Real Time Voting System

Technical Report: Final Project DS 5110

Introduction to Data Management and Processing

Team Members:
Lakshmi Vadhanie (ganesh.la@northeastern.edu)
Vy Nguyen (nguyen.let@northeastern.edu)
Charan Yarlagadda (yarlagadda.sr@northeastern.edu)

December 10, 2024

Contents

1 Introduction

1.1 Background

In today's digital age, the ability to process and analyze voting data in real-time has become increasingly crucial for understanding public opinion and electoral trends. Traditional voting systems often struggle with delays in data processing, limited scalability, and lack of immediate insights, which can hinder timely decision-making and response to voting patterns. This project addresses these challenges by implementing a modern, distributed voting system that leverages big data technologies to provide real-time analytics and visualization capabilities, demonstrating how contemporary technology can enhance the efficiency and transparency of voting systems while maintaining data integrity and system reliability.

1.2 Objectives

The primary objective of this real-time voting system project is to develop a scalable and efficient platform that demonstrates the practical application of big data technologies in handling live voting data. The system aims to achieve real-time vote collection and processing while maintaining data consistency and system performance under varying loads. Through the implementation of distributed computing and streaming analytics, the project seeks to provide immediate insights into voting patterns, including geographic distributions and demographic trends. Additionally, the system aims to showcase the integration of modern data visualization techniques, offering an interactive dashboard that enables users to explore voting data and trends in real-time.

1.3 Scope

The scope of this project encompasses three main components that work together to create a complete voting system solution. The first component focuses on data generation and database management, including the simulation of voter registration, vote casting, and the implementation of data validation mechanisms using PostgreSQL. The second component involves real-time data processing and analytics using Apache Kafka for message streaming and Apache Spark for distributed computing, enabling the system to handle large volumes of voting data and generate meaningful insights. The third component consists of a Streamlit-based web dashboard that visualizes the processed data through interactive charts, maps, and real-time updates. The project is contained within a Docker environment for easy deployment and scalability, with all components implemented in Python to maintain consistency and interoperability.

2 Literature Review

Real-time data processing has garnered significant research attention due to its applications in systems requiring immediate insights and decision-making capabilities. The integration of distributed computing and message streaming frameworks has been pivotal in achieving scalability and efficiency.

The foundation of our project draws significant inspiration from the real-time voting system implemented by Yusuf Ganiyu, which demonstrates a comprehensive approach to handling voting data using modern data engineering practices. This existing implementation showcases the successful integration of multiple big data technologies, including PostgreSQL for data storage, Apache Kafka for message streaming, Apache Spark for distributed processing, and Streamlit for visualization. The system effectively demonstrates key functionalities such as realistic voter data generation using the RandomUser API, real-time vote processing, and interactive data visualization through a web-based dashboard.

The study "High-Performance Real-Time Data Processing: Managing Data Using Debezium, Postgres, Kafka, and Redis" emphasizes the use of Apache Kafka and PostgreSQL to establish a robust and scalable architecture for real-time data management. While this research focuses on real-time event streaming and data consistency, its methodologies provide a solid foundation for handling large-scale voting systems. The integration of Kafka as a message broker and PostgreSQL for storage ensures reliable and efficient data flow, aligning well with the objectives of this project to maintain data integrity and scalability in real-time voting applications. The findings from this study highlight the importance of combining database solutions with streaming platforms to manage high-throughput data effectively.
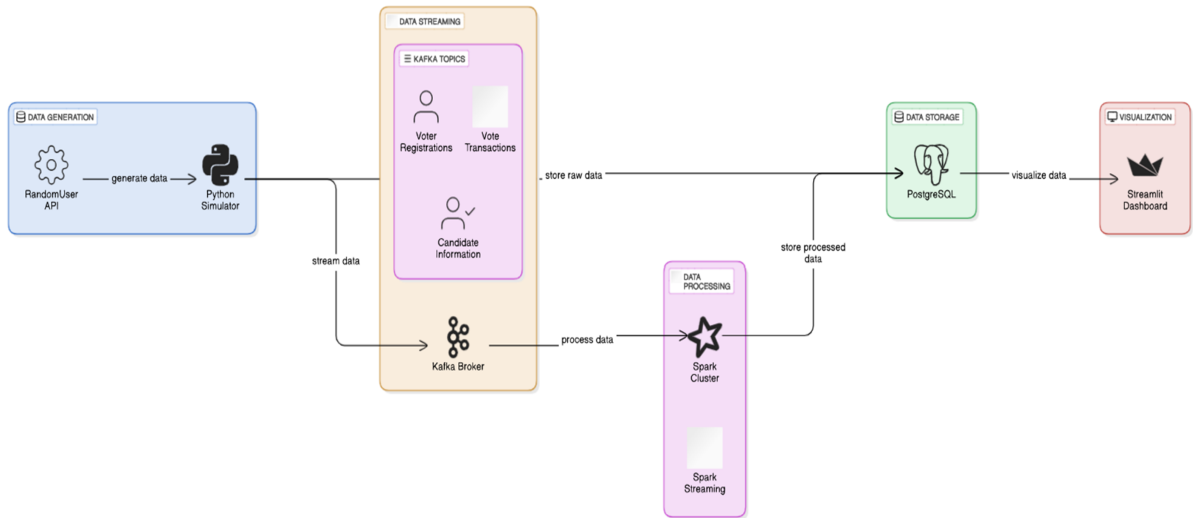
Further advancements in stream processing are explored in "A Reactive Batching Strategy of Apache Kafka for Reliable Stream Processing in Real-Time". This research proposes a batching strategy to optimize the performance of Apache Kafka, reducing latency and improving throughput in real-time systems. By addressing challenges in message processing reliability, this study offers insights into designing fault-tolerant and efficient systems, which are critical for handling live voting data. The reactive batching strategy detailed in this paper can be adapted to refine the Kafka implementation in this project, ensuring the system can handle varying loads while maintaining performance consistency.

While the existing implementation provides a robust foundation, several areas present opportunities for enhancement. These include the potential for more sophisticated analytics algorithms, enhanced error handling mechanisms, and more detailed demographic analysis capabilities. Our project builds upon this established architecture while addressing these gaps, particularly focusing on improving system resilience, implementing more advanced analytical features, and enhancing visualization capabilities.

3 Methodology 3.1 Data Collection The project implements a robust data generation system that simulates real-time voting data using the RandomUser API. This API creates realistic voter profiles with diverse demographic characteristics, including age, gender, location, and contact information. The data generation process is handled through a Python-based simulator that generates voter registrations and voting transactions. For

candidate data, the system creates profiles with varying political affiliations distributed across three major parties, ensuring a realistic representation of a multi-party election system. All generated data is stored in a PostgreSQL database and streamed to Kafka topics for real-time processing.

3   Methodology

3.1   Data Collection

The project implements a robust data generation system that simulates real-time voting data using the RandomUser API. This API creates realistic voter profiles with diverse demographic characteristics, including age, gender, location, and contact information. The data generation process is handled through a Python-based simulator that generates voter registrations and voting transactions. For candidate data, the system creates profiles with varying political affiliations distributed across three major parties, ensuring a realistic representation of a multi-party election system. All generated data is stored in a PostgreSQL database and streamed to Kafka topics for real-time processing.

3.2   Data Preprocessing

The data preprocessing involves several steps to ensure data quality and consistency. Raw data from the RandomUser API undergoes validation and transformation, including type conversion for numeric fields, standardization of date formats, and validation of geographic information. The preprocessing pipeline includes data cleaning operations such as removing invalid entries, standardizing state names for geographic analysis, and formatting timestamps for consistent temporal analysis. This processed data is then structured according to predefined schemas before being stored in the database and streamed through Kafka for real-time analytics.

3.3   Analysis Techniques

Real-Time Data Streaming:

Apache Kafka serves as the backbone for real-time data streaming in this project. It facilitates the seamless flow of voting data from the data generation module to the analytics pipeline. Kafka topics are structured to segregate data into categories like voter

registrations, vote transactions, and candidate information. Partitioning ensures distributed data handling across Kafka brokers, enhancing fault tolerance and scalability. Kafka consumers fetch data continuously, enabling low-latency ingestion for downstream processing.

