VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

Database Management Systems CO3021 – Semester 242

BIG VELOCITY MANAGEMENT

Team Assignment Report

Group:	Group 07	
Students:	Khưu Vĩnh Kiên	2211720
	Hoàng Hải Phương	2152250
	Bùi Thái Bảo	2252060
	Phan Quang Nhân	2053286

HO CHI MINH CITY, APRIL 2025

CONTENTS

CONTRIBUTION TABLE

Name	Student ID	Contribution	Percentage
Khưu Vĩnh Kiên	2211720	Designed and implemented pipeline architecture	100%
Hoàng Hải Phương	2152250	Managed data workflows and writing report	100%
Bùi Thái Bảo	2252060	Data searching and Managed data workflows	100%
Phan Quang Nhân	2053286	Data preprocessing and implemented pipeline architecture, writing report	100%

1. INTRODUCTION

1.1. What is Big Data Velocity?

Big Data Velocity refers to the speed at which data is generated, collected, and processed in real-time or near real-time environments. With the increasing digitization of every aspect of modern life—from health monitoring devices to social media, financial transactions, and sensor networks—the volume of data being created continues to grow exponentially and arrive at unprecedented speed. This surge in data speed presents unique challenges and opportunities for organizations aiming to gain timely insights and make informed decisions.

Unlike traditional batch processing, which handles data at rest, velocity-centric big data systems must process data in motion—meaning they must handle input streams in real time. As such, velocity is a critical component in scenarios where delay can lead to missed opportunities or even catastrophic consequences, such as fraud detection, healthcare alerts, and dynamic pricing systems.

Managing big data velocity requires specialized systems capable of ingesting, analyzing, and storing fast-moving data while ensuring low latency and high reliability. Technologies like Apache Kafka, Redis Streams, and Apache Flink have emerged to support such high-throughput environments.

In this report, we investigate the concept of Big Data Velocity through the lens of a real-world use case in the healthcare domain. We propose a Redis-based DBMS architecture to manage the rapid ingestion and real-time processing of streaming healthcare data. This approach demonstrates the significance and practical benefits of adopting velocity-oriented data solutions in modern applications.

1.1.2. Characteristics of Big Data Velocity

*a) **High Throughput***

High data velocity involves the continuous flow of data from multiple sources such as IoT sensors, online transactions, social media, or healthcare monitoring systems. Systems must handle large volumes of incoming data per second, demanding high-throughput infrastructure.

b) ***Low Latency***

Minimizing delay is critical when data needs to be acted upon immediately. For example, in healthcare systems, detecting abnormal patient vitals in real-time can prevent emergencies. Technologies like Redis or Kafka enable low-latency data transmission and processing.

c) ***Real-Time Decision Making***

Velocity necessitates on-the-fly analytics. Instead of storing data for later analysis, systems must be capable of drawing insights and triggering actions instantly—such as fraud detection systems or dynamic traffic signal control.

d) ***Stream Processing over Batch Processing***

In contrast to batch systems that analyze data in chunks after collection, velocity systems process data in streams. Stream processing engines like Apache Flink, Apache Storm, or Spark Streaming are used to enable continuous computation on incoming data.

e) ***Temporal Relevance***

Data loses its value rapidly in high-velocity environments. For example, stock prices or health sensor data are only useful if processed and acted upon immediately. Velocity-based systems must prioritize timely processing to retain value**.**

f) Data Volatility

High-velocity data is often transient. Not all data is stored permanently—instead, it might be analyzed in memory and discarded. This transient nature demands fast, in-memory processing solutions like Redis or in-flight processing models.

These characteristics require systems that are scalable, fault-tolerant, and optimized for real-time stream processing. Choosing appropriate DBMS technologies such as Redis for temporary, fast-access data stores is critical for handling the velocity component effectively.

1.1.3. Data Types of Big Data Variety

In high-velocity environments, data arrives rapidly from diverse sources and in multiple formats. To handle such inflows efficiently, it's important to categorize the data based on its structure and format. High-velocity data is generally divided into three types: real-time structured data, streaming semi-structured data, and high-frequency unstructured data.

1.1.3.1. Structured Data

Real-time structured data refers to highly organized data that follows a fixed schema and can be quickly processed by traditional database systems. This type of data often originates from systems like transactional databases, point-of-sale terminals, or monitoring equipment.

Examples:

- Stock exchange feeds with real-time price updates.
- Banking systems processing thousands of transactions per second.
- Medical devices continuously sending vital signs to hospital servers.

Because of its well-defined format, structured data is ideal for real-time analytics and dashboards, where immediate decisions are required based on incoming metrics.

