21) NOT NULL, PRIMARY KEY (`county_id`), KEY `county_state_index` (`state`), KEY `county_county_index` (`county`)) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

- **CREATE TABLE `demographic`** (`county_id` int NOT NULL, `total_pop` int DEFAULT NULL, `hispanic` double DEFAULT NULL, `white` double DEFAULT NULL, `black` double DEFAULT NULL, `native` double DEFAULT NULL, `asian` double DEFAULT NULL, `pacific` double DEFAULT NULL, `income` int DEFAULT NULL, `income_per_cap` int DEFAULT NULL, `poverty` double DEFAULT NULL, `child_poverty` double DEFAULT NULL, `employed` int DEFAULT NULL, `unemployment` double DEFAULT NULL, PRIMARY KEY (`county_id`), KEY `demo_total_pop_index` (`total_pop`), KEY `demo_hispanic_index` (`hispanic`), KEY `demo_white_index` (`white`), KEY `demo_black_index` (`black`), KEY `demo_native_index` (`native`), KEY `demo_asian_index` (`asian`), KEY `demo_pacific_index` (`pacific`), KEY `demo_income_per_cap_index` (`income_per_cap`), KEY `demo_poverty_index` (`poverty`), KEY `demo_unemployment_index` (`unemployment`), CONSTRAINT `fk_demo_cid` FOREIGN KEY (`county_id`) REFERENCES `county` (`county_id`)) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

- **CREATE TABLE `equipment`** (`id` int NOT NULL, `nsn` varchar(22) DEFAULT NULL, `item_name` varchar(200) DEFAULT NULL, `item_type` varchar(6) DEFAULT NULL, `quantity` int DEFAULT NULL, `total_cost` decimal(19,2) DEFAULT NULL, `county_id` int DEFAULT NULL, PRIMARY KEY (`id`), KEY `fk_equip_cid_idx` (`county_id`), KEY `equip_nsn_index` (`nsn`), KEY `equip_item_name_index` (`item_name`), KEY `equip_item_type_index` (`item_type`), KEY `equip_quantity_index` (`quantity`), KEY `equip_total_cost_index` (`total_cost`), CONSTRAINT `fk_equip_cid` FOREIGN KEY (`county_id`) REFERENCES `county` (`county_id`)) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

- **CREATE TABLE `protest`** (`event_id` int NOT NULL, `event_date` date DEFAULT NULL, `event_type` varchar(26) DEFAULT NULL, `sub_event_type` varchar(34) DEFAULT NULL, `notes` varchar(1581) DEFAULT NULL, `fatalities` int DEFAULT NULL, `county_id` int DEFAULT NULL, PRIMARY KEY (`event_id`), KEY `fk_protest_cid_idx` (`county_id`), KEY `protest_event_date_index` (`event_date`), KEY `protest_event_type_index` (`event_type`), KEY `protest_event_sub_type_index` (`sub_event_type`), KEY `protest_fatalities_index` (`fatalities`), CONSTRAINT `fk_protest_cid` FOREIGN KEY (`county_id`) REFERENCES `county` (`county_id`)) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

4. ER Diagram



5. Normal Form Definition & Justification

Third Normal Form (3NF): For every relation scheme R and for every X → A that holds over R, either A ∈ X (trivial), or X is a superkey for R, or A is a member of some key for R.

Boyce-Codd Normal Form (BCNF): For every relation scheme R and for every X → A that holds over R, either A ∈ X (trivial) or X is a superkey for R.

A justification for each relation is given here:

- **County:** The candidate keys are county_id and state, county. The functional dependencies are county_id → state, county and state, county → county_id. The relation violates BCNF since the left-hand side of each dependency is not a superkey. However, it satisfies 3NF since the right-hand side of each dependency is a member of some key.
- **Demographic:** The key is county_id and the functional dependencies are county_id → total_pop, hispanic, white, black, native, asian, pacific, income, income_per_cap, poverty, child_poverty, employed, unemployment. The

relation satisfies BCNF since the left-hand side of each dependency is a superkey of county_id.

- **Equipment:** The key is id and the functional dependencies are id → nsn, item_name, item_type, quantity, total_cost, county_id. The relation satisfies BCNF since the left-hand side of each dependency is a superkey of id.
- **Protest:** The key is event_id and the functional dependencies are event_id → event_date, event_type, sub_event_type, notes, fatalities, county_id. The relation satisfies BCNF since the left-hand side of each dependency is a superkey of event_id.