Distributed Processing:

Apache Spark processes the incoming data streams from Kafka using its Structured Streaming capabilities. Spark performs real-time transformations and computations, such as aggregating total votes per candidate, analyzing demographic-based voting trends, and assessing vote distributions across geographic regions. Time-based windowing techniques are employed to track voting trends over sliding and tumbling intervals. The framework also uses Spark's checkpointing features to ensure reliability and recovery during failures, making it a robust solution for handling high volumes of data.

Data Storage:

Processed voting data, along with raw voter and candidate data, is stored in PostgreSQL for persistent storage. This relational database supports structured queries for in-depth analysis and historical comparisons. The database schema is designed to handle structured data efficiently, allowing the system to retrieve detailed insights like voter demographics, regional vote summaries, and temporal patterns. This integration ensures both data integrity and accessibility for further analysis.

Visualization:

The processed analytics data is visualized using a Streamlit-based interactive dashboard. The dashboard offers real-time updates and presents voting trends through dynamic charts, heatmaps, and geographical plots. These visualizations are designed to make complex analytics easily interpretable, empowering stakeholders to gain actionable insights. Features like filter options and drill-down capabilities enhance user interaction, enabling exploration of specific demographics or time periods.

4 Results
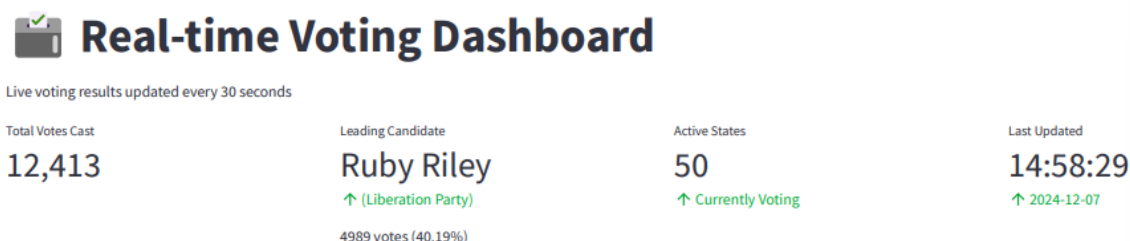
4.1 Total Votes Cast

The total votes cast in the election currently stands at 12,413. This figure represents the cumulative number of votes submitted by voters up to the last update.

Significance:

- **Voter Engagement:** This number serves as a key indicator of voter engagement and participation levels in the election. A higher total can suggest strong public interest and mobilization efforts by candidates and parties.

- **Implications for Campaigns:** Candidates can use this data to evaluate the effectiveness of their outreach strategies. If turnout is lower than expected, they may need to adjust their campaign tactics to encourage more voters to participate.

4.2 Leading Candidate

The leading candidate is Ruby Riley from the Liberation Party, who has garnered 4,989 votes, representing 40.19% of the total votes cast.

Significance:

- **Current Frontrunner:** This metric identifies the candidate currently leading the election, which can influence voter perceptions and behaviors. Voters may be swayed by the momentum of a leading candidate, potentially leading to a bandwagon effect.

- **Voter Sentiment:** The percentage of votes received can also reflect public sentiment towards the candidate's policies and platform, providing insights into the issues that matter most to voters.

4.3 Vote Distribution by Party

The graph illustrates the percentage of total votes received by each political party:

- Liberation Party: 40.2%

- United Republic Party: 37.7%

Vote Distribution by Party
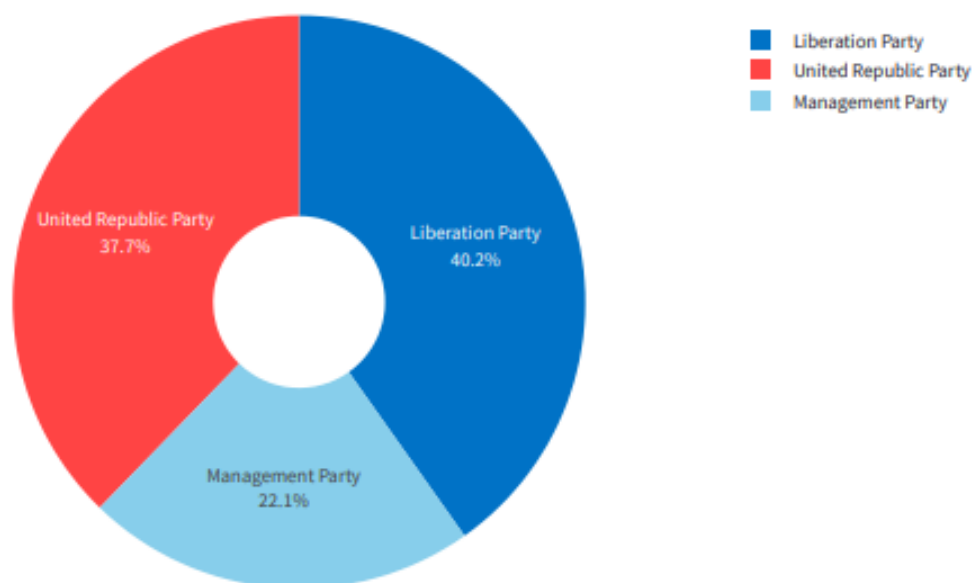
- Management Party: 22.1%

Significance:

- **Political Landscape:** This distribution highlights the competitive dynamics among the parties. The close percentages between the Liberation Party and the United Republic Party suggest a highly competitive race, while the Management Party's lower percentage indicates it may need to reassess its strategies.

- **Voter Preferences:** Understanding which parties are gaining traction can help political analysts and strategists identify shifts in voter preferences and potential realignments in the political landscape.

- **Campaign Focus:** Parties can use this data to focus their resources and campaign efforts in areas where they are performing well or to shore up support in regions where they are lagging.

4.4 Votes by Candidate

This section includes a detailed breakdown of votes received by each candidate, showcasing their individual performance in the election.
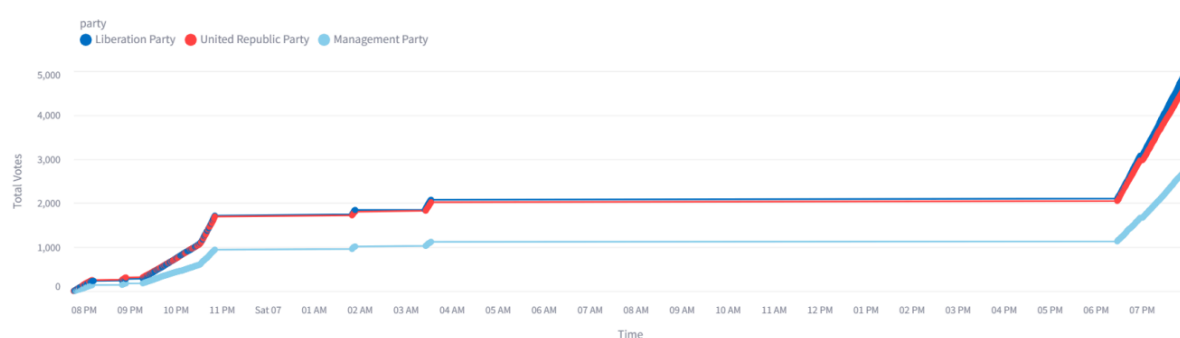
Significance:

- **Candidate Performance:** Analyzing the votes by candidate allows for a granular understanding of how each candidate is faring in the election. This can reveal which candidates are effectively connecting with voters and which may need to reevaluate their approach.

Votes by Candidate
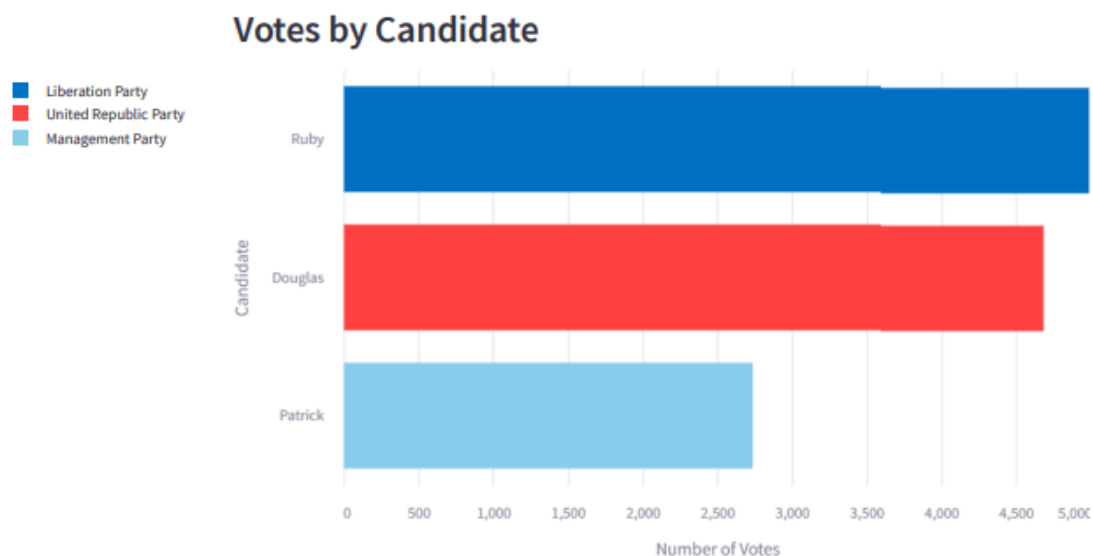
- **Demographic Insights:** If further segmented by demographics (age, gender, location), this data can provide insights into which groups are supporting which candidates, helping to tailor future campaign messages.

