GDC Cancer Data Warehouse using Redis, MongoDB, PostgresSQL and Kafka

by

Suleyman Selcuk and Ømer Cetinkaya

in

IKT453-G 25V

Intelligent Data Management

Supervised by Vladimir Iosifovich Zadorozny

Grimstad, Spring 2025

Mandatory Group Declaration

Each student is individually responsible for understanding what are considered legal aids, the guidelines for their use, and the rules for referencing sources. This declaration is intended to raise awareness among students of their responsibilities and the consequences that cheating may lead to. The lack of a declaration does not absolve students from their responsibilities.

1.	We hereby declare that our submission is our own work, and that we have not used other sources or received any help other than what is mentioned in the submission.	**Yes** / No
2.	**We further declare that this submission:** • Has not been used for any other exam at another department/university/college, either domestically or abroad. • Does not reference others' work without giving credit. • Does not reference our own previous work without stating it. • Lists all references in the bibliography. • Is not a copy, duplicate, or transcription of others' work or submissions.	**Yes** / No
3.	We are aware that violations of the above may be considered cheating and can result in the annulment of the exam and exclusion from universities and colleges in Norway, according to the Universities and Colleges Act §§4-7 and 4-8 and the Regulations on Examinations §§31.	**Yes** / No
4.	We are aware that all submitted assignments may be subject to plagiarism checks.	**Yes** / No
5.	We are aware that the University of Agder will handle all cases where there is suspicion of cheating according to the university's guidelines for handling cheating cases.	**Yes** / No
6.	We have familiarized ourselves with the rules and guidelines for using sources and references on the library's website.	**Yes** / No
7.	A majority of us have agreed that the effort within the group is noticeably different, and therefore wish to be assessed individually. Normally, all participants in the project are assessed together.	Yes / **No**

Publishing Agreement

Power of attorney for electronic publication of the assignment The author(s) hold the copyright to the assignment. This means, among other things, the exclusive right to make the work available to the public (Copyright Act §2).
Assignments that are exempt from public disclosure or confidential will not be published.

We hereby grant the University of Agder a non-compensated right to make the assignment available for electronic publication:	Yes / **No**
Is the assignment confidential?	Yes / **No**
Is the assignment exempt from public disclosure?	**Yes** / No

Preface

This project was carried out as part of the course IKT453-G 25V Intelligent Data Management at University of Agder. The objective of the project was to develop a comprehensive solution for integrating clinical data from the GDC (Genomic Data Commons) API, using various technologies such as MongoDB, PostgreSQL, Redis, and Kafka, along with a React frontend for data visualization. The primary aim was to create a system capable of fetching, processing, storing, and displaying clinical data in an accessible way for users, while also demonstrating how these different technologies could work together seamlessly.

Contribution

Suleyman Selcuk and Ømer Cetinkaya collaborated closely throughout the project using pair programming and joint planning sessions. While both contributed to all aspects of the work, their roles were defined by their strengths and areas of focus.

Suleyman Selcuk was primarily responsible for the development work. He led the coding efforts on the frontend using ReactJS and TypeScript, implementing core features such as data selection, updates, deletions, and insertions across the different databases. Suleyman's contributions were crucial in building a user-friendly interface and ensuring smooth interactions across the integrated systems.

Ømer Cetinkaya focused on the architectural aspects of the project. He designed the overall system architecture, selecting and integrating the various technologies—including PostgreSQL, MongoDB, Redis, Kafka, and Docker—to form a cohesive solution. Ømer's work ensured that the project was scalable, robust, and aligned with industry best practices, supporting the vision for a data warehouse and research platform.

Together, through continuous collaboration and pair programming, they successfully developed an MVP that demonstrates the potential of the integrated system and sets the foundation for future enhancements.

The source code for the project can be found in this Github repository.

Contents

1 Introduction

The integration of multiple technologies for real-time data processing and visualization is increasingly important in today's data-driven world. In this project, we focus on building a web application that integrates several modern technologies to provide real-time data streaming and interactive data visualizations. Specifically, the project involves fetching clinical cancer data from the **Genomic Data Commons (GDC) API**, processing the data using a Kafka-based message broker, and storing it in PostgreSQL, Redis, and MongoDB for various use cases. The data is then displayed on a web frontend built with ReactJS, providing real-time insights into the cancer datasets.

The core goal of this project is to create a working demonstration of how technologies such as Kafka, PostgreSQL, Redis, MongoDB, Docker, Node.js, Vite, and ExpressJS can be integrated to fetch, process, store, and display real-time data. The data from the GDC API will be used as the basis for the cancer data integration. The application will allow users to view different types of cancer data through an interactive and user-friendly web interface, showcasing how each technology plays a role in the process.

We decided to incorporate three types of databases into this project as a deliberate challenge to explore and integrate diverse technologies. While Redis is clearly valuable for caching and enhancing performance, and PostgreSQL offers robust relational data management, the inclusion of MongoDB was primarily for experimental purposes. Although PostgreSQL might logically suffice for our needs, integrating MongoDB allowed us to push our boundaries, learn from using a NoSQL approach, and determine how well these systems can work together at once.

This project aims to solve the problem of efficiently managing and visualizing large-scale clinical data. Cancer research often involves dealing with vast amounts of data that require high-performance systems to ensure real-time processing and access. By leveraging Kafka for real-time data streaming, PostgreSQL for analytics, MongoDB for storing clinical records, and Redis for caching, this project demonstrates how to integrate these systems for an end-to-end solution.

2 Theoretical background

This project integrates a wide range of modern technologies that allow for the efficient management, analysis, and display of clinical data. Below is a brief overview of each of the key technologies employed in the project.

2.1 PostgreSQL

PostgreSQL is an open-source relational database management system (RDBMS) that is widely used for managing structured data. It is known for its robustness, scalability, and support for advanced features such as ACID compliance and complex queries. In this project, PostgreSQL is used for storing and analyzing analytics data. The structured nature of PostgreSQL makes it ideal for querying large sets of structured data, such as the insights generated from the GDC clinical data.[6]

2.2 Redis

Redis is an in-memory data store commonly used for caching purposes. It supports various data structures such as strings, hashes, lists, sets, and sorted sets. Redis is used in this project to cache data from the GDC API and improve the performance of data retrieval. By storing frequently accessed data in memory, Redis significantly reduces the response time when fetching clinical data or analytics insights, thus optimizing the overall performance of the system.[8]

2.3 MongoDB

MongoDB is a NoSQL document-oriented database. Unlike traditional relational databases, MongoDB stores data in flexible, JSON-like documents, making it suitable for handling unstructured or semi-structured data. In this project, MongoDB is used to store clinical records, which are often complex and variable in structure. The ability to store clinical data in a flexible format allows the system to easily accommodate changes in data schema without requiring complex migrations.[4]

2.4 Kafka

Apache Kafka is a distributed event streaming platform that is used for building real-time data pipelines and streaming applications. Kafka provides high throughput, fault tolerance, and scalability, making it an excellent choice for handling large volumes of streaming data. In this project, Kafka is used to stream clinical data, analytics, and cache updates between different services in real-time. It acts as a central communication hub between the backend services and the frontend, enabling

real-time updates and synchronization of the data stored in PostgreSQL, MongoDB, and Redis.[1]

2.5 Docker

Docker is a platform that enables developers to package applications and their dependencies into containers. Containers are lightweight, portable, and ensure that applications run consistently across different environments. In this project, Docker is used to containerize the entire application stack, including PostgreSQL, Redis, MongoDB, Kafka, and the backend services. Docker ensures that the application runs reliably and efficiently, whether it's being developed locally or deployed to production.[2]

2.6 Node.js

Node.js is a runtime environment that allows JavaScript to be run on the server side. It is built on Chrome's V8 JavaScript engine and provides a non-blocking, event-driven architecture, making it ideal for building scalable and performant web applications. In this project, Node.js is used as the backend runtime environment, handling API requests, integrating with databases, and managing Kafka communication. [5]

2.7 Vite

Vite is a modern, fast build tool and development server for frontend projects. It is optimized for performance and provides fast hot module replacement (HMR), making it ideal for rapid development of React applications. In this project, Vite is used to bundle and serve the React frontend. Its fast development server and optimized build pipeline help streamline the development process.[5]

2.8 Express.js

Express.js is a minimalist web framework for Node.js, commonly used for building APIs and web applications. It simplifies the process of handling HTTP requests, routing, and middleware in Node.js applications. In this project, Express.js is used to build the API layer, handling requests from the frontend, processing them, and interacting with Kafka and databases. [3]

2.9 React.js

React.js is a popular JavaScript library for building user interfaces, particularly single-page applications (SPAs). React provides a component-based architecture,

allowing developers to create reusable UI components and manage the state of the application efficiently. In this project, React.js is used to build the frontend, which interacts with the backend API, displays clinical data and analytics, and integrates with Kafka for real-time data updates.[7]

2.10 TailwindCSS

TailwindCSS is a utility-first CSS framework that provides a set of low-level utility classes to build custom designs without writing custom CSS. It promotes a more efficient and maintainable approach to styling by using predefined classes for layout, spacing, typography, and other common design elements. In this project, TailwindCSS is used to style the frontend, enabling rapid development of a clean and responsive user interface.[9]

2.11 TypeScript

TypeScript is a statically typed superset of JavaScript that adds optional type checking to the language. It helps catch errors at compile time and improves code maintainability. In this project, TypeScript is used throughout both the frontend and backend code to ensure type safety, improve developer productivity, and facilitate better tooling support. TypeScript is particularly useful for larger applications, where the type system helps catch bugs and improve code quality.[10]

3 System Overview and User Identification

This section provides an overview of the system, identifies the various types of users and administrators, lists assumptions made about the system, describes the data maintained, and outlines the data loading process.

3.1 System Overview

The system is designed to facilitate the flow of cancer-related data from the GDC Cancer API to various stakeholders through a robust architecture involving data streaming, storage, and real-time analytics. It integrates several technologies to ensure seamless data flow and accessibility, including Apache Kafka for data streaming, ExpressJS and NodeJS for backend services, PostgreSQL, MongoDB, and Redis for data storage and caching, and ReactJS for the frontend interface. Docker is used for containerization to facilitate deployment across different environments.

3.2 User Identification

The system caters to various types of users, each with specific roles and access levels:

- **Researchers:** Access the system to analyze clinical data and derive insights for cancer research. They primarily interact with the frontend to visualize data and perform exploratory analysis.
- **Clinicians:** Use the system to access patient data and analytics for informed decision-making in clinical settings. They require real-time updates and easy access to patient records.
- **System Administrators:** Responsible for maintaining the system's infrastructure, ensuring uptime, and managing user access. They interact with the backend and database systems to monitor performance and troubleshoot issues.
- **Data Administrators:** Oversee data integrity and manage data ingestion processes. They ensure that data from the GDC Cancer API is accurately streamed and stored in the databases.
- **Application Developers:** Work on enhancing the system's features and functionality. They are involved in both frontend and backend development, ensuring seamless integration and user experience.
- **Data Engineers:** Focus on optimizing data pipelines and storage solutions. They work with Kafka, databases, and caching mechanisms to ensure efficient data processing and retrieval.

3.3 Assumptions

1. The GDC Cancer API provides reliable and consistent data that can be ingested into the system.
2. Users have the necessary permissions and access rights to interact with the system components relevant to their roles.
3. The system will handle a moderate volume of data, with scalability options available for increased data loads.
4. Data privacy and security measures are in place to protect sensitive patient information.
5. The system will be deployed in an environment that supports Docker and the necessary database technologies.