In summary, the county relation satisfies 3NF but not BCNF, while the demographic, equipment, and protest relations satisfy BCNF. Since BCNF is strictly stronger than 3NF, all relations satisfy 3NF. Therefore, the database is in 3NF.

QUERIES

1. Query 1: Country Statistics

Returns protest and equipment statistics for the entire country.
Columns: country, population, peaceful, non_peaceful, fatalities, peaceful_per_100K, non_peaceful_per_100K, equip_qty, weapon_qty, equip_cost, weapon_cost, cost_per_equip, cost_per_weapon, equip_cost_per_cap, weapon_cost_per_cap

```
WITH d1 AS (SELECT "United States" AS country, SUM(total_pop) AS population
FROM demographic),
p1 AS (SELECT COUNT(*) AS peaceful FROM protest WHERE
sub_event_type="Peaceful protest"),
p2 AS (SELECT COUNT(*) AS non_peaceful FROM protest WHERE
event_type<>"Strategic Developments" AND sub_event_type<>"Peaceful protest"),
p3 AS (SELECT COUNT(*) AS num_fatalities FROM protest WHERE
event_type<>"Strategic Developments" AND fatalities>0),
e1 AS (SELECT SUM(quantity) AS equip_qty, SUM(total_cost) AS equip_cost FROM
equipment WHERE item_type<>"WEAPON"),
e2 AS (SELECT SUM(quantity) AS weapon_qty, SUM(total_cost) AS weapon_cost
FROM equipment WHERE item_type="WEAPON")
```

```
SELECT country, population, peaceful, non_peaceful, num_fatalities AS "fatalities",
ROUND((peaceful/population*100000),2) AS peaceful_per_100K,
ROUND((non_peaceful/population*100000),2) AS non_peaceful_per_100K, equip_qty,
weapon_qty, equip_cost, weapon_cost, ROUND((equip_cost/equip_qty),2) AS
cost_per_equip, ROUND((weapon_cost/weapon_qty),2) AS cost_per_weapon,
ROUND((equip_cost/population),2) AS equip_cost_per_cap,
ROUND((weapon_cost/population),2) AS weapon_cost_per_cap FROM d1, p1, p2, p3,
e1, e2;
```

2. Query 2: State/Territory Statistics

Returns protest and equipment statistics for each state/territory.
Columns: state, state, population, peaceful, non_peaceful, peaceful_per_100K, non_peaceful_per_100K, equip_qty, weapon_qty, equip_cost, weapon_cost, cost_per_equip, cost_per_weapon, equip_cost_per_cap, weapon_cost_per_cap

```
WITH d1 AS (SELECT state, SUM(total_pop) AS population FROM demographic d
JOIN county c ON d.county_id=c.county_id GROUP BY state),
p1 AS (SELECT state, COUNT(*) AS peaceful FROM protest p JOIN county c ON
p.county_id=c.county_id WHERE sub_event_type="Peaceful protest" GROUP BY
state),
p2 AS (SELECT state, COUNT(*) AS non_peaceful FROM protest p JOIN county c ON
p.county_id=c.county_id WHERE event_type<>"Strategic Developments" AND
sub_event_type<>"Peaceful protest" GROUP BY state),
e1 AS (SELECT state, SUM(quantity) AS equip_qty, SUM(total_cost) AS equip_cost
FROM equipment e JOIN county c ON e.county_id=c.county_id WHERE
item_type<>"WEAPON" GROUP BY state),
e2 AS (SELECT state, SUM(quantity) AS weapon_qty, SUM(total_cost) AS weapon_cost
FROM equipment e JOIN county c ON e.county_id=c.county_id WHERE
item_type="WEAPON" GROUP BY state)
SELECT d1.state, population, peaceful, non_peaceful,
ROUND((peaceful/population*100000),2) AS peaceful_per_100K,
ROUND((non_peaceful/population*100000),2) AS non_peaceful_per_100K, equip_qty,
weapon_qty, equip_cost, weapon_cost, ROUND((equip_cost/equip_qty),2) AS
cost_per_equip, ROUND((weapon_cost/weapon_qty),2) AS cost_per_weapon,
ROUND((equip_cost/population),2) AS equip_cost_per_cap,
ROUND((weapon_cost/population),2) AS weapon_cost_per_cap FROM d1 LEFT
OUTER JOIN p1 on d1.state=p1.state LEFT OUTER JOIN p2 on d1.state=p2.state LEFT
OUTER JOIN e1 ON d1.state=e1.state LEFT OUTER JOIN e2 ON d1.state=e2.state;
```

3. Query 6: Highest Population Counties

Returns demographic, protest, and equipment statistics for the 10 highest population counties in a particular state.