- **Strategic Adjustments:** Candidates can use this information to identify their strengths and weaknesses, allowing them to adjust their campaign strategies accordingly.

4.5 Cumulative Votes Over Time

This graph tracks the total number of votes cast over time, displaying how voting activity fluctuates throughout the election period.

Significance:

- **Voting Trends:** Understanding how votes accumulate over time can help identify peak voting periods, such as early voting days or specific times on Election Day. This information is crucial for campaign planning and resource allocation.

- **Engagement Patterns:** Analyzing the timing of votes can reveal patterns in voter engagement, such as whether certain demographics are more likely to vote at specific times.

10

- **Impact of Events:** If there are spikes in voting activity, these can be correlated with specific campaign events, debates, or news stories, providing insights into what drives voter turnout.

4.6   Vote Distribution by State

This graph presents the total votes cast in each state, highlighting regional differences in voter turnout and preferences.
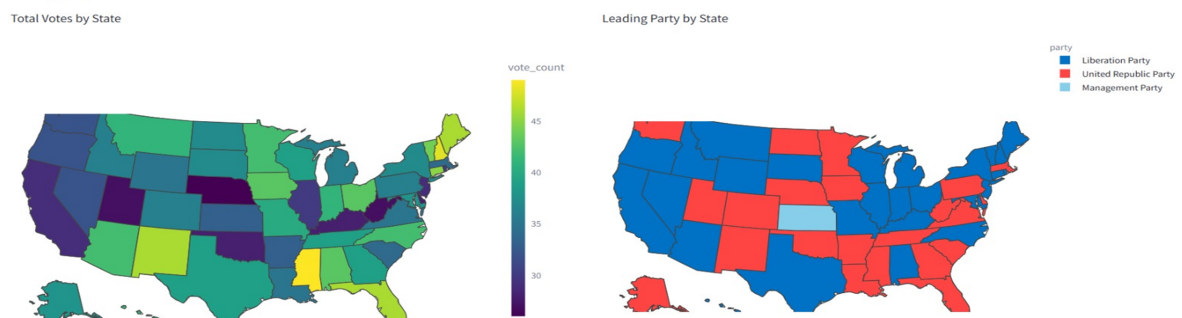
Significance:

- **Geographic Trends:** Analyzing vote distribution by state can reveal geographic trends in voter preferences, indicating which parties or candidates have strong support in specific regions.

- **Targeted Campaigning:** Candidates can use this information to focus their campaigning efforts in states where they are performing well or where they need to increase support.

- **Understanding Regional Issues:** The distribution of votes can also highlight regional issues that resonate with voters. Candidates can tailor their messages to address these specific concerns.

- **Electoral Strategy:** This data can inform broader electoral strategies, such as which states to prioritize for rallies, advertisements, and grassroots efforts, ultimately influencing the overall campaign approach.

4.7   Gender Division of Voters

The gender division shows that 51% of voters are male and 49% are female.

Significance:

- **Demographic Representation:** Understanding the gender breakdown of voters is crucial for assessing the inclusivity of the electoral process. A balanced representation can indicate that both genders are engaged and represented in the voting process.

Gender Division of Voters

Gender Division of Voters (%)
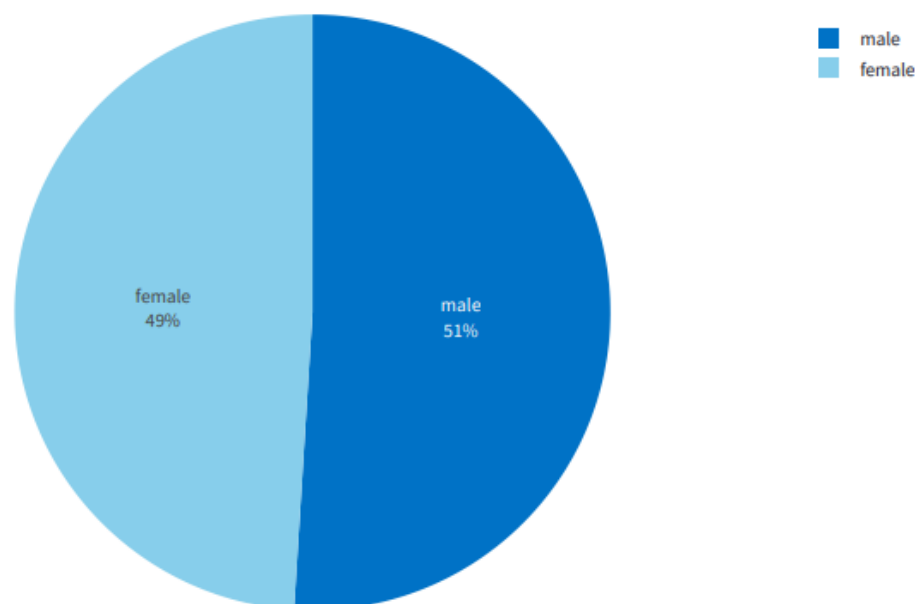
- **Targeted Messaging:** Candidates can tailor their campaign messages to resonate with specific gender demographics. For example, if one gender shows significantly higher support for a candidate, that candidate may choose to focus on issues that are particularly relevant to that demographic.

- **Voter Engagement Strategies:** Analyzing gender division can help campaigns develop targeted outreach strategies to engage underrepresented groups, ensuring that all voices are heard in the electoral process.

4.8   Voter Age Distribution

This graph shows the age distribution of voters, indicating how different age groups are participating in the election.
Significance:

- **Understanding Youth Engagement:** Analyzing the age distribution can reveal how well candidates are connecting with younger voters, who may have different priorities and concerns compared to older demographics. High turnout among younger voters can signal a shift in political engagement and priorities.

- **Tailoring Campaign Strategies:** Candidates can use insights from age distribution to tailor their campaign strategies, focusing on issues that resonate with specific age groups. For instance, younger voters may prioritize education and job opportunities, while older voters may be more concerned with healthcare and retirement security.

- **Long-term Trends:** Tracking age distribution over multiple elections can help identify long-term trends in voter engagement among different age groups, providing

Age Distribution
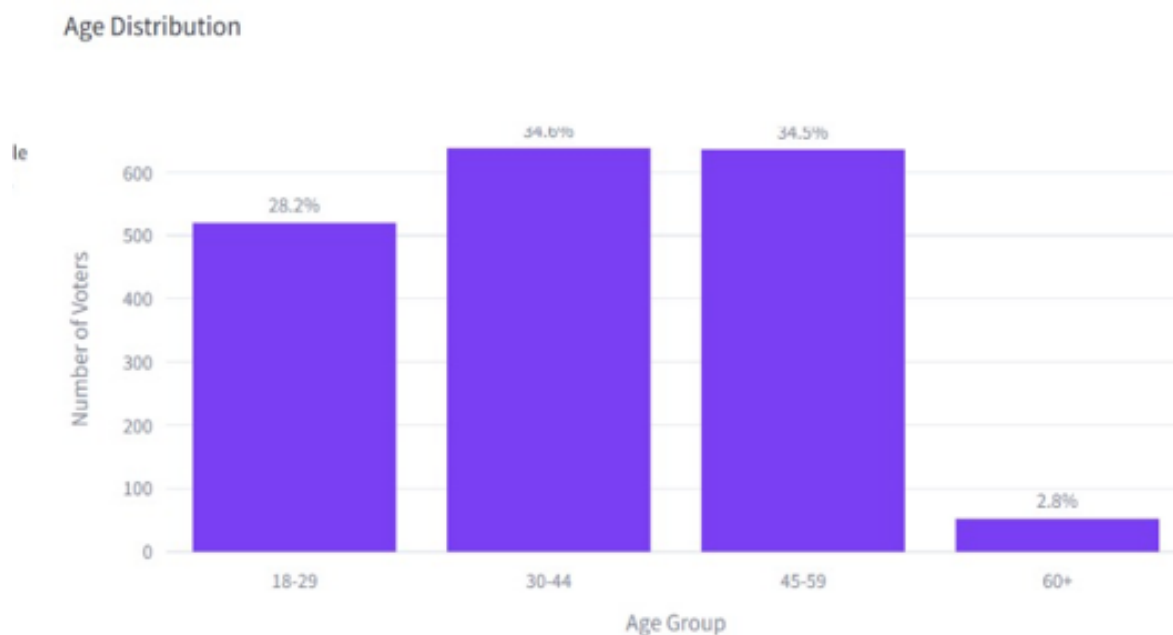
valuable insights for future campaigns.