3.4 Data Description

The system maintains various types of data, including:

- **Clinical Data:** Patient records, diagnosis details, treatment information, and outcomes.
- **Analytics Data:** Aggregated statistics, trends, and insights derived from clinical data.
- **Metadata:** Information about data sources, data quality, and processing logs.

3.5 Data Loading Process

The data loading process involves several key steps:

- **Data Ingestion:** Data is fetched from the GDC Cancer API and streamed into the system using Apache Kafka.
- **Data Cleaning:** Raw data is cleaned to remove duplicates, correct errors, and handle missing values. This ensures data quality and consistency.
- **Data Transformation:** Data is transformed into a suitable format for storage in PostgreSQL, MongoDB, and Redis. This includes normalization, denormalization, and aggregation as needed.
- **Data Storage:** Cleaned and transformed data is stored in the appropriate databases, with Redis used for caching frequently accessed data.

This process ensures that the system maintains high-quality data that is readily accessible for analysis and decision-making.

4 Implementation

This section describes the methodology adopted for the design and development of the system, focusing on the integration of Kafka, databases (PostgreSQL, Redis, MongoDB), and real-time updates, which serve as the backbone for processing and displaying cancer data from the GDC Cancer API. The overall architecture of the system consists of several components interacting in real time to fetch, process, store, and display data efficiently. The project also leverages Docker for containerization, ensuring the application is portable, scalable, and easy to deploy.

4.1 System Architecture Overview

The system is based on an event-driven architecture using Kafka, a distributed streaming platform, for real-time data processing and communication between various components of the system. The data flow starts with fetching clinical data from the GDC API, followed by processing and storing it in databases (PostgreSQL, MongoDB, and Redis). The frontend, built using React, communicates with the backend via an Express API, which provides access to the data stored in the databases and also pushes real-time updates to the frontend through WebSockets, providing a live data stream.

The architecture can be broken down into the following components:

- **Kafka:** Used as the central event-driven data pipeline that allows communication between services and provides real-time data updates.
- **GDC API:** The source of clinical data, specifically data related to cancer studies, such as disease types, consent timings, and primary sites.
- **PostgreSQL:** Used to store and analyze aggregated analytics data, which is relational and structured.
- **MongoDB:** Stores clinical records that may have unstructured or semi-structured data, making MongoDB an ideal choice for this data type.
- **Redis:** A caching layer that holds frequently accessed data, helping reduce latency and improve system performance.
- **ExpressJS API:** The backend server that processes client requests, interacts with the databases, and serves data to the frontend.
- **ReactJS Frontend:** A user interface that visualizes the data retrieved from the backend, with real-time updates powered by WebSockets.
- **Docker:** All components are containerized using Docker to ensure a consistent and scalable deployment environment.

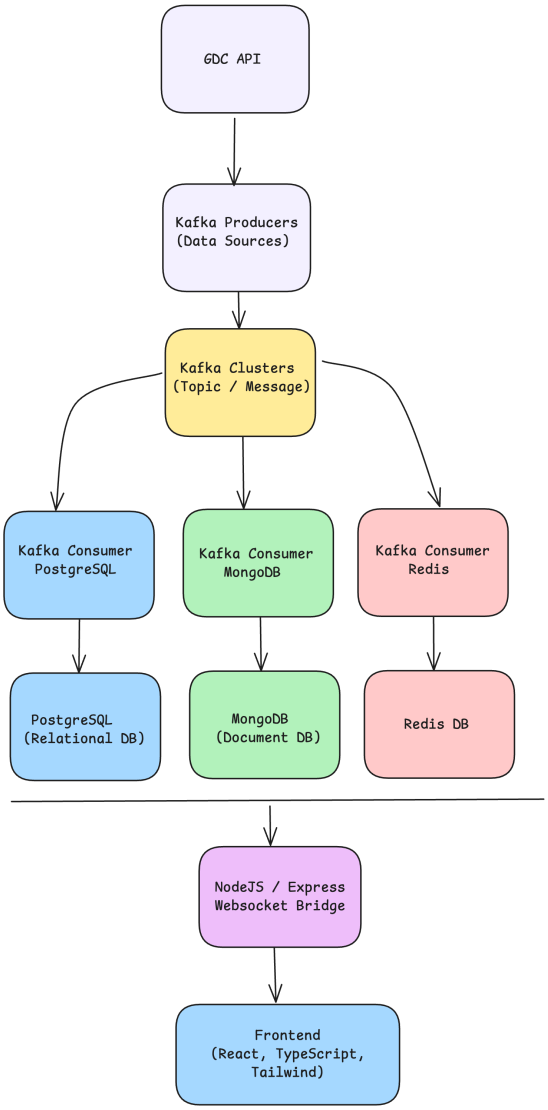

Figure 1: System architecture

4.2 Kafka as the Central Data Streaming Layer

Kafka serves as the backbone of the system by managing real-time data streams and enabling the communication between the various components. The choice of Kafka is driven by its ability to handle high-throughput data streams, its scalability, and its ability to decouple services in an efficient way. Kafka topics are used to categorize and distribute different types of data. For this project, three primary topics are used:

- **clinical-data:** Contains clinical records fetched from the GDC API.
- **analytics-data:** Stores aggregated or processed analytics data that is stored in PostgreSQL.
- **cache-data:** Holds frequently accessed data that is cached in Redis to reduce database load and improve performance.

The system uses Kafka producers to send data to these topics and Kafka consumers to retrieve data from them. The Kafka consumers process the incoming data and then populate the respective databases (PostgreSQL, MongoDB, and Redis) with the data. This architecture decouples data producers and consumers, which allows the system to scale independently and respond to real-time changes effectively.

4.3 Fetching Data from the GDC API

The data retrieval process begins with the GDC Cancer API, a public data source for cancer research, providing a wealth of information about clinical trials, cancer studies, and patient data. The API is queried to fetch relevant datasets based on certain parameters, such as disease type, consent timings, and primary site. The data obtained from the API is highly structured but can contain different types of information, ranging from numeric data like patient age and survival times to categorical data like disease type and treatment protocols.

The request to the GDC API is made periodically to keep the data up-to-date. The fetched data is then processed and serialized into JSON format, which is ready for transmission into Kafka topics. This data is used to populate the three databases:

- **Clinical data (MongoDB):** Patient and disease-related data, such as tumor type, disease stage, and treatment outcomes, are inserted into MongoDB as unstructured documents.
- **Analytics data (PostgreSQL):** Aggregated statistics, such as survival rates, treatment efficacy, and disease prevalence, are stored in PostgreSQL as structured records that can be queried using SQL.
- **Cache data (Redis):** Frequently requested data, such as common disease types or recent clinical trials, are cached in Redis for quick retrieval.

4.4 Database Population and Interaction with Kafka

Once the data is streamed into Kafka topics, Kafka consumers continuously listen to these topics and process the messages. Each consumer is designed to handle specific data and populate the corresponding database. For example, when new clinical data is received in the `clinical-data` topic, the consumer for this topic reads the

data and stores it in MongoDB. Similarly, when analytics data is produced into the `analytics-data` topic, it is stored in PostgreSQL, and frequently requested data is cached in Redis.

The advantage of using Kafka in this manner is that it allows for real-time processing of incoming data and ensures that the data in the databases is always synchronized with the latest updates. By decoupling the data production and consumption process, Kafka ensures that each database is populated independently, reducing the complexity of the system.

The interaction between Kafka and the databases ensures that data is stored in the most appropriate system:

- **MongoDB:** MongoDB stores unstructured data such as clinical records and patient information. Its flexible schema allows for the easy accommodation of varied data types, making it ideal for storing medical data.
- **PostgreSQL:** PostgreSQL is used for structured analytics data, where complex queries and reporting are required. It provides powerful querying capabilities, making it the ideal choice for handling aggregated and statistical data.
- **Redis:** Redis serves as a high-performance caching layer, reducing the need to query the databases repeatedly for frequently accessed data. This ensures faster response times for commonly requested data and reduces the load on the databases.

4.5 ExpressJS API: Data Access and Real-Time Updates

The ExpressJS API is responsible for handling client requests and providing access to the data stored in the databases. It exposes various endpoints for fetching data from PostgreSQL, MongoDB, and Redis, which the frontend can use to display data to users. The API also handles the real-time streaming of data from Kafka, ensuring that the frontend receives live updates as new data is produced into Kafka topics.

One of the key features of the API is its ability to listen for Kafka messages and send updates to the frontend via WebSockets. WebSockets provide a persistent connection between the backend and the frontend, allowing the server to push data to the frontend in real-time. This is essential for updating the user interface without the need for periodic polling or page refreshes.

4.6 Frontend Visualization with React

The frontend of the application is built using ReactJS, providing an interactive and responsive user interface that displays data to the user in various formats, such as

charts, tables, and graphs. The frontend communicates with the backend via the ExpressJS API, fetching data from the databases and displaying it on the page.

A key feature of the frontend is its ability to receive real-time updates from Kafka. The React components are connected to the WebSocket server, which listens for incoming messages from the backend. Whenever new data is received via Kafka, the frontend is updated in real-time to reflect the latest information. This allows the user to see live updates without needing to refresh the page.

4.7 Real-Time Data Streaming with Kafka

The real-time data streaming functionality is powered by Kafka, which streams data to the frontend through the WebSocket connection. As data is produced into Kafka topics, the WebSocket server pushes the messages to the frontend, allowing users to see updates as they happen. The `KafkaStream` component on the frontend displays the data received from Kafka, providing users with real-time feedback about the data flow.

The real-time aspect is crucial in scenarios where the data changes frequently, and users need up-to-date information. Kafka's ability to manage high-throughput data streams ensures that the system can scale to handle large amounts of data without affecting performance.

4.8 Dockerized Application for Scalability and Portability

To ensure the system is scalable and portable, all components are containerized using Docker. Each service: PostgreSQL, MongoDB, Redis, Kafka, ExpressJS, and React - is packaged into its own Docker container. Docker ensures that the application runs consistently across different environments, whether it is in development, testing, or production.

Using Docker for containerization allows the application to be easily deployed on any infrastructure, whether on a local machine, on-premise servers, or in the cloud. Additionally, Docker makes it easier to scale the application by adding or removing containers as needed. This is particularly important for services like Kafka, where the ability to scale horizontally is essential for handling high-throughput data streams.

The methodology described here outlines the steps taken to design and implement the system, from fetching and processing data from the GDC API to displaying it in real-time on the frontend.

5 Requirements

This document outlines the planned requirements using the MoSCoW prioritization method. We don't have every detail yet, but this blueprint shows how the system is planned to evolve. The "Must Have" features—such as the ability to fetch clinical data—are critical for the initial release, while "Should Have," "Could Have," and "Won't Have" requirements will be defined and refined over time as the project progresses.

5.0.1 Must Have

- As a user, I want to be able to fetch clinical data from the GDC API.
- As a user, I want to see real-time updates of clinical data in my frontend application.
- As a user, I want the application to display analytics data stored in PostgreSQL.
- As a user, I want the system to cache frequently accessed data using Redis for faster performance.
- As a user, I want to interact with the frontend via ReactJS, displaying clinical and analytics data.
- As a user, I want to be able to fetch new data by pressing a "Get new data" button, which will refresh the data from the GDC API and update the frontend.
- As a user, I want to see the Kafka stream status and data received in real-time.
- As a user, I want the data to be stored correctly in PostgreSQL, MongoDB, and Redis, based on the type of data.