Columns: county, population, income_per_cap, poverty, unemployment, events, events_per_100K, item_qty, item_cost, cost_per_item, cost_per_cap

```
WITH d1 AS (SELECT c.county_id, county, total_pop AS population, income_per_cap,
poverty, unemployment FROM demographic d JOIN county c ON
d.county_id=c.county_id WHERE state="Pennsylvania" GROUP BY county_id),
p1 AS (SELECT c.county_id, COUNT(*) AS num_events FROM protest p JOIN county c
ON p.county_id=c.county_id WHERE state="Pennsylvania" GROUP BY county_id),
e1 AS (SELECT c.county_id, SUM(quantity) AS item_qty, SUM(total_cost) AS item_cost
FROM equipment e JOIN county c ON e.county_id=c.county_id WHERE
state="Pennsylvania" GROUP BY county_id)
SELECT county, population, income_per_cap, poverty, unemployment, num_events
AS "events", ROUND((num_events/population*100000),2) AS events_per_100K,
item_qty, item_cost, ROUND((item_cost/item_qty),2) AS cost_per_item,
ROUND((item_cost/population),2) AS cost_per_cap FROM d1 LEFT OUTER JOIN p1
ON d1.county_id=p1.county_id LEFT OUTER JOIN e1 on e1.county_id=d1.county_id
ORDER BY population DESC LIMIT 10;
```

4. Query 8: Most Similar Counties

Returns demographic, protest, and equipment statistics for a particular county and the 10 most similar counties based on population size.

Columns: county, state, population, income_per_cap, poverty, unemployment, events, events_per_100K, item_qty, item_cost, cost_per_item, cost_per_cap

```
WITH d1 AS (SELECT c.county_id, state, county, total_pop AS population,
income_per_cap, poverty, unemployment FROM demographic d JOIN county c ON
d.county_id=c.county_id GROUP BY county_id),
p1 AS (SELECT c.county_id, COUNT(*) AS num_events FROM protest p JOIN county c
ON p.county_id=c.county_id GROUP BY county_id),
e1 AS (SELECT c.county_id, SUM(quantity) AS item_qty, SUM(total_cost) AS item_cost
FROM equipment e JOIN county c ON e.county_id=c.county_id GROUP BY
county_id),
```

o1 AS (SELECT d1.county_id, population, income_per_cap, poverty, unemployment, num_events, (num_events/population*100000) AS events_per_100K, item_qty, item_cost, (item_cost/population) AS item_cost_per_cap FROM d1 LEFT OUTER JOIN e1 ON d1.county_id=e1.county_id LEFT OUTER JOIN p1 on p1.county_id=d1.county_id WHERE state="Pennsylvania" AND county="Philadelphia")
SELECT county, state, population, income_per_cap, poverty, unemployment, num_events AS "events", ROUND((num_events/population*100000),2) AS events_per_100K, item_qty, item_cost, ROUND((item_cost/item_qty),2) AS cost_per_item, ROUND((item_cost/population),2) AS cost_per_cap FROM d1 LEFT OUTER JOIN p1 ON d1.county_id=p1.county_id LEFT OUTER JOIN e1 on d1.county_id=e1.county_id ORDER BY ABS((SELECT population FROM o1) - population) LIMIT 11;

5. Query 11: Additional Country Statistics

Returns the number of events per 100K people and the item cost per capita for the entire country, majority/minority white counties, and the lowest/highest poverty counties.