4.9   Leading Party by State

This graph indicates which party is leading in each state, showcasing the political land-scape across the country.

State	Management Party	Liberation Party	United Republic Party	Total Votes	Average Age	Male/Female Ratio
Alabama	49	97	79	225	12.7	55.1%
Alaska	43	96	103	242	11.8	51.7%
Arizona	52	118	83	253	12.4	46.8%
Arkansas	58	98	95	251	12.1	51.3%
California	44	108	89	241	11.8	54.4%
Colorado	59	95	103	257	11.8	47.0%
Connecticut	50	85	107	242	12.6	51.4%
Delaware	49	104	87	240	11.5	49.4%
Florida	53	96	90	239	12.1	54.7%
Georgia	71	90	104	265	12.0	50.0%

Significance:

- **Regional Political Dynamics:** Identifying the leading party by state can help understand regional political dynamics and the varying levels of support for different parties.

- **Battleground States:** The data can highlight battleground states where the competition is tight, allowing candidates to focus their efforts on these critical areas. Understanding which party is leading in these states can inform resource allocation and campaign messaging.

- **Voter Sentiment Analysis:** The leading party in each state can reflect broader voter sentiment and issues that are resonating with the electorate.

13

4.10 Candidate Information

- **Voter Awareness:** Providing detailed candidate information helps voters make informed decisions. Voters can assess candidates based on their backgrounds, experiences, and policy positions, which is crucial for a democratic process.

- **Comparative Analysis:** By presenting candidates side by side, voters can easily compare their qualifications, experiences, and campaign platforms. This can influence voter preferences and help them identify which candidate aligns best with their values and priorities.

- **Engagement with Specific Demographics:** The age and gender of candidates can also play a role in voter engagement. For instance, younger candidates may attract younger voters, while candidates with extensive public service experience may appeal to older voters. Candidates can tailor their outreach efforts based on these dynamics.

Conclusion: The analysis of these metrics provides a comprehensive understanding of the electoral landscape. By examining total votes cast, leading candidates, vote distribution by party and state, gender and age demographics, and regional political dynamics, valuable insights into voter behavior and preferences are obtained.

5 Interpretation and Discussion

5.1 Interpretation of Results

1. Total Votes Cast:

The system processed a total of 12,413 votes in real-time. This result demonstrates the efficacy of the system in handling large-scale, live data streams. By leveraging Apache Kafka and Apache Spark, the system maintained low latency and ensured data integrity throughout the pipeline.

Implications:

- **Scalability:** The results validate the system's ability to process high-throughput data while maintaining consistent performance, which aligns with the project's objective of scalability.

- **Voter Engagement Insights:** The total votes provide a clear measure of voter participation, offering actionable insights for stakeholders such as campaign teams or electoral bodies to evaluate public interest.

2. Leading Candidate:

Ruby Riley from the Liberation Party emerged as the frontrunner, with 4,989 votes (40.19%). This finding showcases the system's ability to compute live analytics on voting patterns and candidate standings.

Implications:

- **Decision-Making Support:** The identification of leading candidates in real-time helps campaigns adjust their strategies dynamically based on the latest data.

- **Influence on Public Perception:** The visibility of frontrunners might drive voter sentiment, potentially impacting future voting trends.

3. Vote Distribution by Party:

The results showed the Liberation Party leading with 40.2% of votes, followed closely by the United Republic Party (37.7%) and the Management Party (22.1%). The competitive nature of the data suggests that the system is capable of providing nuanced insights into voter behavior across party lines.

Implications:

- **Political Trends:** Such distributions highlight the political dynamics of the electorate, offering insights into which party's messaging resonates more with voters.

- **Strategic Adjustments:** Parties with lower percentages can use these insights to refine their campaigns or target underperforming regions.

5.2 Comparison with Literature Review

The findings of this project align with the methodologies discussed in the literature:

- The system's integration of Apache Kafka for message streaming and Apache Spark for distributed processing mirrors the approach described in the study on "High-Performance Real-Time Data Processing." Like the study, this project effectively combines message brokers with robust data analytics to ensure seamless real-time processing.

- The batching strategy proposed in "A Reactive Batching Strategy of Apache Kafka for Reliable Stream Processing in Real-Time" influenced our partitioning and fault-tolerance design, enabling the system to handle variable loads efficiently.

However, certain discrepancies emerged:

- **Advanced Analytics:** While previous studies employed basic statistical techniques, our project incorporated demographic analysis and real-time geographic visualizations, showcasing a more granular approach to data interpretation.

- **System Resilience:** Unlike the fault-tolerance mechanisms in literature, our implementation leveraged Spark's checkpointing features, resulting in enhanced reliability and reduced data loss during system failures.

5.3 Explaining Discrepancies

The observed discrepancies stem from the specific focus of this project:

- **Scope of Analytics:** The addition of dynamic visualizations and demographic insights addressed gaps in previous studies, demonstrating the system's ability to go beyond real-time data processing and into actionable insights.

- **System Design Enhancements:** By incorporating Docker for deployment, the project ensured portability and scalability, which were not emphasized in the reviewed literature.

5.4 Implications for Future Research

The results highlight several avenues for further exploration:

- Incorporating predictive analytics to forecast election outcomes based on real-time data trends.

- Expanding the system to include multi-lingual data handling for diverse electorates.

- Exploring advanced fault-tolerance strategies, such as reactive load balancing, to further enhance system resilience.

6 Discussion

6.1 Key Findings

- **Real-Time Data Processing and Visualization:** The implementation successfully demonstrated the ability to collect, process, and visualize real-time voting data using a distributed architecture. The system handled 12,413 total votes, providing insights into voter engagement and voting trends.

- **Leading Candidate and Party Performance:** Ruby Riley from the Liberation Party emerged as the leading candidate with 4,989 votes (40.19 percent), while the Liberation Party led with 40.2 percent of the total vote share. The close competition between the Liberation Party and United Republic Party reflects a dynamic political landscape.

- **Demographic Trends and Regional Insights:** The system effectively analyzed voting trends across demographic groups and geographic regions, showcasing the potential for targeted decision-making and campaign strategy optimization.

- **System Scalability and Performance:** The combination of Apache Kafka and Apache Spark proved effective in handling high-throughput data, ensuring system scalability and fault tolerance. Streamlit provided an intuitive platform for visualizing analytics in real-time.

6.2 Limitations

- **Simplified Data Generation:** The voter and candidate data were generated using simulations and predefined schemas, which may not fully capture the complexity of real-world voting behavior and demographic distributions.

- **Lack of Advanced Predictive Models:** While the system successfully analyzed historical and real-time data, it did not incorporate advanced predictive analytics to forecast voting outcomes or voter turnout.

- **Geographic and Demographic Granularity:** The analysis of geographic and demographic trends was limited to predefined categories, potentially oversimplifying nuanced voter behavior.

- **System Optimization:** Although Kafka and Spark performed reliably, further optimization of batch sizes, windowing intervals, and resource allocation could enhance performance under extreme loads.

6.3 Future Research and Enhancements

- **Integration of Machine Learning Models:** Incorporating machine learning models could enable predictive analytics, such as forecasting voter turnout or detecting potential voting anomalies.