5.0.2 Should Have

- As a user, I should be able to filter data based on specific attributes (e.g., disease type, consent timing).
- As a user, I should be able to see an overview of the data in an easy-to-understand format, such as charts or tables.
- As a user, I should be able to interact with data, such as sorting and searching clinical records.
- As a user, I should be able to refresh the frontend without reloading the entire page.
- As a user, I should be able to receive notifications when the data is updated.

5.0.3 Nice to Have

- As a user, I would like to be able to download the data displayed on the frontend as CSV or Excel files.
- As a user, I would like to have advanced filtering options, such as grouping data by disease type or primary site.
- As a user, I would like to visualize the data in multiple formats, such as bar charts, pie charts, or line graphs.

5.0.4 Will Not Have

Features that were considered but decided not to be included in the project:

- Real-time collaboration between users in the frontend (e.g., multiple users viewing and interacting with the same data).
- Data export functionality to formats other than CSV or Excel (e.g., PDF).

5.1 Non-Functional Requirements

Non-functional requirements define essential characteristics of the system that underpin performance, security, and overall user experience. Although these requirements are not directly linked to user actions, they establish measurable benchmarks in areas such as performance, privacy, and quality. Quantifying these requirements is crucial for ensuring that system expectations are both clear and attainable.

5.1.1 Must Have

- The website must load within 5 seconds.
- Passwords must be securely stored using robust hashing algorithms (e.g., bcrypt).
- Personal data must be permanently removed when a user's profile is deleted.
- The system must support a minimum of 100 concurrent users without performance degradation.
- Data must be encrypted in transit using HTTPS/SSL to ensure secure communications.
- Real-time updates shall be pushed to the frontend within 1 second after data is produced to Kafka.

5.1.2 Should Have

- The system should scale horizontally to support increasing data volumes and user loads.
- Frontend data retrieval should complete within 2 seconds under optimal conditions.
- All major system actions and errors should be logged for effective monitoring and troubleshooting.
- The application should be compatible with modern browsers such as Chrome, Firefox, and Edge.

5.1.3 Nice to Have

- The application should be mobile-friendly, adapting responsively to various screen sizes.
- The system may support third-party authentication services (e.g., Google or Facebook logins).
- The frontend UI could offer customizable options such as a dark mode to enhance user experience.

5.1.4 Will Not Have

- The application will not support voice commands or voice-assisted navigation.
- Integration with external third-party APIs, except for the GDC API, is beyond the scope of this system.

6 Solution

6.1 Product

The product described in this section is a proposed solution that meets all the functional requirements outlined as must have previously in the report. This section presents the conceptualization of the product, which serves as the foundation for the upcoming implementation. The goal of this stage is to ensure that the design can effectively meet all the requirements before proceeding with the actual development.

At this stage of the project, the product is represented through diagrams, sketches, or prototypes. These designs help in visualizing the overall structure, user interfaces, and user interactions within the system. They provide a clear understanding of how the application will function and how users will interact with the system.

The design of the system aligns with the goal of delivering a Minimum Viable Product (MVP), which includes the core features necessary for the solution to function properly. The design iterations ensure that features are developed in a way that can be tested early on and adjusted based on user feedback.

Each design choice is made in direct reference to fulfilling the functional requirements set forth earlier in the project. The design was developed to provide a high-level understanding of the system and to serve as a blueprint for implementation.

6.1.1 User interface

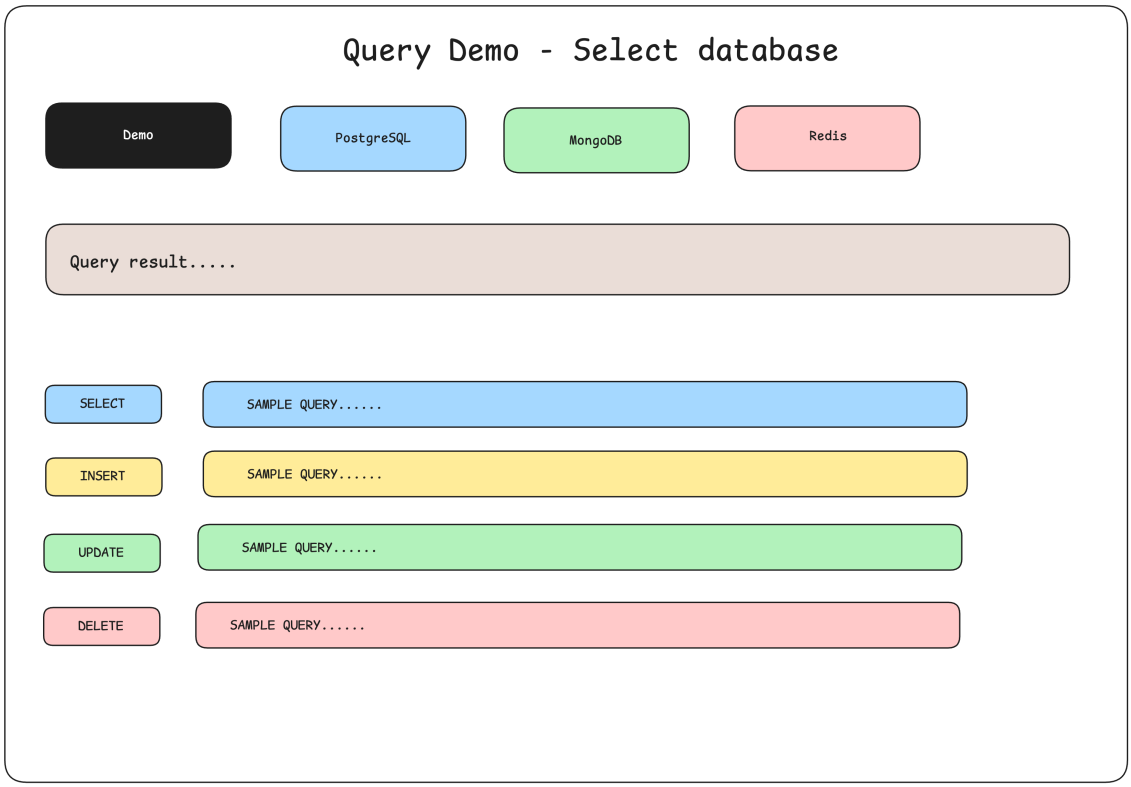

Figure 2: Query Demo page

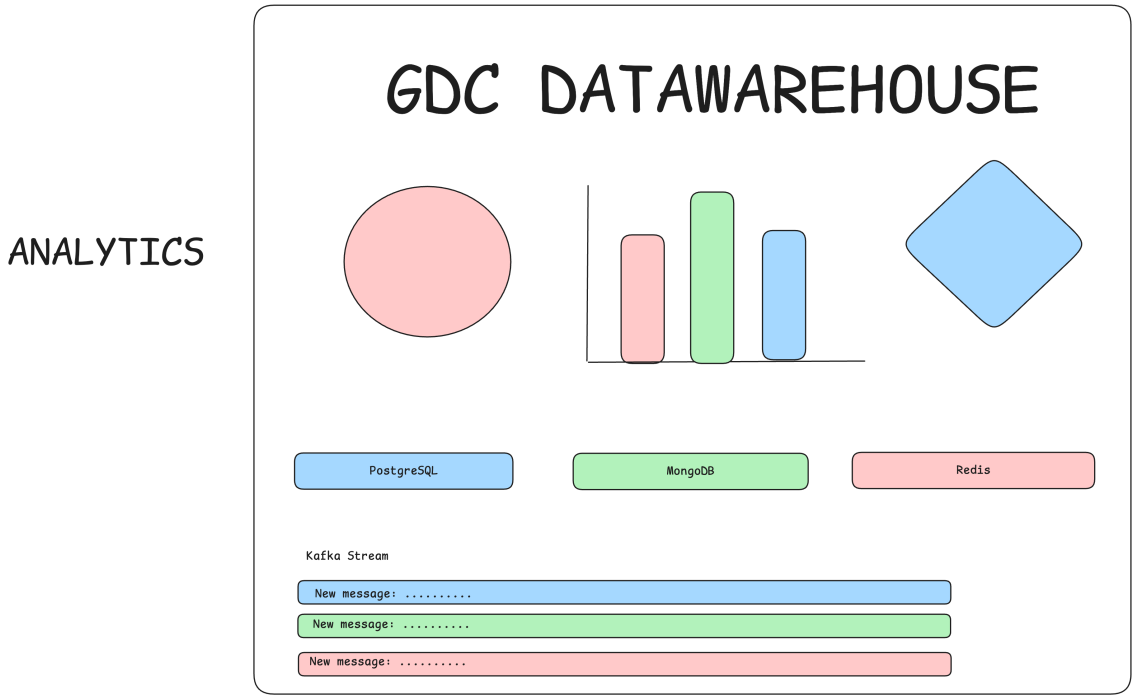

Figure 3: Analytics page (/demo page)

The layout is minimalistic, ensuring users can navigate the application without being overwhelmed by unnecessary complexity. The use of dropdown menus and modular components ensures scalability and flexibility in future feature additions. The main menu also adapts to different screen sizes, ensuring a responsive design that works on both desktop and mobile devices. The main menu serves as the primary navigation tool within the application. The design focuses on creating an intuitive and user-friendly interface that allows users to easily access various features of the system. Key functionalities like testing sample queries, triggering new data fetch requests, viewing clinical data, inspecting analytics, and observing the Kafka data stream are all readily accessible.

The main menu design is closely aligned with the goal of providing users with an easy-to-navigate interface, making the interaction process efficient and intuitive.

The application is designed to provide a comprehensive yet concise overview of the clinical data available in the system. The user will be presented with interactive elements like graphs, charts, and summary statistics that present a snapshot of the data.

Key elements of the application includes:

- **Clinical data summary**: A concise summary of the data available, including the number of records, disease types, and other relevant clinical metrics.
- **Data fetch trigger**: A button that triggers the process of fetching new data from the GDC API, populating the databases, and updating the system in real-time.
- **Kafka stream updates**: A component showing the latest data events that are flowing through the Kafka system, enabling users to see how data is processed and streamed.

The dashboard is interactive, providing users with the ability to click through various visual elements to explore the data in more detail. The data is updated automatically once new records are fetched via the Kafka stream.

Detailed analytics: It is planned that users can interact with the data through filter options and sorting mechanisms, enabling them to look at subsets of the data that interest them.
Data visualization: The application uses Chart.js to display clinical data, such as disease type distribution, patient count by age group, and other statistical analyses. This enables users to visualize trends and patterns in the data.
Real-time Kafka updates: The screen shows a real-time Kafka stream component, where users can track how new data is processed and displayed as it arrives in the system.

The goal of this screen is to give users a deep dive into the clinical data, providing them with a powerful tool to analyze the dataset and gain insights.

6.1.2 Kafka Integration Details

Kafka plays a crucial role in this system as it handles the real-time data streaming aspect. The integration with Kafka is designed to fetch new data from the GDC API and stream it into the system, populating the various databases (PostgreSQL, MongoDB, Redis) and making it available to users instantly.

The Kafka integration works as follows:

1. When the page renders, the frontend triggers a Kafka producer that fetches the clinical data from the GDC API.
2. The data is then sent to Kafka topics: ‘clinical‘, ‘analytics‘, and ‘cache‘.
3. These topics correspond to the relevant databases (MongoDB for clinical records, PostgreSQL for analytics, and Redis for caching).
4. Kafka consumers continuously consume these topics and update the corresponding databases. Once the data is populated, it is displayed in real-time on the

frontend for the user to interact with.