Columns: events_per_100K_national, white, non_white, low_poverty, high_poverty, item_cost_per_cap_national, white, non_white, low_poverty, high_poverty

WITH p1 AS (SELECT COUNT(*) AS num_events FROM protest),
pop1 AS (SELECT SUM(total_pop) AS population FROM demographic),
e1 AS (SELECT SUM(total_cost) AS item_cost FROM equipment),
d2 AS (SELECT county_id, total_pop FROM demographic WHERE (hispanic+black+native+asian+pacific)<=white),
p2 AS (SELECT COUNT(*) AS num_events FROM protest p JOIN d2 ON p.county_id=d2.county_id),
pop2 AS (SELECT SUM(total_pop) AS population FROM d2),
e2 AS (SELECT SUM(total_cost) AS item_cost FROM equipment e JOIN d2 ON e.county_id=d2.county_id),
d3 AS (SELECT county_id, total_pop FROM demographic WHERE (hispanic+black+native+asian+pacific)>white),
p3 AS (SELECT COUNT(*) AS num_events FROM protest p JOIN d3 ON p.county_id=d3.county_id),
pop3 AS (SELECT SUM(total_pop) AS population FROM d3),

```
e3 AS (SELECT SUM(total_cost) AS item_cost FROM equipment e JOIN d3 ON
e.county_id=d3.county_id),
d4 AS (SELECT county_id, total_pop FROM demographic ORDER BY poverty LIMIT
322),
p4 AS (SELECT COUNT(*) AS num_events FROM protest p JOIN d4 ON
p.county_id=d4.county_id),
pop4 AS (SELECT SUM(total_pop) AS population FROM d4),
e4 AS (SELECT SUM(total_cost) AS item_cost FROM equipment e JOIN d4 ON
e.county_id=d4.county_id),
d5 AS (SELECT county_id, total_pop FROM demographic ORDER BY poverty DESC
LIMIT 322),
p5 AS (SELECT COUNT(*) AS num_events FROM protest p JOIN d5 ON
p.county_id=d5.county_id),
pop5 AS (SELECT SUM(total_pop) AS population FROM d5),
e5 AS (SELECT SUM(total_cost) AS item_cost FROM equipment e JOIN d5 ON
e.county_id=d5.county_id)
SELECT ROUND((p1.num_events/pop1.population*100000),2) AS
events_per_100K_national, ROUND((p2.num_events/pop2.population*100000),2) AS
white, ROUND((p3.num_events/pop3.population*100000),2) AS non_white,
ROUND((p4.num_events/pop4.population*100000),2) AS low_poverty,
ROUND((p5.num_events/pop5.population*100000),2) AS high_poverty,
ROUND((e1.item_cost/pop1.population),2) AS item_cost_per_cap_national,
ROUND((e2.item_cost/pop2.population),2) AS white,
ROUND((e3.item_cost/pop3.population),2) AS non_white,
ROUND((e4.item_cost/pop4.population),2) AS low_poverty,
ROUND((e5.item_cost/pop5.population),2) AS high_poverty FROM p1, pop1, e1, p2,
pop2, e2, p3, pop3, e3, p4, pop4, e4, p5, pop5, e5;
```

PERFORMANCE EVALUATION

1. Overview

We tried to “push” SELECT/PROJECT statements to the base relations wherever possible to minimize intermediate results. Pushing PROJECT statements reduces the size of result tuples, while pushing SELECT statements reduces the number of tuples. JOINs were placed in the FROM clause using the JOIN … ON syntax rather than being embedded in the WHERE clause. We also tried to reorder JOINs such that smaller tables were joined first. DISTINCT, UNION, and ORDER BY were only used if necessary. TEXT fields were converted to VARCHAR to reduce the size of tuples. All primary keys, foreign keys, and referenced columns were indexed to reduce the cost of read operations. Finally, queries were rewritten if the measured execution time was significantly worse than the expected execution time.