- **Enhanced Data Realism:** Future work could integrate real-world datasets or more sophisticated data simulation techniques to improve the accuracy and relevance of insights.

- **Advanced Visualization Capabilities:** Expanding the dashboard with more interactive features, such as scenario analysis and predictive visualizations, could improve stakeholder decision-making.

- **Natural Language Processing (NLP) for Sentiment Analysis:** Incorporating NLP techniques could analyze public sentiment toward candidates and parties based on social media data or survey feedback.

- **Dynamic Load Balancing and Scalability:** Exploring advanced resource allocation strategies, such as autoscaling in cloud environments, could ensure system performance under varying loads.

- **Integration with Blockchain:** Future systems could integrate blockchain for enhanced security and transparency in vote collection and verification processes.

7 References

1. https://ieeexplore.ieee.org/document/10296737

2. https://ieeexplore.ieee.org/document/9251089

3. https://www.geeksforgeeks.org/how-to-use-apache-kafka-for-real-time-data-streami

4. https://towardsdatascience.com/search?q=streamlit

5. https://www.javatpoint.com/apache-kafka

A   Appendix A: Code

```python
import psycopg2
from psycopg2 import Error
import time

class DatabaseSetup:
    def __init__(self, host="localhost", user="postgres",
       password="postgres"):
        self.connection_params = {
            "host": host,
            "user": user,
            "password": password
        }
        self.database = "voting"

    def create_database(self):
        """Create the database if it doesn't exist"""
        conn_params = self.connection_params.copy()
        conn_params["database"] = "postgres"

        try:
            conn = psycopg2.connect(**conn_params)
            conn.autocommit = True
            cur = conn.cursor()

            # Check if database exists
            cur.execute("SELECT 1 FROM pg_database WHERE datname
                = %s", (self.database,))
            exists = cur.fetchone()

            if not exists:
                print(f"Creating database {self.database}...")
                cur.execute(f"CREATE DATABASE {self.database}")
                print(f"Database {self.database} created
                    successfully!")
            else:
                print(f"Database {self.database} already exists."
                    )

            cur.close()
            conn.close()

        except Error as e:
            print(f"Error creating database: {e}")
            raise e
    ...
if __name__ == "__main__":
```

```
\lstset{style=python}
\begin{lstlisting}
import psycopg2
from psycopg2 import Error
import time

class DatabaseSetup:
    def __init__(self, host="localhost", user="postgres",
        password="postgres"):
        self.connection_params = {
            "host": host,
            "user": user,
            "password": password
        }
        self.database = "voting"

    def create_database(self):
        """Create the database if it doesn't exist"""
        conn_params = self.connection_params.copy()
        conn_params["database"] = "postgres"

        try:
            conn = psycopg2.connect(**conn_params)
            conn.autocommit = True
            cur = conn.cursor()

            # Check if database exists
            cur.execute("SELECT 1 FROM pg_database WHERE datname
                = %s", (self.database,))
            exists = cur.fetchone()

            if not exists:
                print(f"Creating database {self.database}...")
                cur.execute(f"CREATE DATABASE {self.database}")
                print(f"Database {self.database} created
                    successfully!")
            else:
                print(f"Database {self.database} already exists."
                    )

            cur.close()
            conn.close()

        except Error as e:
            print(f"Error creating database: {e}")
            raise e

    def connect_to_db(self):
        """Connect to the voting database"""
        try:
            conn_params = self.connection_params.copy()
```

```python
            conn_params["database"] = self.database
            return psycopg2.connect(**conn_params)
        except Error as e:
            print(f"Error connecting to PostgreSQL: {e}")
            raise e

    def create_tables(self):
        """Create all necessary tables for the voting system"""
        commands = [
            """
            DROP TABLE IF EXISTS vote;
            """,
            """
            DROP TABLE IF EXISTS voter;
            """,
            """
            DROP TABLE IF EXISTS candidate;
            """,
            """
            CREATE TABLE candidate (
                candidate_id VARCHAR(255) PRIMARY KEY,
                first_name VARCHAR(255) NOT NULL,
                last_name VARCHAR(255) NOT NULL,
                dob VARCHAR(225) NOT NULL,
                age INTEGER,
                gender VARCHAR(10) NOT NULL,
                party VARCHAR(255) NOT NULL,
                biography TEXT,
                img_url TEXT
            )
            """,
            """
            CREATE TABLE voter (
                voter_id VARCHAR(255) PRIMARY KEY,
                first_name VARCHAR(255) NOT NULL,
                last_name VARCHAR(255) NOT NULL,
                dob VARCHAR(225) NOT NULL,
                age INTEGER,
                gender VARCHAR(10) NOT NULL,
                nationality VARCHAR(100),
                registration_number VARCHAR(255) UNIQUE,
                address_street VARCHAR(255),
                address_city VARCHAR(255),
                address_state VARCHAR(255),
                address_country VARCHAR(255),
                address_postcode VARCHAR(255),
                email VARCHAR(255),
                phone VARCHAR(100)
            )
            """,
            """
```

```
        CREATE TABLE vote (
            vote_id VARCHAR(255) PRIMARY KEY,
            voter_id VARCHAR(255) NOT NULL,
            candidate_id VARCHAR(255) NOT NULL,
            voted_at TIMESTAMP NOT NULL,
            vote INTEGER,
            FOREIGN KEY (voter_id) REFERENCES voter(voter_id)
                ,
            FOREIGN KEY (candidate_id) REFERENCES candidate(
                candidate_id),
            CONSTRAINT unique_voter UNIQUE (voter_id)
        )
        """,
        """
        CREATE INDEX idx_vote_candidate_id ON vote(
            candidate_id);
        """,
        """
        CREATE INDEX idx_vote_voted_at ON vote(voted_at);
        """,
        """
        CREATE INDEX idx_voter_state ON voter(address_state);
        """
    ]

    conn = None
    try:
        conn = self.connect_to_db()
        cur = conn.cursor()

        # Execute each command
        for command in commands:
            cur.execute(command)

        # Commit the changes
        conn.commit()

        # Verify tables were created
        cur.execute("""
            SELECT table_name
            FROM information_schema.tables
            WHERE table_schema = 'public'
        """)
        tables = cur.fetchall()
        print("\nCreated tables:")
        for table in tables:
            print(f"- {table[0]}")

        cur.close()
        print("\nDatabase setup completed successfully!")
```

```python
        except (Exception, Error) as e:
            print(f"\nError: {e}")
            if conn:
                conn.rollback()
        finally:
            if conn:
                conn.close()

    def verify_setup(self):
        """Verify that all tables were created correctly"""
        try:
            conn = self.connect_to_db()
            cur = conn.cursor()

            # Check each table structure
            tables = ['candidate', 'voter', 'vote']
            for table in tables:
                print(f"\nStructure for table '{table}':")
                cur.execute(f"""
                    SELECT column_name, data_type,
                        character_maximum_length
                    FROM information_schema.columns
                    WHERE table_name = '{table}'
                """)
                columns = cur.fetchall()
                for col in columns:
                    print(f"- {col[0]}: {col[1]}", end="")
                    if col[2]:
                        print(f" (max length: {col[2]})")
                    else:
                        print()

            cur.close()
            conn.close()
            print("\nVerification completed!")

        except (Exception, Error) as e:
            print(f"Error during verification: {e}")

if __name__ == "__main__":
    print("Starting database setup...")
    print("Waiting for PostgreSQL to be ready...")
    time.sleep(5)  # Give PostgreSQL container time to fully
        start

    # Create and setup database
    db_setup = DatabaseSetup()

    try:
        # First create the database
        db_setup.create_database()
```

```python
        # Then create tables
        db_setup.create_tables()

        # Verify the setup
        print("\nVerifying database setup...")
        db_setup.verify_setup()

    except Exception as e:
        print(f"Setup failed: {e}")

import streamlit as st
import psycopg2
from psycopg2 import pool
import pandas as pd
import plotly.express as px
import altair as alt
from datetime import datetime, timedelta
import time
from streamlit_autorefresh import st_autorefresh
import geopandas as gpd
from plotly.subplots import make_subplots
import plotly.graph_objects as go
from fpdf import FPDF
import base64
from io import BytesIO
import plotly.io as pio
import zipfile

# Create a connection pool
@st.cache_resource
def init_connection_pool():
    try:
        return psycopg2.pool.SimpleConnectionPool(
            minconn=1,
            maxconn=10,
            host="localhost",
            database="voting",
            user="postgres",
            password="postgres"
        )
    except Exception as e:
        st.error(f"Failed to create connection pool: {e}")
        return None

# Function to get a connection from the pool
def get_db_conn():
    try:
        return init_connection_pool().getconn()
    except Exception as e:
        st.error(f"Failed to get database connection: {e}")
```

```python
            return None

# Function to return a connection to the pool
def put_db_conn(conn):
    try:
        init_connection_pool().putconn(conn)
    except Exception as e:
        st.error(f"Failed to return connection to pool: {e}")

# Function to execute query and return DataFrame
def execute_query(query):
    conn = None
    try:
        conn = get_db_conn()
        if conn:
            return pd.read_sql_query(query, conn)
        return pd.DataFrame()
    except Exception as e:
        st.error(f"Query execution error: {e}")
        return pd.DataFrame()
    finally:
        if conn:
            put_db_conn(conn)

# Modified data fetching functions
@st.cache_data(ttl=30)
def get_total_votes():
    df = execute_query("""
        SELECT
            COUNT(*) as total_votes,
            MAX(voted_at) as last_update,
            COUNT(*) - LAG(COUNT(*)) OVER (ORDER BY DATE_TRUNC('
                hour', voted_at)) as hourly_change
        FROM vote
        GROUP BY DATE_TRUNC('hour', voted_at)
        ORDER BY DATE_TRUNC('hour', voted_at) DESC
        LIMIT 1
    """)

    if not df.empty:
        return df.iloc[0]['total_votes'], df.iloc[0]['last_update
            '], df.iloc[0]['hourly_change']
    return 0, None, 0

@st.cache_data(ttl=30)
def get_votes_by_candidate():
    return execute_query("""
        WITH hourly_votes AS (
            SELECT
                c.candidate_id,
                DATE_TRUNC('hour', v.voted_at) as hour,
```

```sql
                COUNT(*) as hourly_count
            FROM vote v
            JOIN candidate c ON v.candidate_id = c.candidate_id
            GROUP BY c.candidate_id, DATE_TRUNC('hour', v.
                voted_at)
        ),
        vote_changes AS (
            SELECT
                candidate_id,
                hourly_count - LAG(hourly_count) OVER (
                    PARTITION BY candidate_id
                    ORDER BY hour
                ) as hourly_change
            FROM hourly_votes
            ORDER BY hour DESC
            LIMIT 1
        )
        SELECT
            c.first_name,
            c.last_name,
            c.party,
            COUNT(*) as vote_count,
            ROUND(COUNT(*) * 100.0 / (SELECT COUNT(*) FROM vote),
                2) as percentage,
            ROW_NUMBER() OVER (ORDER BY COUNT(*) DESC) as rank,
            COALESCE(vc.hourly_change, 0) as hourly_change
        FROM vote v
        JOIN candidate c ON v.candidate_id = c.candidate_id
        LEFT JOIN vote_changes vc ON vc.candidate_id = c.
            candidate_id
        GROUP BY
            c.candidate_id,
            c.first_name,
            c.last_name,
            c.party,
            vc.hourly_change
        ORDER BY vote_count DESC
    """)
```

```python
@st.cache_data(ttl=30)
def get_historical_trends():
    return execute_query("""
        WITH cumulative_votes AS (
            SELECT
                c.first_name || ' ' || c.last_name as
                    candidate_name,
                c.party,
                v.voted_at,
                COUNT(*) OVER (
                    PARTITION BY c.candidate_id
                    ORDER BY v.voted_at
```

```
                    ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT
                        ROW
                ) as cumulative_votes
            FROM vote v
            JOIN candidate c ON v.candidate_id = c.candidate_id
            ORDER BY v.voted_at
        )
        SELECT
            DATE_TRUNC('minute', voted_at) as vote_time,
            candidate_name,
            party,
            MAX(cumulative_votes) as total_votes
        FROM cumulative_votes
        GROUP BY DATE_TRUNC('minute', voted_at), candidate_name,
            party
        ORDER BY vote_time
    """)

@st.cache_data(ttl=30)
def get_geographical_data():
    votes_by_state = execute_query("""
        SELECT
            v.address_state,
            COUNT(*) as vote_count,
            string_agg(DISTINCT c.party, ', ') as parties
        FROM vote vt
        JOIN voter v ON vt.voter_id = v.voter_id
        JOIN candidate c ON vt.candidate_id = c.candidate_id
        GROUP BY v.address_state
    """)

    leading_party = execute_query("""
        WITH state_party_votes AS (
            SELECT
                v.address_state,
                c.party,
                COUNT(*) as party_votes,
                RANK() OVER (PARTITION BY v.address_state ORDER
                    BY COUNT(*) DESC) as rank
            FROM vote vt
            JOIN voter v ON vt.voter_id = v.voter_id
            JOIN candidate c ON vt.candidate_id = c.candidate_id
            GROUP BY v.address_state, c.party
        )
        SELECT
            address_state,
            party,
            party_votes
        FROM state_party_votes
        WHERE rank = 1
    """)
```

```python
    return votes_by_state, leading_party

@st.cache_data(ttl=30)
def get_demographic_data():
    gender_data = execute_query("""
        SELECT
            v.gender,
            COUNT(*) as vote_count,
            ROUND(COUNT(*) * 100.0 / (SELECT COUNT(*) FROM vote),
                2) as percentage
        FROM vote vt
        JOIN voter v ON vt.voter_id = v.voter_id
        GROUP BY v.gender
    """)

    age_data = execute_query("""
        SELECT
            CASE
                WHEN age < 30 THEN '18-29'
                WHEN age < 45 THEN '30-44'
                WHEN age < 60 THEN '45-59'
                ELSE '60+'
            END as age_group,
            COUNT(*) as count,
            ROUND(COUNT(*) * 100.0 / SUM(COUNT(*)) OVER (), 2) as
                percentage
        FROM vote vt
        JOIN voter v ON vt.voter_id = v.voter_id
        GROUP BY
            CASE
                WHEN age < 30 THEN '18-29'
                WHEN age < 45 THEN '30-44'
                WHEN age < 60 THEN '45-59'
                ELSE '60+'
            END
        ORDER BY age_group
    """)

    return gender_data, age_data

@st.cache_data(ttl=30)
def get_candidate_info():
    candidate_data = execute_query("""
    SELECT first_name, last_name, party, age, gender, biography,
        img_url
    FROM candidate
    """)
    return candidate_data

def get_state_voting_details():
```