5. Kafka consumers continuously consume these topics and update the corresponding databases. Once the data is populated, it is displayed in real-time on the frontend for the user to interact with.

Kafka is essential for ensuring that the system can handle large volumes of data in real-time and deliver up-to-date information to users without any noticeable delay.

6.1.3 Database Integration Details

The system integrates with PostgreSQL, MongoDB, and Redis to store and manage the clinical data. These databases serve different purposes, with PostgreSQL handling analytics and complex queries, MongoDB storing unstructured clinical records, and Redis acting as a caching layer for quick data retrieval.

PostgreSQL: Used to store analytical data and perform complex queries. PostgreSQL provides advanced features for aggregating and analyzing large datasets, making it ideal for handling analytical workloads.
MongoDB: This NoSQL database stores the clinical records. MongoDB is chosen for its ability to handle semi-structured data, making it an excellent choice for storing medical records that may have varying schemas.
Redis: Redis is used as a caching layer to improve the performance of the application. Frequently accessed data is stored in Redis, ensuring faster retrieval times and reduced load on the other databases.

These databases are continuously updated by Kafka consumers as new data flows in. The integration ensures that the data remains consistent and up-to-date across the entire system.

6.2 Technological Choices

The technological choices made during the planning phase were critical in ensuring that the solution met both functional and non-functional requirements, including scalability, performance, and ease of use. In this section, we will discuss the main technologies selected and why they were chosen for this project.

6.2.1 Architecture

The architecture of the system is based on a microservices model that divides the application into distinct, independently scalable components. The use of microservices allows the system to scale more effectively and makes it easier to maintain and update different parts of the application.

The overall architecture includes:
Backend: Built using Node.js with ExpressJS for the API layer. This provides a lightweight, fast, and scalable solution for handling incoming requests and interacting with the databases and Kafka.
Frontend: Built using ReactJS, providing a dynamic and responsive user interface. React enables the creation of reusable components, which makes the frontend development more efficient.
Kafka: Used for real-time data streaming and communication between services. Kafka ensures that data flows seamlessly between the GDC API, the databases, and the frontend.
Databases: PostgreSQL, MongoDB, and Redis are used for different purposes, as previously mentioned. The system's ability to integrate with these databases efficiently is key to ensuring that the application can handle large amounts of data and provide real-time insights.

This architecture allows for easy scaling of individual components and ensures that the system can handle growing amounts of data as it expands.

6.2.2 Frameworks and Libraries

The following frameworks and libraries were selected to ensure the system meets the functional and non-functional requirements:
Node.js: Chosen for its asynchronous, event-driven model, which is well-suited for handling high volumes of real-time data. It ensures the backend can handle numerous connections efficiently.
ExpressJS: A minimal and flexible Node.js framework used to build the RESTful API. It provides a simple way to define routes and manage API calls.
ReactJS: A JavaScript library for building user interfaces. React allows for the development of fast, interactive UIs with reusable components, making it an ideal choice for building the frontend of the application.
KafkaJS: This library simplifies integration with Kafka by providing an easy-to-use API for producing and consuming messages. It ensures smooth interaction with Kafka topics.
Chart.js: A library for creating interactive charts. It is used in the frontend to visualize the clinical data, including disease types, patient demographics, and analytics.
TailwindCSS: A utility-first CSS framework that allows for quick and responsive styling of the frontend. It is easy to customize and helps maintain a consistent design.

The selection of these technologies ensures that the system is both performant and scalable while also being user-friendly and easy to maintain.

7 Implementation

This section outlines the implementation of the project, focusing on the core technical components that make up the solution. The goal of this section is to provide a clear understanding of the system's architecture, key components, and their interactions. We will present code excerpts that demonstrate the most important features of the project. The complete code will be included in the appendices.

7.1 Overall Architecture

The architecture of the project is designed to integrate several technologies in a seamless manner, allowing data to flow from the GDC Cancer API to the frontend via Kafka, with storage and caching managed by PostgreSQL, MongoDB, and Redis.

The system is divided into several key components:

- **GDC Cancer API Integration**: This is the data source for the project, from which clinical and analytics data are fetched.
- **Kafka**: Kafka is used to handle the streaming of data. It acts as the central messaging system that transmits data between the backend services and other components.
- **Backend (ExpressJS & NodeJS)**: The backend, built with ExpressJS and NodeJS, serves as the intermediary layer that connects the frontend to the databases and handles Kafka message consumption.
- **Databases**: PostgreSQL stores the analytics data, MongoDB holds the clinical records, and Redis acts as a caching layer to speed up access to frequently requested data.
- **Frontend (ReactJS & Vite)**: The frontend, built with ReactJS and styled using TailwindCSS, provides a user interface for displaying the clinical data, analytics, and Kafka stream updates.
- **Docker**: Docker is used to containerize the entire application, making it easier to deploy and run the system in different environments.

The overall architecture can be visualized as follows:

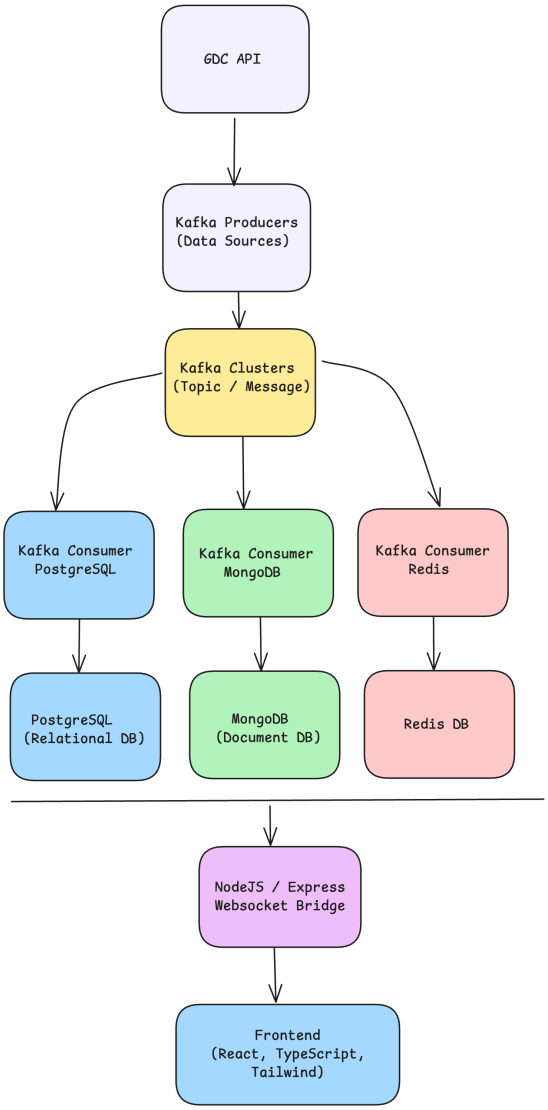

Figure 4: System Architecture Overview

This diagram illustrates the flow of data from the GDC Cancer API through Kafka, to the backend where it populates the databases, and is finally presented on the frontend. The Kafka stream also allows real-time updates to be visible in the frontend.

7.2 Main Functionality

The implementation of the main functionality can be broken down into several key actions.

7.2.1 1. Fetch Data from GDC Cancer API

The first step is fetching data from the GDC Cancer API. This data is fetched asynchronously using Node.js **fetch** calls. The data received includes clinical data, disease types, and other relevant metadata from the API.

```
const casesEndpoint = "https://api.gdc.cancer.gov/cases";


  const parameters = {
    filters: JSON.stringify({
      op: "in",
      content: {
        field: "project.project_id",
        value: [
          "TCGA-BRCA", "TCGA-LUAD", "TCGA-LAML", "TCGA-GBM", "TCGA-KIRC",
          "TCGA-OV", "TCGA-COAD", "TCGA-PRAD", "TCGA-STAD", "TCGA-SKCM"....
        ]
      }
    }),
    size: 100,
    expand: "diagnoses"      // Include diagnosis data inline
  };

  const response = await fetch(casesEndpoint, {
    method: "GET",
    headers: {
      "Content-Type": "application/json",
      Accept: "application/json",
    },
    params: parameters,
  });

  const jsonResponse: any = response.data;
  const clinicalData = jsonResponse.data?.hits || [];
  return clinicalData;
```

7.2.2 2. Produce Data to Kafka

Once the data is fetched, it is serialized and produced to the relevant Kafka topics: clinical data, analytics data, and cache data. Each type of data is produced under a different topic, allowing consumers to read from the correct stream.

```
router.post('/send-clinical-data', async (req: Request, res: Response): Promise<voi
  const { clinicalData }: { clinicalData: ClinicalData } = req.body;

  try {
    await producer.send({
      topic: 'clinical-data',
      messages: [{ value: JSON.stringify(clinicalData) }],
    });
    res.status(200).send('Data sent to Kafka');
  } catch (error) {
    console.error('Error sending data to Kafka:', error);
    res.status(500).send('Error sending data to Kafka');
  }
});
router.post('/send-analytics-data', async (req: Request, res: Response): Promise<vo
  const { analyticsData }: { analyticsData: AnalyticsData } = req.body;

  try {
    await producer.send({
      topic: 'postgres-analytics',
      messages: [{ value: JSON.stringify(analyticsData) }],
    });
    res.status(200).send('Data sent to Kafka');
  } catch (error) {
    console.error('Error sending data to Kafka:', error);
    res.status(500).send('Error sending data to Kafka');
  }
});
router.post('/send-cache-data', async (req: Request, res: Response): Promise<void>
  const { cacheData }: { cacheData: CacheData } = req.body;

  try {
    await producer.send({
      topic: 'redis-cache',
```

```
      messages: [{ value: JSON.stringify(cacheData) }],
    });
    res.status(200).send('Data sent to Kafka');
  } catch (error) {
    console.error('Error sending data to Kafka:', error);
    res.status(500).send('Error sending data to Kafka');
  }
});
```

7.2.3 3. Consume Data and Populate Databases

The backend is responsible for consuming the Kafka streams and populating the databases with the received data. Kafka consumers listen for new messages on each topic and update PostgreSQL, MongoDB, and Redis with the relevant data.

For example, to populate MongoDB with clinical records:

```
const consumeKafka = async () => {
  const consumer = kafka.consumer({ groupId: 'clinical-group' });

  await consumer.connect();
  await consumer.subscribe({ topic: 'clinical-data' });

  await consumer.run({
    eachMessage: async ({ message }) => {
      const clinicalData = JSON.parse(message.value.toString());
      await MongoClient.db('clinical').collection('records').insertOne(clinicalData
    },
  });
};
```

Similar consumers are created for Redis and PostgreSQL, where data is inserted into Redis cache and PostgreSQL analytics tables, respectively.

7.2.4 4. Update Frontend

After the data has been populated in the databases, the frontend is updated to display the new data. The React frontend fetches the latest data from the backend API, which queries the databases.