2. Detailed Optimization Steps

1. **Converted all TEXT fields to VARCHAR**
    a. **County:** \`state\` text NOT NULL, \`county\` text NOT NULL → \`state\` varchar(20) NOT NULL, \`county\` varchar(21) NOT NULL
    b. **Equipment:** \`nsn\` text, \`item_name\` text, \`item_type\` text → \`nsn\` varchar(22) DEFAULT NULL, \`item_name\` varchar(200) DEFAULT NULL, \`item_type\` varchar(6) DEFAULT NULL
    c. **Protest:** \`event_type\` text, \`sub_event_type\` text, \`notes\` text → \`event_type\` varchar(26) DEFAULT NULL, \`sub_event_type\` varchar(34) DEFAULT NULL, \`notes\` varchar(1581) DEFAULT NULL
2. **Indexed primary keys**
    a. **County:** PRIMARY KEY (\`county_id\`)
    b. **Demographic:** PRIMARY KEY (\`county_id\`)
    c. **Equipment:** PRIMARY KEY (\`id\`)
    d. **Protest:** PRIMARY KEY (\`event_id\`)
3. **Indexed foreign keys**
    a. **Equipment:** KEY \`fk_equip_cid_idx\` (\`county_id\`)
    b. **Protest:** KEY \`fk_protest_cid_idx\` (\`county_id\`)
4. **Indexed referenced columns**
    a. **County:** KEY \`county_state_index\` (\`state\`), KEY \`county_county_index\` (\`county\`)
    b. **Demographic:** KEY \`demo_total_pop_index\` (\`total_pop\`), KEY \`demo_hispanic_index\` (\`hispanic\`), KEY \`demo_white_index\` (\`white\`), KEY \`demo_black_index\` (\`black\`), KEY \`demo_native_index\` (\`native\`), KEY \`demo_asian_index\` (\`asian\`), KEY \`demo_pacific_index\` (\`pacific\`), KEY \`demo_income_per_cap_index\` (\`income_per_cap\`), KEY \`demo_poverty_index\` (\`poverty\`), KEY \`demo_unemployment_index\` (\`unemployment\`)
    c. **Equipment:** KEY \`fk_equip_cid_idx\` (\`county_id\`), KEY \`equip_nsn_index\` (\`nsn\`), KEY \`equip_item_name_index\` (\`item_name\`), KEY

`equip_item_type_index` (`item_type`), KEY `equip_quantity_index` (`quantity`), KEY `equip_total_cost_index` (`total_cost`)

d. **Protest:** KEY `protest_event_date_index` (`event_date`), KEY `protest_event_type_index` (`event_type`), KEY `protest_event_sub_type_index` (`sub_event_type`), KEY `protest_fatalities_index` (`fatalities`)

5. **Rewrote queries 9a, 9b, and 9c.** Query 9a is shown as an example and the differences (before and after) are highlighted in yellow.

Before:

WITH date1 AS (SELECT event_date FROM protest GROUP BY event_date),
p1 AS (SELECT event_date FROM protest p JOIN county c ON p.county_id=c.county_id WHERE state="Pennsylvania"),
p2 AS (SELECT date1.event_date, IFNULL(COUNT(p1.event_date),0) AS num_events FROM date1 LEFT OUTER JOIN p1 ON date1.event_date=p1.event_date GROUP BY date1.event_date ORDER BY date1.event_date)
SELECT * FROM p2;

After:

WITH date1 AS (SELECT DISTINCT event_date FROM protest),
p1 AS (SELECT event_date, COUNT(*) AS num_events FROM protest p JOIN county c ON p.county_id=c.county_id WHERE state="Pennsylvania" GROUP BY event_date),
p2 AS (SELECT date1.event_date, IFNULL(num_events,0) AS num_events FROM date1 LEFT OUTER JOIN p1 ON date1.event_date=p1.event_date GROUP BY date1.event_date ORDER BY date1.event_date)
SELECT * FROM p2;

3. Query Timings

All tests were conducted on a db.t2.micro (1vCPU, 20 GiB SSD, 1GiB RAM) database instance using the default configuration parameters from the “AWS Database Set-up Tutorial”. Each database query with a pre-optimization execution time of at least 0.3 seconds was tested 10 times per schema. The (unoptimized) schema ‘final_project2’ was used to record pre-optimization times, while the (optimized) schema ‘final_project’ was used to record post-optimization times. The min, max, and average times (in seconds) for each query are summarized in the table on the next page. Detailed timing data can be found in the ‘Query Timings.xlsx’ spreadsheet on GitHub.