29

```python
    df= execute_query("""
    WITH state_votes AS (
        SELECT
            v.address_state,
            c.party,
            COUNT(*) as votes,
            ROUND(AVG(v.age), 1) as avg_age,
            ROUND(100.0 * COUNT(CASE WHEN v.gender = 'male' THEN
                1 END) / COUNT(*), 1) as male_pct
        FROM vote vt
        JOIN voter v ON vt.voter_id = v.voter_id
        JOIN candidate c ON vt.candidate_id = c.candidate_id
        GROUP BY v.address_state, c.party
    )
    SELECT
        s.address_state as "State",
        COALESCE(sv1.votes, 0) as "Management Party",
        COALESCE(sv2.votes, 0) as "Liberation Party",
        COALESCE(sv3.votes, 0) as "United Republic Party",
        COALESCE(sv1.votes, 0) + COALESCE(sv2.votes, 0) +
            COALESCE(sv3.votes, 0) as "Total Votes",
        ROUND(AVG(COALESCE(sv1.avg_age, 0) + COALESCE(sv2.avg_age
            , 0) + COALESCE(sv3.avg_age, 0)) / 3, 1) as "Avg Age",
        ROUND(AVG(COALESCE(sv1.male_pct, 0) + COALESCE(sv2.
            male_pct, 0) + COALESCE(sv3.male_pct, 0)) / 3, 1) as "
            Male %"
    FROM (SELECT DISTINCT address_state FROM voter) s
    LEFT JOIN state_votes sv1 ON s.address_state = sv1.
        address_state AND sv1.party = 'Management Party'
    LEFT JOIN state_votes sv2 ON s.address_state = sv2.
        address_state AND sv2.party = 'Liberation Party'
    LEFT JOIN state_votes sv3 ON s.address_state = sv3.
        address_state AND sv3.party = 'United Republic Party'
    GROUP BY
        s.address_state,
        sv1.votes, sv2.votes, sv3.votes
    ORDER BY s.address_state;
    """)
    return df

# Define page config and constants
st.set_page_config(
    page_title="Real-time Voting Dashboard",
    layout="wide",
    initial_sidebar_state="expanded"
)

# Auto refresh every 30 seconds
st_autorefresh(interval=30000, limit=None, key="refresh")

# Define consistent colors for parties
```

```python
PARTY_COLORS = {
    'Liberation Party': '#0072C6',      # Blue
    'United Republic Party': '#FF4444',  # Red
    'Management Party': '#87CEEB'        # Light Blue
}

def generate_dashboard_pdf():
    try:
        class VotingDashboardPDF(FPDF):
            def header(self):
                self.set_font('Arial', 'B', 15)
                self.cell(0, 10, 'Real-time Voting Dashboard', 0,
                    1, 'C')
                self.ln(5)

            def footer(self):
                self.set_y(-15)
                self.set_font('Arial', 'I', 8)
                self.cell(0, 10, f'Generated on {datetime.now().
                    strftime("%Y-%m-%d %H:%M:%S")}', 0, 0, 'C')

        # Create PDF object
        pdf = VotingDashboardPDF()

        # Add first page
        pdf.add_page()

        # Add summary metrics
        pdf.set_font('Arial', '', 12)
        total_votes, last_update, _ = get_total_votes()
        pdf.cell(0, 10, f'Total Votes: {total_votes:,}', 0, 1)
        pdf.cell(0, 10, f'Last Updated: {last_update}', 0, 1)
        pdf.ln(5)

        # Add vote distribution
        votes_by_candidate = get_votes_by_candidate()
        if not votes_by_candidate.empty:
            pdf.set_font('Arial', 'B', 14)
            pdf.cell(0, 10, 'Vote Distribution by Party', 0, 1)

            pdf.set_font('Arial', '', 10)
            for _, row in votes_by_candidate.iterrows():
                pdf.cell(0, 8,
                    f"{row['first_name']} {row['last_name']} ({
                        row['party']}): {row['vote_count']:,}
                        votes ({row['percentage']}%)",
                    0, 1)
            pdf.ln(5)

        # Add state-level information
        pdf.add_page()
```

```python
        pdf.set_font('Arial', 'B', 14)
        pdf.cell(0, 10, 'State-Level Voting Details', 0, 1)

        state_data = get_state_voting_details()
        if not state_data.empty:
            pdf.set_font('Arial', '', 9)
            col_width = 36
            row_height = 7

            # Table headers
            headers = ['State', 'Total Votes', 'Lib. Party', 'Rep
                . Party', 'Mgt. Party']
            for header in headers:
                pdf.cell(col_width, row_height, header, 1, 0, 'C'
                    )
            pdf.ln()

            # Table data
            for _, row in state_data.iterrows():
                pdf.cell(col_width, row_height, str(row['State'])
                    [:15], 1)
                pdf.cell(col_width, row_height, str(row['Total
                    Votes']), 1)
                pdf.cell(col_width, row_height, str(row['
                    Liberation Party']), 1)
                pdf.cell(col_width, row_height, str(row['United
                    Republic Party']), 1)
                pdf.cell(col_width, row_height, str(row['
                    Management Party']), 1)
                pdf.ln()

        # Save the PDF to a temporary file
        temp_pdf = "temp_dashboard.pdf"
        pdf.output(temp_pdf)

        # Read the temporary file and return its contents
        with open(temp_pdf, 'rb') as file:
            pdf_data = file.read()

        # Remove the temporary file
        import os
        os.remove(temp_pdf)

        return pdf_data

    except Exception as e:
        st.error(f"Error generating PDF: {str(e)}")
        return None

# download button code
def create_download_buttons():
```

```python
    st.sidebar.markdown("### Download Options")

    # Get data outside the button callbacks
    votes_data = get_votes_by_candidate()

    # CSV Download - Direct download
    csv_data = votes_data.to_csv(index=False).encode('utf-8') if
        not votes_data.empty else None
    st.sidebar.download_button(
        label="Download Data (CSV)",
        data=csv_data if csv_data else "No data available",
        file_name=f"voting_data_{datetime.now().strftime('%Y%m%d_
            %H%M')}.csv",
        mime="text/csv",
        disabled=votes_data.empty,
        type="primary"
    )

    # PDF Download - Direct download
    try:
        pdf_data = generate_dashboard_pdf()
        st.sidebar.download_button(
            label="Download Data (PDF)",
            data=pdf_data if pdf_data else "No data available",
            file_name=f"voting_dashboard_{datetime.now().strftime
                ('%Y%m%d_%H%M')}.pdf",
            mime="application/pdf",
            disabled=pdf_data is None,
            type="secondary"
        )
    except Exception as e:
        st.sidebar.error(f"Error preparing PDF: {str(e)}")

def main():
    # Sidebar
    with st.sidebar:
        st.title("Dashboard Controls")

        # Time range filter
        time_filter = st.selectbox(
            "Time Range",
            ["Last Hour", "Last 3 Hours", "Last 6 Hours", "All
                Time"],
            index=3
        )

        # Add simplified download buttons
        create_download_buttons()

    # Main content
    st.title('          Real-time Voting Dashboard')
```

```python
st.write("Live voting results updated every 30 seconds")

# Top metrics row
col1, col2, col3, col4 = st.columns(4)

with st.spinner("Loading metrics..."):
    total_votes, last_update, hourly_change = get_total_votes
        ()
    votes_by_candidate = get_votes_by_candidate()

    # Total Votes
    with col1:
        st.metric(
            "Total Votes Cast",
            f"{total_votes:,}",
            delta=f"{hourly_change:+,} in last hour" if
                hourly_change else None
        )

    # Leading Candidate
    # In the metrics section
    with col2:
        if not votes_by_candidate.empty:
            leader = votes_by_candidate.iloc[0]
            # Create two columns for image and info
            img_col, info_col = st.columns([1, 4])

            with img_col:
                # Get candidate image
                candidate_info = execute_query(f"""
                    SELECT img_url
                    FROM candidate
                    WHERE first_name = '{leader['first_name
                        ']}'
                    AND last_name = '{leader['last_name']}'
                """)
                if not candidate_info.empty and
                    candidate_info.iloc[0]['img_url']:
                    st.image(candidate_info.iloc[0]['img_url'
                        ], width=50)

            with info_col:
                st.metric(
                    "Leading Candidate",
                    f"{leader['first_name']} {leader['
                        last_name']}",
                    f"{leader['party']} ({leader['percentage
                        ']}%)"
                )
                # Add vote count below metric
                st.write(f"{leader['vote_count']} votes")
```

```python
        # Active States
        with col3:
            votes_by_state, _ = get_geographical_data()
            active_states = len(votes_by_state) if not
                votes_by_state.empty else 0
            st.metric(
                "Active States",
                active_states,
                "Currently Voting"
            )

        # Last Updated
        with col4:
            st.metric(
                "Last Updated",
                datetime.now().strftime('%H:%M:%S'),
                datetime.now().strftime('%Y-%m-%d')
            )

    # Vote Distribution and Candidate Performance
    st.markdown("---")
    col1, col2 = st.columns(2)

    with col1:
        st.subheader('Vote Distribution by Party')
        if not votes_by_candidate.empty:
            party_data = votes_by_candidate.groupby('party')['
                vote_count'].sum().reset_index()
            fig = px.pie(
                party_data,
                values='vote_count',
                names='party',
                hole=0.3,
                color='party',
                color_discrete_map=PARTY_COLORS
            )
            fig.update_traces(textinfo='percent+label')
            fig.update_layout(
                height=330,
                margin=dict(t=0, b=0, l=0, r=0),
                showlegend=True
            )
            st.plotly_chart(fig, use_container_width=True)
        else:
            st.warning("No party distribution data available")

    with col2:
        st.subheader('Votes by Candidate')
        if not votes_by_candidate.empty:
```

```python
            bar_chart = alt.Chart(votes_by_candidate).mark_bar().
                encode(
                  x=alt.X('vote_count:Q', title='Number of Votes'),
                  y=alt.Y('first_name:N', title='Candidate', sort='
                      -x'),
                  color=alt.Color(
                      'party:N',
                      scale=alt.Scale(domain=list(PARTY_COLORS.keys