For example, React components like **MongoClinical**, **PostgresAnalytics**, and **RedisCache** fetch the data and display it in the UI:

```
const MongoClinical = () => {
  const [clinicalData, setClinicalData] = useState([]);

  useEffect(() => {
    fetch('/api/clinical')
      .then(response => response.json())
      .then(data => setClinicalData(data));
  }, []);

  return (
    <div>
      {clinicalData.map(record => (
        <div key={record.id}>{record.name}</div>
      ))}
    </div>
  );
};
```

7.3 Dockerization

To ensure the application can be easily deployed across different environments, the entire project is containerized using Docker. Each component, including the backend, frontend, and databases, is packaged into separate Docker containers. A **docker-compose.yml** file is used to orchestrate the containers.

```
version: '3.8'

services:
  vite-app:
    build: .
    ports:
      - "5173:5173"
    command: ["sh", "-c", "docker ps -q --filter publish=5173 | xargs -r docker sto
    depends_on:
      - express-api
      - kafka
      - ikt453-mongodb
      - ikt453-postgres
      - ikt453-redis-stack

  ikt453-mongodb:
```

```
  image: mongodb/mongodb-community-server:latest
  container_name: ikt453-mongodb
  ports:
    - "27017:27017"
  networks:
    - backend

ikt453-postgres:
  image: postgres:latest
  container_name: ikt453-postgres
  environment:
    POSTGRES_PASSWORD: example
  ports:
    - "5432:5432"
  networks:
    - backend

ikt453-redis-stack:
  image: redis/redis-stack:latest
  container_name: ikt453-redis-stack
  ports:
    - "6379:6379"
    - "8001:8001"
  networks:
    - backend

kafka:
  image: apache/kafka:latest
  container_name: kafka
  environment:
    KAFKA_ADVERTISED_LISTENER: INSIDE_KAFKA:9092
    KAFKA_LISTENER_SECURITY_PROTOCOL: PLAINTEXT
    KAFKA_LISTENER_NAME: INSIDE_KAFKA
    KAFKA_LISTENER_PORT: 9092
    KAFKA_LISTENER_INSIDE: INSIDE_KAFKA
    KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
    KAFKA_LISTENER_MODE: BROKER
  ports:
    - "9092:9092"
  networks:
```

```
    - backend
  depends_on:
    - zookeeper

zookeeper:
  image: wurstmeister/zookeeper:3.5.7
  container_name: zookeeper
  environment:
    ZOOKEEPER_CLIENT_PORT: 2181
  ports:
    - "2181:2181"
  networks:
    - backend

express-api:
  image: node:latest
  container_name: express-api
  working_dir: /usr/src/app
  volumes:
    - ./src/backend:/usr/src/app
  ports:
    - "3001:3001"
  networks:
    - backend
  command: ["sh", "-c", "npm install && ts-node server.ts"]
  depends_on:
    - ikt453-mongodb
    - ikt453-postgres
    - ikt453-redis-stack
    - kafka

websocket:
  image: node:latest
  container_name: websocket
  working_dir: /usr/src/app
  volumes:
    - ./src/backend:/usr/src/app
  ports:
    - "8080:8080"
  networks:
```

```
      - backend
    command: ["sh", "-c", "npm install && ts-node websocket.ts"]
    depends_on:
      - ikt453-mongodb
      - ikt453-postgres
      - ikt453-redis-stack
      - kafka

networks:
  backend:
    driver: bridge
```

This setup allows all components to run in isolation while still being able to communicate with each other. The application can be started with a simple **docker-compose up** command, ensuring that all dependencies are properly initialized.

7.4 Screenshots

The MVP consists of a web interface where users can click on different sections to view the latest clinical data, analytics, and cached results. The frontend provides a user-friendly display of the data, which is constantly updated in real-time through Kafka.