	Pre-Optimization			Post-Optimization			
Query	**Min**	**Max**	**Avg**	**Min**	**Max**	**Avg**	**% Change**
Query 1	0.406	0.485	0.449	0.281	0.344	0.297	-33.8
Query 2	0.922	1.094	0.987	0.609	0.688	0.636	-35.6
Query 8	0.469	0.531	0.503	0.328	0.359	0.331	-34.1
Query 9a	8.938	9.547	9.108	0.063	0.110	0.091	-99.0
Query 9b	8.906	9.954	9.269	0.047	0.110	0.077	-99.2
Query 9c	9.094	9.313	9.175	0.047	0.078	0.061	-99.3
Query 10a	0.312	0.375	0.333	0.187	0.203	0.195	-41.3
Query 10b	0.843	0.906	0.864	0.578	0.610	0.595	-31.1
Query 10c	0.328	0.344	0.339	0.094	0.110	0.102	-70.0
Query 10d	0.312	0.359	0.328	0.078	0.094	0.080	-75.7
Query 11	1.125	1.188	1.150	0.468	0.500	0.481	-58.2
Query 12c	0.313	0.375	0.339	0.328	0.344	0.334	-1.3
Query 12d	0.297	0.375	0.331	0.328	0.359	0.336	1.4
Query 13c	0.281	0.344	0.311	0.312	0.344	0.320	3.1
Query 13d	0.281	0.359	0.326	0.312	0.329	0.325	-0.4
						Avg:	**-45.0**

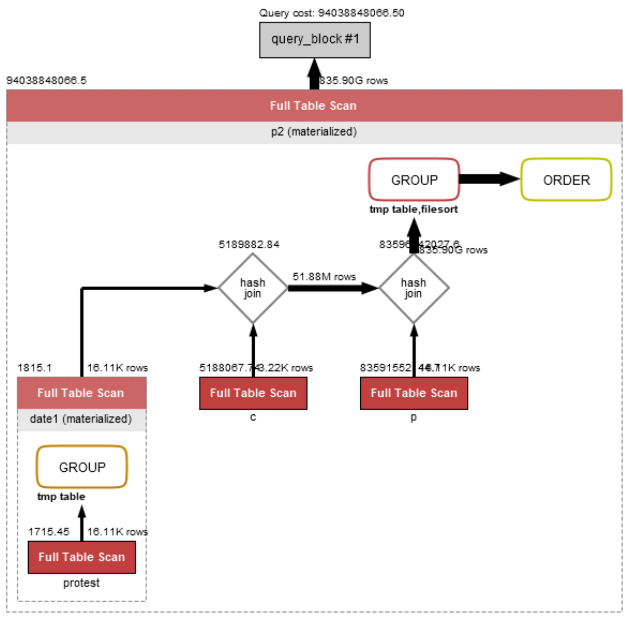
4. Analysis

Changing the ‘total_cost’ column in ‘equipment’ from type DECIMAL(19,2) to type int saved ~0.3 seconds in some of the more complex queries. There was no observed performance difference between DECIMAL(19,2) and DECIMAL(10,2), which has a smaller range. We wanted cost to have a decimal representation, so DECIMAL(19,2) was kept despite the cost.

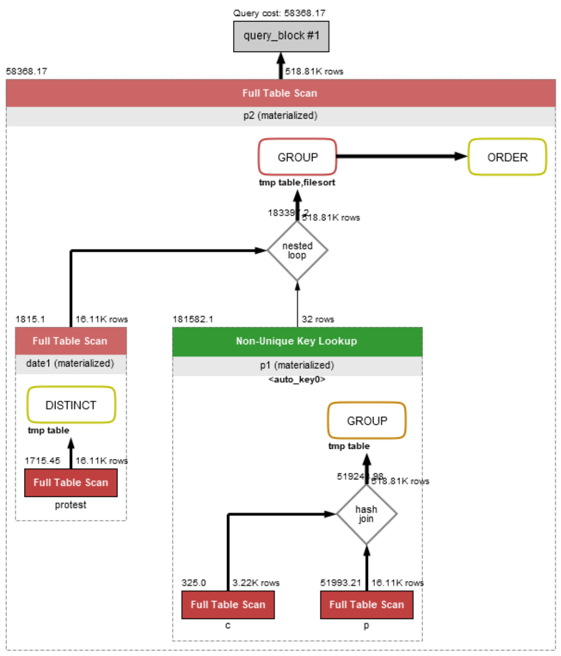
Among all timed queries, the average execution time was reduced by 45%. 11 of the 15 queries benefited from the optimization. In queries 1, 2, 8, 10a, 10b, 10c, 10d, and 11, the reduction ranged from 33.8% to 75.7%. Queries 9a, 9b, and 9c were rewritten to place the aggregation on ‘event_date’ in result p1 before the join with result p2. This decreased the execution time by more than 99% without affecting the final result in terms of the returned tuples. Execution plans for query 9a using the (unoptimized) schema ‘final_project2’ are shown on the next page. Note that the query cost is 94038848066.50 before rewriting and 58368.17 after rewriting. The query cost is further reduced to 210.06 after optimization.