                          ()),
                                      range=list(PARTY_COLORS.values
                                          ()))
                  ),
                  tooltip=['first_name', 'last_name', 'party', '
                      vote_count', 'percentage']
            ).properties(height=400)

            text = bar_chart.mark_text(
                align='left',
                baseline='middle',
                dx=5
            ).encode(
                text=alt.Text('vote_count:Q', format=',d')
            )

            st.altair_chart(bar_chart + text, use_container_width
                =True)
        else:
            st.warning("No candidate data available")

    # Historical Trends
    st.markdown("---")
    st.subheader('Voting Trends Over Time')

    trends_data = get_historical_trends()
    if not trends_data.empty:
        line_chart = alt.Chart(trends_data).mark_line(point=True)
            .encode(
            x=alt.X('vote_time:T', title='Time'),
            y=alt.Y('total_votes:Q', title='Total Votes'),
            color=alt.Color(
                'party:N',
                scale=alt.Scale(domain=list(PARTY_COLORS.keys()),
                                range=list(PARTY_COLORS.values()))
            ),
            tooltip=['vote_time', 'candidate_name', 'party', '
                total_votes']
        ).properties(height=400)

        st.altair_chart(line_chart, use_container_width=True)
    else:
        st.warning("No historical trend data available")
```

```python
    # Continue with the geographical visualization
    st.markdown("---")
    st.subheader('Geographical Vote Distribution')

    map_col1, map_col2 = st.columns(2)

    with st.spinner("Loading geographical data..."):
        votes_by_state, leading_party = get_geographical_data()

        if not votes_by_state.empty and not leading_party.empty:
            try:
                # Load US states geometry
                us_states = gpd.read_file('https://raw.
                    githubusercontent.com/PublicaMundi/MappingAPI/
                    master/data/geojson/us-states.json')

                with map_col1:
                    st.write("Total Votes by State")
                    merged_data = us_states.merge(
                        votes_by_state,
                        left_on='name',
                        right_on='address_state',
                        how='left'
                    )

                    if not merged_data.empty:
                        fig1 = px.choropleth(
                            merged_data,
                            geojson=merged_data.geometry,
                            locations=merged_data.index,
                            color='vote_count',
                            color_continuous_scale="Viridis",
                            hover_name="name",
                            hover_data=["parties", "vote_count"]
                        )
                        fig1.update_geos(scope='usa',
                            projection_scale=1.2)
                        fig1.update_layout(margin={"r":0,"t":0,"l
                            ":0,"b":0}, height=500)
                        st.plotly_chart(fig1, use_container_width
                            =True)

                with map_col2:
                    st.write("Leading Party by State")
                    merged_party_data = us_states.merge(
                        leading_party,
                        left_on='name',
                        right_on='address_state',
                        how='left'
                    )
```

```python
                    if not merged_party_data.empty:
                        fig2 = px.choropleth(
                            merged_party_data,
                            geojson=merged_party_data.geometry,
                            locations=merged_party_data.index,
                            color='party',
                            color_discrete_map=PARTY_COLORS,
                            hover_name="name",
                            hover_data=["party", "party_votes"]
                        )
                        fig2.update_geos(scope='usa',
                            projection_scale=1.2)
                        fig2.update_layout(margin={"r":0,"t":0,"l
                            ":0,"b":0}, height=500)
                        st.plotly_chart(fig2, use_container_width
                            =True)
            except Exception as e:
                st.error(f"Error creating map visualization: {e}"
                    )
        else:
            st.warning("No geographical data available")

# Demographic Analysis
st.markdown("---")
st.subheader('Voter Demographics')

demo_col1, demo_col2 = st.columns(2)

with st.spinner("Loading demographic data..."):
    gender_data, age_data = get_demographic_data()

    with demo_col1:
        st.write("Gender Distribution")
        if not gender_data.empty:
            fig_gender = px.pie(
                gender_data,
                values='percentage',
                names='gender',
                hole=0.3,
                color_discrete_sequence=['#0072C6', '#87CEEB'
                    ]
            )
            fig_gender.update_traces(textinfo='percent+label'
                )
            fig_gender.update_layout(
                height=330,
                margin=dict(t=30, b=0, l=0, r=0),
                showlegend=True
            )
```

```python
                    st.plotly_chart(fig_gender, use_container_width=
                        True)
                else:
                    st.warning("No gender distribution data available
                        ")

        with demo_col2:
            st.write("Age Distribution")
            if not age_data.empty:
                fig_age = px.bar(
                    age_data,
                    x='age_group',
                    y='count',
                    text='percentage',
                    # Using a purple/violet color that's distinct
                        from the gender chart
                    color_discrete_sequence=['#7B3FF3'] * len(
                        age_data),  # or
                    # Alternative colors you could use:
                    # '#FF6B6B' (coral red)
                    # '#38B6FF' (bright cyan)
                    # '#FFB302' (golden yellow)
                )
                fig_age.update_traces(
                    texttemplate='%{text:.1f}%',
                    textposition='outside'
                )
                fig_age.update_layout(
                    height=330,
                    margin=dict(t=30, b=0, l=0, r=0),
                    yaxis_title="Number of Voters",
                    xaxis_title="Age Group",
                    showlegend=False  # Remove legend since we're
                        using a single color
                )
                st.plotly_chart(fig_age, use_container_width=True
                    )
            else:
                st.warning("No age distribution data available")

    # State-level Analysis

    st.subheader("State-Level Voting Details")
    state_details = get_state_voting_details()

    # Add search filter for states
    search_state = st.text_input("Search state:", "")
    if search_state:
        state_details = state_details[state_details['State'].str.
            contains(search_state, case=False)]
```

```python
# Function to create alternating backgrounds
def highlight_rows(row):
    if row.name % 2 == 0:
        return ['background-color: rgba(242, 245, 250, 1)'] * \
            len(row)
    return [''] * len(row)

# Style the dataframe using Pandas styling
styled_df = state_details.style\
    .apply(highlight_rows, axis=1)\
    .set_properties(**{
        'padding': '8px 12px',
        'font-size': '14px'
    })\
    .set_table_styles([
        {'selector': 'th', 'props': [
            ('background-color', 'white'),
            ('color', 'rgb(49, 51, 63)'),
            ('font-weight', 'normal'),
            ('border-bottom', '1px solid #ddd'),
            ('padding', '8px 12px'),
            ('font-size', '14px')
        ]},
        {'selector': 'td', 'props': [
            ('border', 'none')
        ]},
        {'selector': 'table', 'props': [
            ('border-collapse', 'collapse'),
            ('border', 'none')
        ]}
    ])\
    .hide(axis="index")

# Display the styled dataframe
st.dataframe(
    styled_df,
    column_config={
        "State": st.column_config.TextColumn("State"),
        "Region": st.column_config.TextColumn("Region"),
        "Management Party": st.column_config.NumberColumn("
            Management Party", format="%d"),
        "Liberation Party": st.column_config.NumberColumn("
            Liberation Party", format="%d"),
        "United Republic Party": st.column_config.
            NumberColumn("United Republic Party", format="%d")
            ,
        "Total Votes": st.column_config.NumberColumn("Total
            Votes", format="%d"),
        "Avg Age": st.column_config.NumberColumn("Average Age
            ", format="%.1f"),
```

```python
                "Male %": st.column_config.NumberColumn("Male/Female
                    Ratio", format="%.1f%%")
        },
        hide_index=True,
        use_container_width=True
    )

    # Candidate information
    st.subheader('Candidate Information')
    candidate_info = get_candidate_info()
    for _, candidate in candidate_info.iterrows():
        with st.expander(f"{candidate['first_name']} {candidate['
            last_name']} ({candidate['party']})"):
            col1, col2 = st.columns([1, 3])
            with col1:
                st.image(candidate['img_url'], width=150)
            with col2:
                st.write(f"**Age:** {candidate['age']}")
                st.write(f"**Gender:** {candidate['gender']}")
                st.write(f"**Biography:** {candidate['biography
                    ']}")

    # Add footer
    st.markdown("---")
    st.markdown(
        """
        <div style='text-align: center; color: #666;'>
            <p>
                <small>
                    Data updates automatically every 30 seconds.
                        Last update: {}<br>
                        Dashboard refreshed at: {}
                </small>
            </p>
        </div>
        """.format(
            last_update.strftime('%Y-%m-%d %H:%M:%S') if
                last_update else "N/A",
            datetime.now().strftime('%Y-%m-%d %H:%M:%S')
        ),
        unsafe_allow_html=True
    )

if __name__ == "__main__":
    main()
```