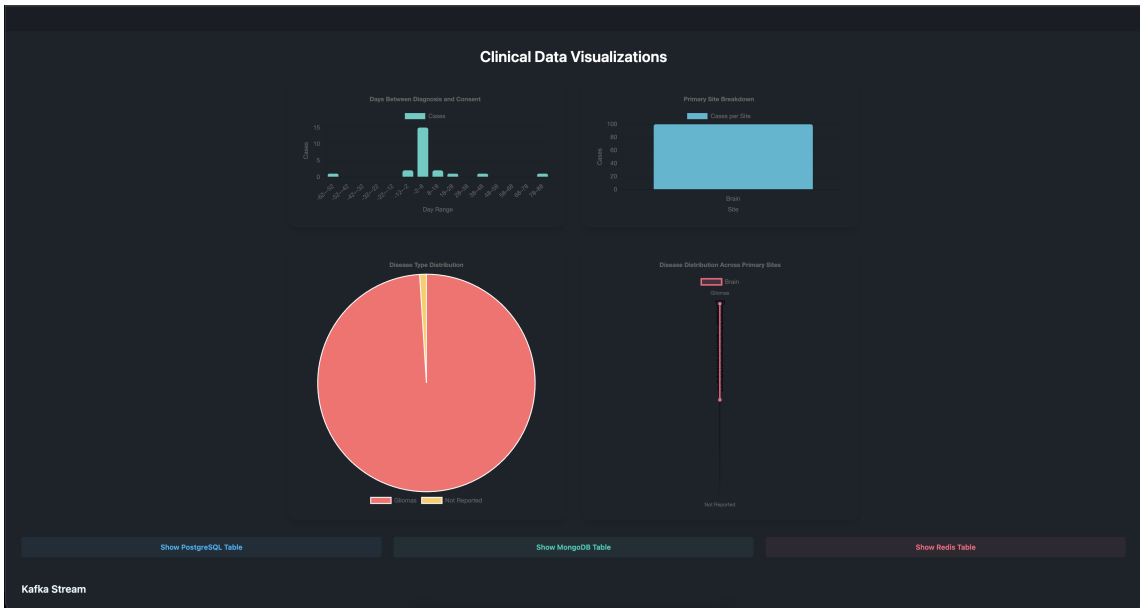

Figure 5

Figure 6

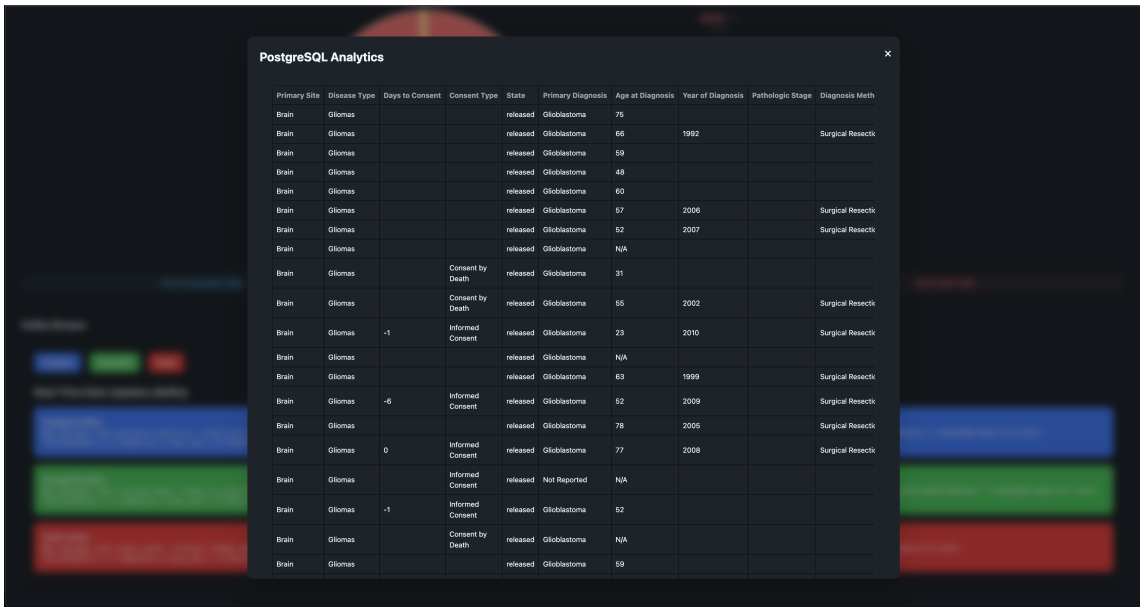

Figure 7

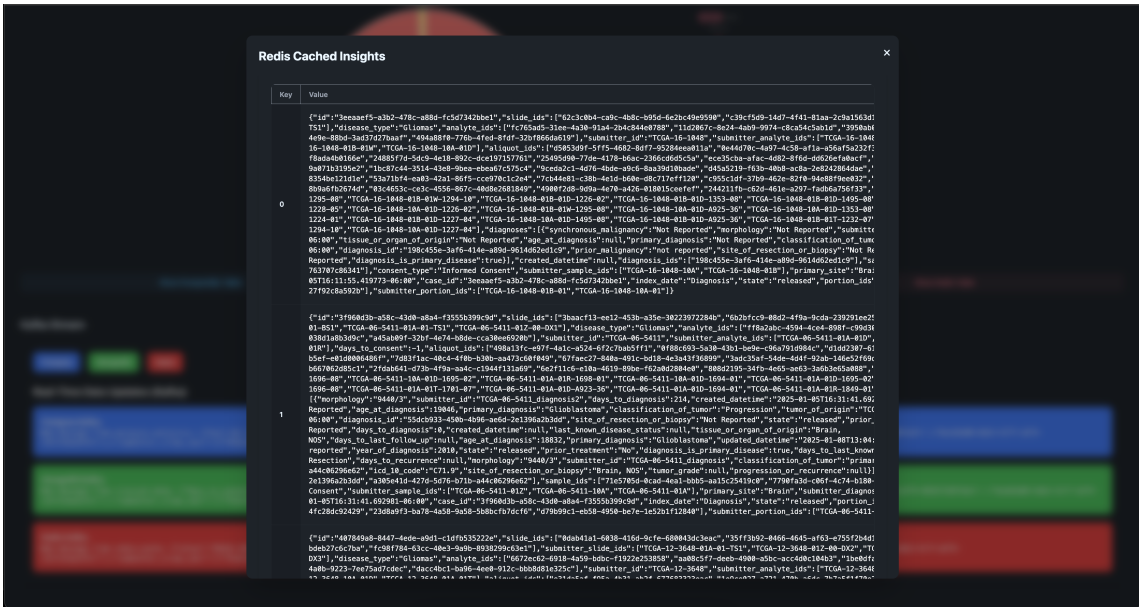

Figure 8

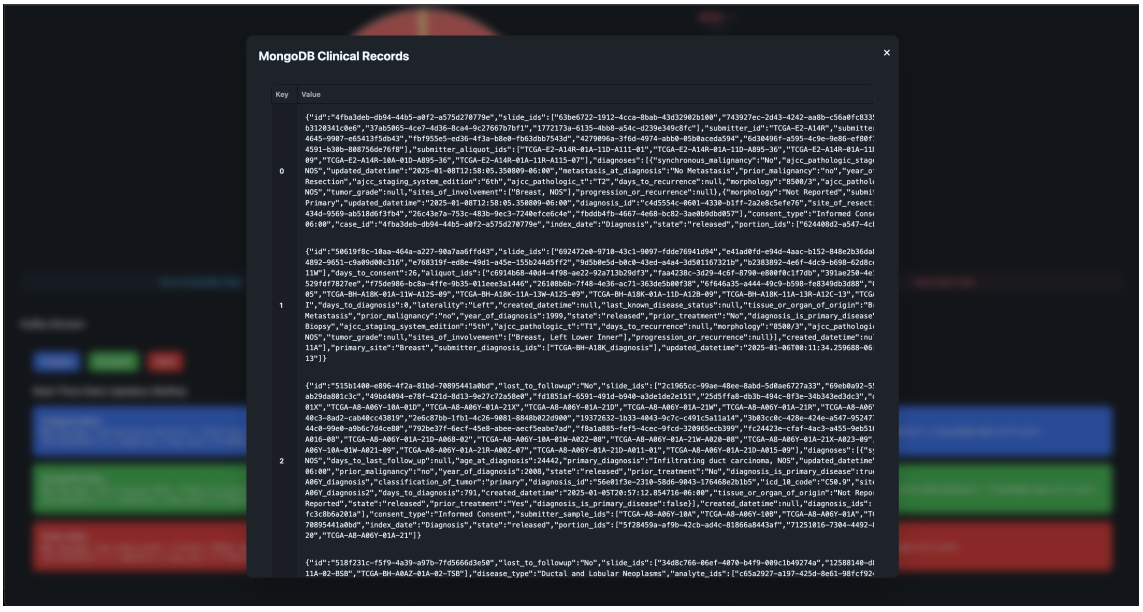

Figure 9

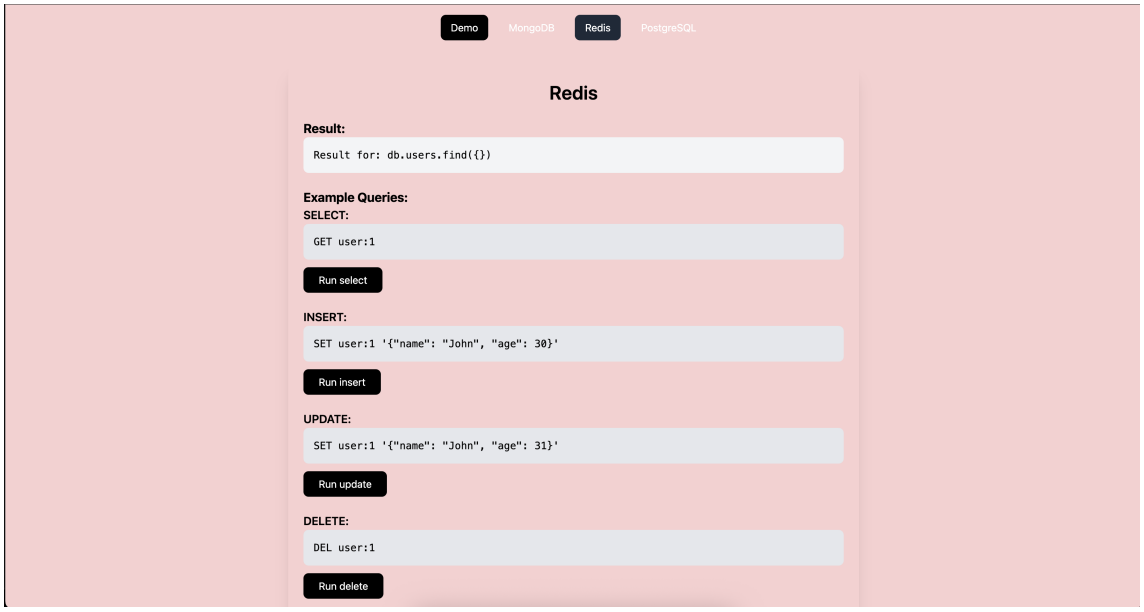

Figure 10

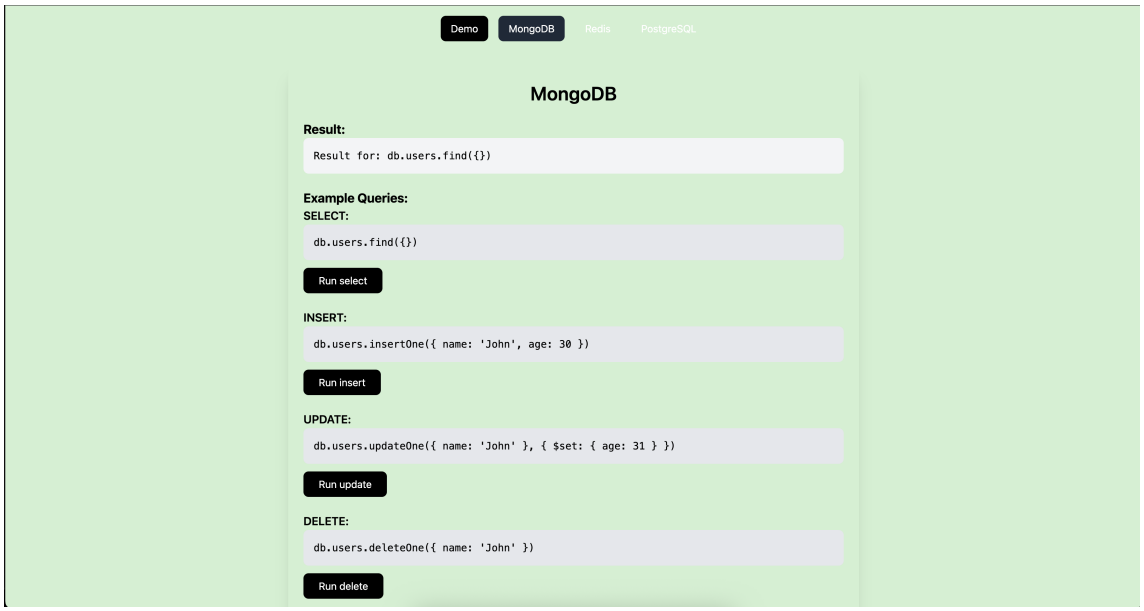

Figure 11


Demo
MongoDB
Redis
PostgreSQL
PostgreSQL
Result:
Result for: SELECT * FROM users;
Example Queries:
SELECT:
SELECT * FROM users;
Run select
INSERT:
INSERT INTO users (name, age) VALUES ('John', 30);
Run insert
UPDATE:
UPDATE users SET age = 31 WHERE name = 'John';
Run update
DELETE:
DELETE FROM users WHERE name = 'John';
Run delete

Figure 12

8 PostgreSQL

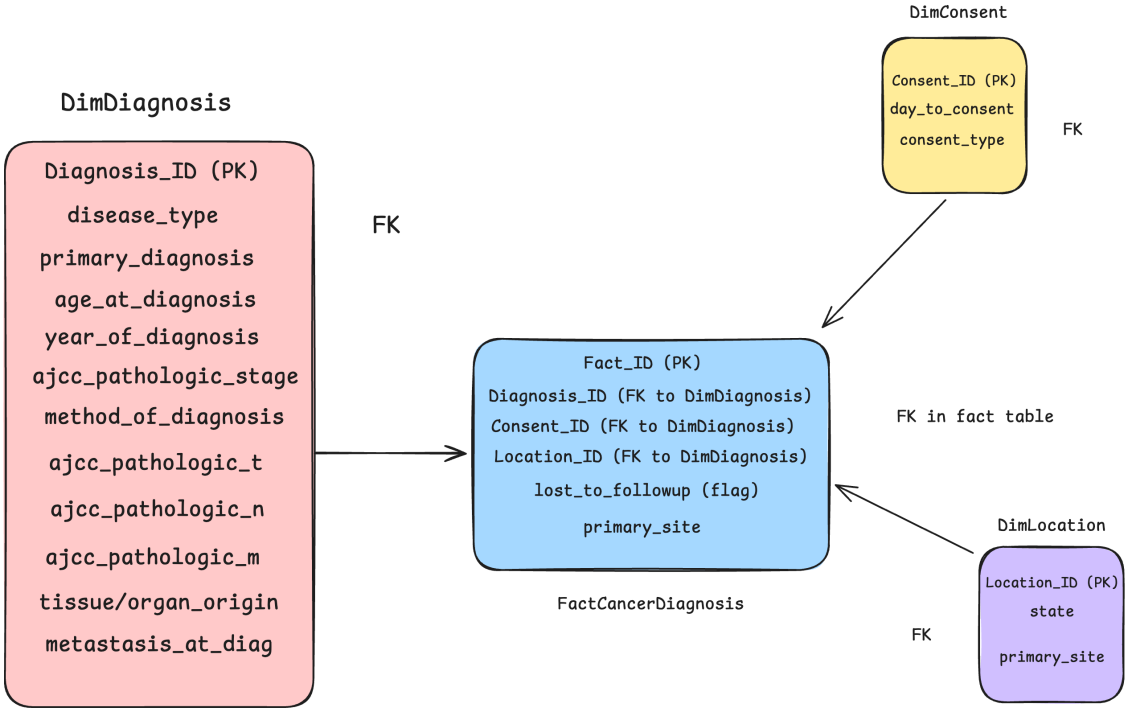

Figure 13

The relational data warehouse implementation is designed using a star schema architecture, which is widely recognized as the industry standard for building efficient, query-optimized data warehouses. In our model, the central fact table: **FactCancerDiagnosis** captures the core events or transactions regarding patient diagnosis data from the GDC cancer portal, and it is surrounded by three primary dimension tables: **DimDiagnosis**, **DimConsent**, and **DimLocation**.

At the heart of the star schema, the **FactCancerDiagnosis** table contains foreign keys linking to **DimDiagnosis, DimConsent, and DimLocation**. This design ensures that each core event integrates highly relevant details including diagnosis specifics, consent information, and location data.
The fact table stores numeric and Boolean flags such as lost_to_followup, alongside identifiers for the associated dimensions. For instance, the Diagnosis_ID stored in the FactCancerDiagnosis table points directly to DimDiagnosis, which contains fields such as disease_type, primary_diagnosis, age_at_diagnosis, and stage-related information (ajcc_pathologic_stage, ajcc_pathologic_t, ajcc_pathologic_n, ajcc_pathologic_m).

These fields are essential for decision support applications, enabling users to query and slice data by various clinical criteria and outcomes.

The DimConsent table is dedicated to capturing consent details, including days_to_consent and consent_type. This separation is crucial because consent information might be subject to specific audit or regulatory review processes. Similarly, the DimLocation table holds geographic details such as primary_site and state, which is important when analyzing regional trends or demographic patterns in disease diagnosis and progression.

This star schema design was selected because it maximizes query performance. In a data warehouse environment, complex analytical queries typically involve aggregating large volumes of data. With the star schema, the fact table is denormalized relative to the operational systems, and the dimensions provide descriptive context without the need for complex joins across many normalized tables. This results in faster execution of typical decision support queries, such as total cases by disease type for a given state or average days to consent by stage of diagnosis.

The design also provides a robust support for scalability and maintainability. Each dimension can be expanded with additional attributes if needed (for example, incorporating more granular geographic metadata or extended diagnosis details) without affecting the integrity of the fact data. Moreover, the separation of concerns - storing transactional facts and descriptive dimensions independently -allows for incremental refreshes or updates in near real-time, especially when dealing with simulated data or streaming updates.

The fact table's simple structure—focusing on numeric and key values—provides high efficiency and minimal disk overhead, while the dimension tables support rich text attributes for filtering and comprehensive data analysis. The star schema also facilitates the use of OLAP tools and business intelligence software that are optimized for such models, ensuring that end-users and decision-makers have the best possible performance when running s̈lice and dicëqueries.

An important assumption in our design is that each event record is assumed to reflect a single consolidated diagnosis scenario; however, if multiple diagnoses were to be logged per patient, a bridge table between the fact and the diagnosis dimension could be introduced. In our current design, we assume that the primary or most relevant diagnosis is captured within the **DimDiagnosis** table, thus simplifying the model.

Overall, the star schema represents the most optimal solution for our data warehouse because of its scalability, simplicity in design, and query performance. It also provides a solid foundation for analytic processing in a relational database system like PostgreSQL. The alternative models: MongoDB and Redis are explored with differ-

ent trade-offs, but for large-scale, structured analytical workloads, the star schema serves as the benchmark for data warehouse design.