Query 9a Execution

Before Rewriting



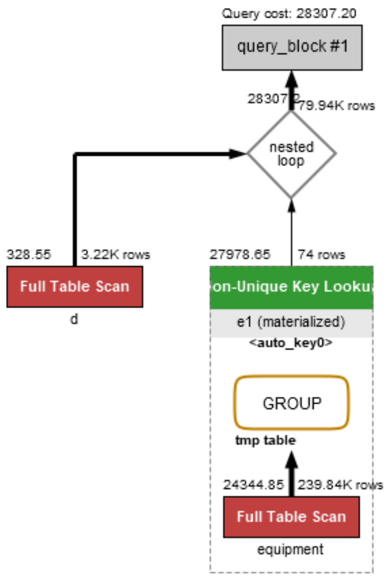
After Rewriting



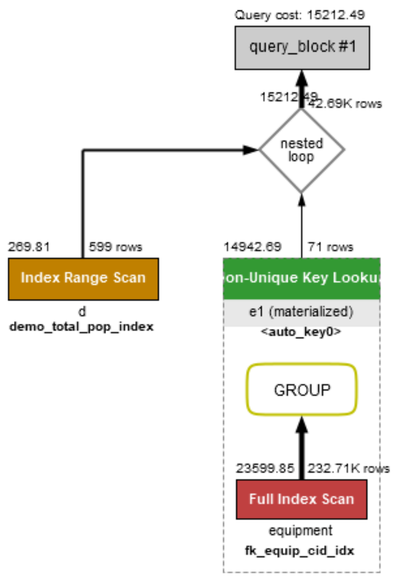
Queries 12c, 12d, 13c, and 13d showed no performance improvement. In each, a full scan on the ~240K row 'equipment' table is required, resulting in a high query cost both pre and post-optimization. The execution plans for query 12c are shown below. The query cost is 28307.20 and 15212.49 before and after optimization, respectively. Indexing has less of an effect on small tables or on large tables where most of the rows need to be processed.

Query 12c Execution

Pre-Optimization



Post-Optimization



Overall, rewriting inefficient queries had the biggest impact on performance, followed by indexing the primary/foreign keys, followed by indexing the referenced columns. In particular, indexing 'county_id' — which is a primary key in 'county' and 'demographic' and a foreign key in 'equipment' and 'protest' — had a large impact since 'county_id' is used extensively in joins. In contrast, indexing the referenced columns had a smaller impact since each index is less frequently used, if at all, in a given query execution plan. In general, indexing can greatly reduce the read cost but may slightly increase the update cost. It also uses a marginal amount of storage on the disk drive. Since the tested database workload consisted entirely of reads, indexing resulted in a significant performance gain in a majority of the timed queries and no performance loss in the worst case.

TECHNICAL CHALLENGES

One difficulty we encountered in data cleaning was very low mapping rates when joining the three tables with county and state name as key. We improved the mapping rates (and therefore usable data) by repetitively examining the county names with the highest counts that are not mapped, and made a series of changes including capitalizing, standardizing the use of punctuation and space, changing Spanish letters to English ones, etc. Mapping rates improved to over 99% from less than 50% after taking all the steps.

Another difficulty came from the implementation of the webapp. None of us have any web development experience, so we learned it as we went. The largest source of difficulty was from the varying efficacy of material online. Because React and Node have been used for so long, there are a lot of deprecated functions that need to be weeded out. On top of this, it was very difficult finding a good chart library and making it work. Many were more suited to static data that is inputted in ahead of time or in a non-React environment. There were very few options that used dynamic data pulled from databases that also clearly expounded on how to implement that functionality.

Another source of difficulty was in the back-end. A lot of time was spent deciding if the Node+Express implementation we decided on was the most appropriate choice. We tried multiple other server implementations, such as those involving GraphQL, but we ultimately decided to stay with Node+Express. This was largely because of our exposure to it earlier in the course. Other implementations required us to learn what seemed like an entirely new framework from the ground up. GraphQL had seemed promising, but we could not get our queries situated within it. Finally, it was an unexpected challenge to link together the front and back-end in addition to the multiple components of each. A good deal of planning went into the structure.