9 MongoDB

```
_id: ObjectId("..."),
primary_site: "Lung",
lost_to_followup: false,
state: "CA",
consent: {
days_to_consent: 15,
consent_type: "in-person"
},
diagnosis: {
disease_type: "Carcinoma",
primary_diagnosis: "Adenocarcinoma",
age_at_diagnosis: 65,
year_of_diagnosis: 2021,
ajcc_pathologic_stage: "Stage II",
method_of_diagnosis: "Biopsy",
ajcc_pathologic_t: "T2",
ajcc_pathologic_n: "N1",
ajcc_pathologic_m: "M0",
tissue_or_organ_of_origin: "Lung",
metastasis_at_diagnosis: "No",
diagnoses: [
{ diagnosis_code: "DX1", description: "Additional Detail 1" },
{ diagnosis_code: "DX2", description: "Additional Detail 2" }
]
}
}
```

Figure 14

Our alternative data warehouse implementation leverages MongoDB's document-oriented model. In contrast with the normalized approach of relational databases, the MongoDB model is denormalized, embedding related data within a single document. Our sample document schema is designed to encapsulate all facets of a patient's diagnosis event in one self-contained JSON-like document.

In the MongoDB design, each document represents a comprehensive event that includes fields for primary_site, lost_to_followup, and state, as well as embedded subdocuments for consent and diagnosis details. The consent subdocument includes

fields like days_to_consent and consent_type, while the diagnosis subdocument contains all relevant diagnostic details such as disease_type, primary_diagnosis, age_at_diagnosis, year_of_diagnosis, ajcc_pathologic_stage, method_of_diagnosis, and additional stage partitioning (ajcc_pathologic_t, ajcc_pathologic_n, ajcc_pathologic_m). Furthermore, the model incorporates an array called diagnoses, which is capable of storing multiple diagnosis objects if the event involves more than one diagnosis. Each embedded diagnosis object may include further information such as diagnosis codes and detailed descriptions.

One of the major advantages of this denormalized document approach is the reduced need to perform joins. Since MongoDB is designed to operate in a non-relational schema, all data associated with a diagnosis event is stored in one place. This has multiple benefits: the simplicity of data access increases overall performance for read operations, and the document model is highly suitable for applications where the data structure is expected to evolve over time. MongoDB's flexible schema allows adding new fields to individual documents without impacting the entire collection, which is particularly useful if new types of diagnosis information or consent details need to be incorporated.

The document model is especially appealing in scenarios where real-time analysis of streaming data is required. For instance, when data from receiving Kafka streams is ingested into MongoDB, the application can immediately access a c̈ompletep̈icture of the event without reassembling data from distributed tables. This contributes significantly to performance optimization when queries must be executed rapidly to facilitate real-time decision making.

Moreover, MongoDB's indexing features (such as compound indexes on embedded fields) can enhance performance further, allowing for efficient execution of complex queries that filter on attributes across nested fields. For example, a compound index on both diagnosis.ajcc_pathologic_stage and state can allow for fast filtering of cases within specific geographic areas and with particular staging details.

Our alternative model also accommodates data transformation use cases seamlessly. When integrating data from the portal or simulated sources, transformation logic can be applied to produce a canonical document format that integrates all dimensions (diagnosis, consent, and location data) in a single insert operation. This reduces overhead on write operations and simplifies the application logic. The data model's flexibility is ideal for iterative development, where new data attributes might be introduced during exploratory analysis or decision support query definition.

Importantly, while the document model offers significant performance benefits on read-heavy workloads, it also presents a trade-off with respect to data redundancy. There is an inherent risk of duplicate storage when the same consent or location

information is embedded across documents. However, in many real-time analytics scenarios, the trade-off is justified by the gains in query performance and simplicity in development. In practice, data warehouses based on MongoDB often incorporate periodic aggregation processes to clean or consolidate redundant fields, ensuring that the overall system remains optimized for query performance.

The MongoDB document schema offers a strong alternative to the relational star schema, particularly in situations where the flexibility of the schema, real-time analytics, and rapid read access are paramount. It represents a robust model for decision support that is particularly well-suited for diverse data streams and evolving data structures in modern application domains.

10 Redis

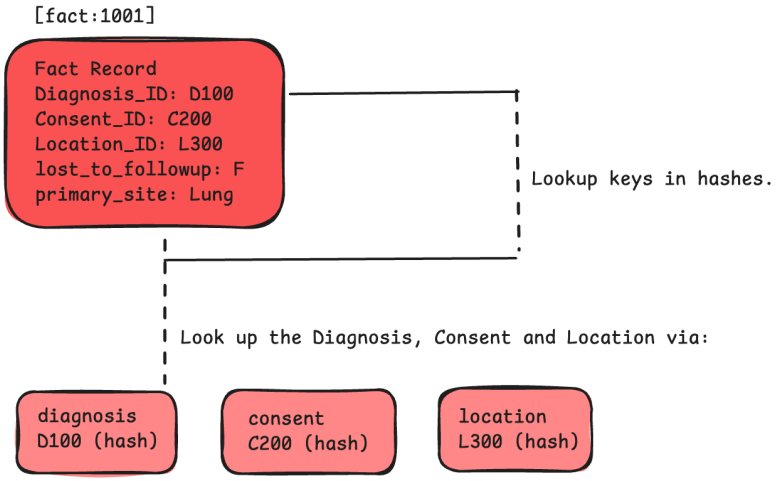

Figure 15

The third approach in our data warehouse design leverages the in-memory capabilities of Redis by deploying a key–value/Hash model architecture. Redis is traditionally known as a caching and real-time processing tool, but with its support for advanced data types such as hashes, sorted sets, and lists, it can also act as an effective NoSQL data store for analytical workloads requiring low-latency responses.

In our Redis design, we partition the data into separate keys for the fact records and corresponding dimension records. For example, a fact record is stored in a Redis hash with a key such as f̈act:1001; where the fields include Diagnosis_ID, Consent_ID, Location_ID, lost_to_followup, and primary_site. The individual components—diagnosis, consent, and location—are similarly stored in their respec-

tive hashes (e.g., diagnosis:D100, consent:C200, location:L300). This modular approach mimics the relational star schema, but instead relies on the fast, in-memory operations unique to Redis.

One of Redis's primary strengths is its performance; data is stored in memory, enabling sub-millisecond access to query results. This is ideal for decision support scenarios where real-time analytics is critical and the cost of a cache miss in a relational system would be too high. Redis supports atomic operations, meaning that even when multiple clients access a fact record concurrently, data consistency is maintained—a highly desirable characteristic in environments where Kafka streams trigger frequent updates.

Additionally, Redis allows us to create secondary indexes using its sorted sets or sets. For example, by assigning the diagnosis ID to a sorted set keyed by a relevant attribute (such as disease_type), we can quickly retrieve all diagnosis records for a particular category. This auxiliary indexing mechanism significantly enhances data retrieval operations, as it permits targeted queries without scanning the entire dataset. This design decision is crucial in extending the simple key–value operations into robust decision support queries, such as find all records for carcinoma cases in California or retrieve all records with a specific AJCC stage.

The modularity of the Redis model is particularly beneficial when integrating with other real-time components in the system. For instance, our KafkaStream component is designed to publish new events that are then immediately ingested into Redis. This setup leverages Redis's capacity as both a transient store and a fast analytics engine, where computed aggregates or recent events are cached and made available to front-end dashboards instantly. In this way, Redis acts as a vital piece in an architecture that supports instant insight and iterative data updates.

One challenge with this model is that of persistence and data size management. Since Redis is primarily in-memory, careful configurations such as AOF (Append-Only File) or RDB (Redis Database Backup) snapshots must be in place to ensure data durability in the event of system reboots or failures. With these mechanisms in place, the trade-off between high-speed access and data persistence can be managed effectively.

The Redis key–value model also provides flexibility in terms of schema evolution. Unlike traditional databases, Redis does not enforce a strict schema, allowing for rapid adjustment as new diagnostic fields or consent fields become necessary. Developers can simply add new fields to existing hashes or create new keys without altering predefined schemas, which is extremely favorable during the exploratory phase of a data warehousing project where requirements may shift frequently.

The Redis-based model, through its use of keys and hashes, represents a highly

optimized alternative for scenarios where query speed and real-time analytics take precedence over deep relational integrity. It offers a complementary approach to the relational star schema and MongoDB document model. While it may not provide the same long-term, disk-based persistence out-of-the-box as a relational database, its configuration for persistence along with its unmatched speed and scalability make it a viable and optimal solution for a tiered data warehouse architecture. By implementing secondary indexing and careful key management, the Redis model supports complex decision support queries, ensuring that all three models—relational, document, and key–value—are well-positioned to meet the varying needs of data warehouse operations in our project.

These detailed descriptions provide a comprehensive overview of each data model and explain the reasoning behind our choice of the relational star schema as the primary optimal solution, with MongoDB and Redis serving as valuable alternative implementations for specific performance or flexibility requirements.

1. PostgreSQL – Relational Model DDL and Sample Data (DML)

DDL: Create Tables

```
-- Dimension Table for Diagnosis
CREATE TABLE DimDiagnosis (
  Diagnosis_ID SERIAL PRIMARY KEY,
  disease_type VARCHAR(255),
  primary_diagnosis VARCHAR(255),
  age_at_diagnosis INTEGER,
  year_of_diagnosis INTEGER,
  ajcc_pathologic_stage VARCHAR(50),
  method_of_diagnosis VARCHAR(255),
  ajcc_pathologic_t VARCHAR(50),
  ajcc_pathologic_n VARCHAR(50),
  ajcc_pathologic_m VARCHAR(50),
  tissue_or_organ_of_origin VARCHAR(255),
  metastasis_at_diagnosis VARCHAR(50)
);

-- Dimension Table for Consent
CREATE TABLE DimConsent (
  Consent_ID SERIAL PRIMARY KEY,
  days_to_consent INTEGER,
```

```
  consent_type VARCHAR(255)
);

-- Dimension Table for Location
CREATE TABLE DimLocation (
  Location_ID SERIAL PRIMARY KEY,
  state VARCHAR(50),
  primary_site VARCHAR(255)
);

-- Fact Table for Cancer Diagnosis
CREATE TABLE FactCancerDiagnosis (
  Fact_ID SERIAL PRIMARY KEY,
  Diagnosis_ID INTEGER REFERENCES DimDiagnosis(Diagnosis_ID),
  Consent_ID INTEGER REFERENCES DimConsent(Consent_ID),
  Location_ID INTEGER REFERENCES DimLocation(Location_ID),
  lost_to_followup BOOLEAN,
  primary_site VARCHAR(255)
);
```

DML: Insert Sample Data

DimDiagnosis

```
INSERT INTO DimDiagnosis (
  disease_type, primary_diagnosis, age_at_diagnosis, year_of_diagnosis,
  ajcc_pathologic_stage, method_of_diagnosis, ajcc_pathologic_t,
  ajcc_pathologic_n, ajcc_pathologic_m, tissue_or_organ_of_origin,
  metastasis_at_diagnosis
) VALUES
  ('Carcinoma', 'Lung Cancer', 65, 2020, 'Stage III', 'Biopsy', 'T2', 'N1', 'M0', '
  ('Sarcoma', 'Bone Cancer', 45, 2021, 'Stage II', 'Imaging', 'T1', 'N0', 'M0', 'Bo
```

DimConsent

```
INSERT INTO DimConsent (days_to_consent, consent_type) VALUES
  (10, 'Informed'),
  (15, 'Explicit');
```

DimLocation

```
INSERT INTO DimLocation (state, primary_site) VALUES
  ('NY', 'New York Medical Center'),
  ('CA', 'California Oncology Center');
```

FactCancerDiagnosis

```
-- Assuming IDs:
-- Diagnosis_ID 1, 2
-- Consent_ID 1, 2
-- Location_ID 1, 2

INSERT INTO FactCancerDiagnosis (
  Diagnosis_ID, Consent_ID, Location_ID, lost_to_followup, primary_site
) VALUES
  (1, 1, 1, FALSE, 'New York Medical Center'),
  (2, 2, 2, TRUE, 'California Oncology Center');
```

2. MongoDB – NoSQL DDL and Sample Data

Database and Collection Setup

```
// Use or create "cancerDW"
use cancerDW;

// Create the collection
db.createCollection("patientDiagnosis");

// Create indexes
db.patientDiagnosis.createIndex({ "diagnosis.ajcc_pathologic_stage": 1 });
db.patientDiagnosis.createIndex({ "location.state": 1 });
db.patientDiagnosis.createIndex({ "consent.consent_type": 1 });
```

Insert Sample Document

```
db.patientDiagnosis.insertOne({
  patient_id: "P001",
  diagnosis: {
    disease_type: "Carcinoma",
    primary_diagnosis: "Lung Cancer",
    age_at_diagnosis: 65,
    year_of_diagnosis: 2020,
    ajcc_pathologic_stage: "Stage III",
    method_of_diagnosis: "Biopsy",
    ajcc_pathologic_t: "T2",
    ajcc_pathologic_n: "N1",
```

```
    ajcc_pathologic_m: "M0",
    tissue_or_organ_of_origin: "Lung",
    metastasis_at_diagnosis: "No"
  },
  consent: {
    days_to_consent: 10,
    consent_type: "Informed"
  },
  location: {
    state: "NY",
    primary_site: "New York Medical Center"
  },
  lost_to_followup: false
});
```

3. Redis – Key-Value Model Sample Data

Sample Data Insertion in Redis

```
-- Diagnosis Record
HSET diagnosis:1 \
  disease_type "Carcinoma" \
  primary_diagnosis "Lung Cancer" \
  age_at_diagnosis 65 \
  year_of_diagnosis 2020 \
  ajcc_pathologic_stage "Stage III" \
  method_of_diagnosis "Biopsy" \
  ajcc_pathologic_t "T2" \
  ajcc_pathologic_n "N1" \
  ajcc_pathologic_m "M0" \
  tissue_or_organ_of_origin "Lung" \
  metastasis_at_diagnosis "No"

-- Consent Record
HSET consent:1 \
  days_to_consent 10 \
  consent_type "Informed"

-- Location Record
HSET location:1 \
```

```
  state "NY" \
  primary_site "New York Medical Center"

-- Fact Record
HSET fact:P001 \
  diagnosis_id 1 \
  consent_id 1 \
  location_id 1 \
  lost_to_followup 0 \
  primary_site "New York Medical Center"
```

11 Pre-Aggregated Summary Tables

This section outlines the creation and population of pre-aggregated summary tables in the data warehouse. These tables are designed to improve query performance for common analytical queries by storing aggregated data. We also describe a batch job to automate the loading of data into these tables.

11.1 Summary Table for Diagnosis Counts by Disease Type and Stage

11.1.1 DDL Statement

```
CREATE TABLE SummaryDiagnosisCounts (
  disease_type VARCHAR(255),
  ajcc_pathologic_stage VARCHAR(50),
  total_cases INTEGER,
  PRIMARY KEY (disease_type, ajcc_pathologic_stage)
);
```

11.1.2 SQL to Populate the Summary Table

```
INSERT INTO SummaryDiagnosisCounts (disease_type, ajcc_pathologic_stage, total_case
SELECT
  d.disease_type,
  d.ajcc_pathologic_stage,
  COUNT(*) AS total_cases
FROM
  FactCancerDiagnosis f
JOIN
  DimDiagnosis d ON f.Diagnosis_ID = d.Diagnosis_ID
GROUP BY
```

```
  d.disease_type, d.ajcc_pathologic_stage;
```

11.2 Summary Table for Consent Types by State

11.2.1 DDL Statement

```
CREATE TABLE SummaryConsentByState (
  state VARCHAR(50),
  consent_type VARCHAR(255),
  total_consent INTEGER,
  PRIMARY KEY (state, consent_type)
);
```

11.2.2 SQL to Populate the Summary Table

```
INSERT INTO SummaryConsentByState (state, consent_type, total_consent)
SELECT
  l.state,
  c.consent_type,
  COUNT(*) AS total_consent
FROM
  FactCancerDiagnosis f
JOIN
  DimConsent c ON f.Consent_ID = c.Consent_ID
JOIN
  DimLocation l ON f.Location_ID = l.Location_ID
GROUP BY
  l.state, c.consent_type;
```

11.3 Batch Job Specification

To automate the process of loading data into these pre-aggregated summary tables, a batch job can be set up using a scheduling tool like `cron` on Unix-based systems. Below is a conceptual outline of the batch job setup:

11.3.1 Script Creation

Create a SQL script file (e.g., `populate_summary_tables.sql`) containing the SQL statements to populate the summary tables.

11.3.2 Batch Script

Create a shell script (e.g., `update_summaries.sh`) that connects to the PostgreSQL database and executes the SQL script.

```
#!/bin/bash
PGPASSWORD='your_password' psql -U your_username -d your_database -h your_host -f p
```

11.3.3 Scheduling the Job

Use `cron` to schedule the batch script to run at regular intervals (e.g., nightly).

```
# Edit the crontab file
crontab -e

# Add the following line to run the script every day at 2 AM
0 2 * * * /path/to/update_summaries.sh
```

This setup ensures that the summary tables are regularly updated with the latest aggregated data, providing fast access to pre-computed results for common queries.

12 Discussion

This project serves as a minimum viable product (MVP) for a comprehensive data warehouse and research platform where users can perform analytics on clinical data derived from the GDC cancer database. Developed by a pair of developers working in a pair programming environment, the project not only served as an effective learning experience but also as a proof-of-concept demonstrating the integration of diverse technologies. Our tech stack includes PostgreSQL, MongoDB, Redis, Kafka, Docker, ReactJS, and TypeScript - all orchestrated to provide a cohesive system for data ingestion, storage, and retrieval.

12.1 Product

The final product is a web-based application that allows users to select, update, delete, and insert records across multiple databases. Kafka plays a dual role in both ingesting data from the GDC API and propagating changes in real time across the system. While the application currently addresses a specific problem—a proof-of-concept for clinical data analytics—the underlying architecture is scalable and can be extended to handle more complex analytical queries, enhanced reporting, and additional data sources.

12.2 Implementation

Our implementation leveraged Docker to containerize every component of the system, including the ReactJS frontend, TypeScript backend, and the various databases. This containerization facilitated rapid development and testing while ensuring that the environment remained consistent across different stages of development. The relational schema in PostgreSQL was designed using a star-schema approach, complemented by MongoDB's flexibility for handling unstructured data and Redis for caching critical queries. Kafka was integrated to ensure real-time data streaming, though synchronizing the data flows between Kafka and the databases—especially under high load—required iterative tuning and improved error handling.

12.3 Challenges

Throughout the project, we encountered several key challenges:

- **Kafka Integration:** Ensuring smooth data streaming between Kafka and the databases introduced complexities, especially under higher data loads. Although we mitigated issues with buffer management and error handling, further optimization is needed for large-scale deployment.

- **Database Performance:** Certain complex queries in PostgreSQL, particularly those involving aggregations and joins, experienced performance lags. Exploring advanced indexing and query optimization remains a priority for future improvements.
- **GDC API Limitations:** The limitations in responsiveness and data range provided by the GDC API occasionally restricted the diversity and completeness of our clinical data.
- **Frontend Responsiveness:** While the web application delivers a solid desktop experience, the UI requires enhancements to be fully responsive on mobile devices.

12.4 Future Work

Despite achieving a functioning MVP, the project lays the groundwork for far greater possibilities:

- **Scalability and Performance:** Future efforts should focus on optimizing Kafka for higher throughput and fine-tuning database queries, especially in PostgreSQL, to better handle increased data volumes.
- **Enhanced Mobile Experience:** Optimizing the frontend design for mobile responsiveness will be essential in broadening accessibility and improving user engagement.
- **Expanded Data Integration:** Integrating additional data sources and enriching the clinical dataset can significantly enhance the platform's analytical capabilities.
- **Advanced UX Improvements:** Further development of user interface features such as authentication, filtering options, and improved data visualizations will elevate the overall user experience.

This project not only demonstrates our ability to integrate multiple cutting-edge technologies into a unified solution but also highlights the potential for future development into a full-scale, analytics-driven research platform. The hands-on experience with PostgreSQL, MongoDB, Redis, Kafka, Docker, ReactJS, and TypeScript, combined with an agile pair programming approach, provided valuable insights and laid a solid foundation for subsequent enhancements and scalability improvements.

13 Conclusion

In this project, the problem at hand was to create a system that could integrate and display clinical data from the GDC Cancer API while using Kafka for real-time data streaming, along with PostgreSQL, MongoDB, and Redis for data storage and caching. The goal was to showcase how these technologies could work together to provide real-time insights into cancer data.

Our solution utilized Kafka as the core component for handling data streams, enabling the production and consumption of messages to update databases and provide real-time updates to the frontend. PostgreSQL, MongoDB, and Redis were used effectively to store clinical data, analytics, and cache frequently accessed data, respectively. The frontend, built with ReactJS and styled with TailwindCSS, offered an intuitive interface to view the data, while Docker was used to containerize the entire system, ensuring easy deployment and scalability.

The result of our work is a fully functional application that fetches data from the GDC Cancer API, processes it using Kafka, stores the relevant information in PostgreSQL, MongoDB, and Redis, and displays it in an interactive and user-friendly manner on the frontend. The system not only provides the required data but also allows users to see the data updates in real time, making it a powerful tool for healthcare professionals, researchers, and others working in the medical field.

Compared to existing solutions, our system stands out due to the integration of Kafka for real-time data streaming and the use of multiple databases for different types of data storage, making it highly scalable and efficient. The use of Docker containers also allows for seamless deployment and easy environment setup.

The benefits of our solution are clear: users can access clinical data instantly and receive real-time updates, which can significantly improve the workflow of healthcare professionals and researchers. Furthermore, the use of Redis as a cache improves performance, making the application faster and more responsive.

As for the next steps, future work could involve extending the system to handle more complex datasets and incorporating machine learning models to analyze the data. Additionally, we could improve the frontend by adding more detailed charts, making the data more accessible and actionable for users. Finally, implementing a more robust authentication and authorization system would be a valuable feature, especially for users dealing with sensitive clinical data.

Overall, the project successfully addresses the problem at hand, and the solution we have developed can serve as a solid foundation for future enhancements and applications in the healthcare and research domains.

References

[1] *Apache Kafka*. https://kafka.apache.org/.

[2] *Docker*. https://www.docker.com/.

[3] *Express.js*. https://expressjs.com/.

[4] *MongoDB*. https://www.mongodb.com/.

[5] *Node.js*. https://nodejs.org/en.

[6] *PostgreSQL*. https://www.postgresql.org/.

[7] *React*. https://react.dev/.

[8] *Redis*. https://redis.io/.

[9] *Tailwind CSS*. https://tailwindcss.com/.

[10] *TypeScript*. https://www.typescriptlang.org/.