1.1.3.2. Streaming Semi-Structured Data

Streaming semi-structured data includes information that is not organized in a strict tabular format but still contains tags or markers that define data fields. It is widely used in messaging systems, APIs, and IoT ecosystems.

Examples:

- JSON or XML messages in web services or APIs.
- Event logs from applications or operating systems.
- Sensor outputs from smart devices using MQTT or Kafka streams.

Processing semi-structured data in real-time often requires schema inference and flexible parsing mechanisms, such as those found in NoSQL databases or stream processing platforms like Apache Kafka and Apache Flink.

1.1.3.3. High-Frequency Unstructured Data

Unstructured data lacks a fixed schema and arrives in raw formats, making it the most complex to process at high velocity. Yet, it's also the most abundant in today's digital systems.

Examples:

- Social media feeds (text, images, and video).
- Live audio/video streams from surveillance or conferencing systems.
- Real-time customer support chat logs or voice transcripts.

Real-time unstructured data processing often involves advanced techniques such as natural language processing (NLP), image recognition, or machine learning pipelines. Systems must be designed to filter, analyze, and act upon such data as it arrives.

1.1.4. Challenges of Variety Management

1.1.4.1. Real-Time Sources

High-velocity data originates from real-time sources such as IoT devices, social media platforms, mobile apps, and transactional systems. These sources produce continuous streams of data with minimal buffering time.

Challenges:

- **Unpredictable Volume:** Data spikes can overwhelm ingestion pipelines.
- **Network Latency:** Delays in transmitting data can reduce timeliness and accuracy.
- **Source Reliability:** Intermittent failures or inconsistencies from sensors or external APIs can affect downstream systems.

Effective real-time processing requires fault-tolerant ingestion mechanisms, low-latency transport protocols (e.g., Kafka, MQTT), and resilient network design.

1.1.4.2. Stream Integration

Combining multiple real-time data streams into a cohesive pipeline is complex, especially when dealing with asynchronous sources, varying formats, and inconsistent timestamps.

Challenges:

- **Schema Drift:** Real-time streams may evolve over time, requiring dynamic adaptation.
- **Time Synchronization:** Aligning data from different time zones or clock sources introduces challenges in ordering and consistency.
- **Backpressure Handling:** Systems must adapt to fluctuations in input rate without dropping data.

Stream processing frameworks like Apache Flink, Apache Kafka Streams, and Spark Streaming are used to address these issues, but require careful design to maintain throughput and consistency.

1.1.4.3. Data Quality in Real-Time

Ensuring high data quality in a real-time environment is especially difficult due to the speed of data arrival and limited processing time.

Challenges:

- **Incomplete or Noisy Data:** Real-time data often includes missing fields, typos, or erroneous values.
- **No Time for Full Validation:** Unlike batch processing, there's little time for comprehensive cleaning or enrichment.
- **Duplicate or Out-of-Order Events:** Streamed data may arrive in inconsistent order or be repeated due to retries.

Real-time data quality solutions include on-the-fly validation, approximate cleansing techniques, and event time processing logic.

1.1.4.3. Scalability

As real-time data flows increase in speed and volume, systems must scale without compromising latency or accuracy.

Challenges:

- **Compute Resource Limits:** Systems must process data faster than it arrives, often requiring horizontal scaling.
- **State Management:** Real-time applications often maintain in-memory states (e.g., rolling averages), which are hard to scale across nodes.
- **Cost Efficiency:** Scaling real-time infrastructure—especially with low latency and high availability—can be expensive.

Solutions involve auto-scaling architectures, microservices design, and leveraging in-memory datastores like Redis or distributed streaming engines like Apache Pulsar.

1.2. Role of Variety in Big Data

1.2.1. How is Variety Crucial with Big Data?

Velocity is a fundamental pillar in the big data ecosystem, especially in domains where time-sensitive insights can drive critical decision-making. The speed at which data is processed determines how promptly actions can be taken, making velocity essential in real-time applications such as healthcare monitoring, fraud detection, stock trading, and smart cities.

In many scenarios, the value of data diminishes with time. For instance, a delayed alert in a healthcare system could be life-threatening. Real-time data handling ensures that anomalies, trends, or patterns are identified and addressed without latency. Therefore, businesses and institutions that rely on live feedback loops or streaming data must prioritize velocity to maintain competitiveness and efficiency.

Moreover, high-velocity data environments allow organizations to shift from reactive to proactive operations. Rather than analyzing events after they occur, systems powered by real-time analytics can predict and prevent issues ahead of time, thereby optimizing performance and improving service delivery.

1.2.2. How is Variety Used in Big Data?

Big data velocity is applied across various sectors to enable continuous processing and rapid response to new information. Key applications include:

a) Real-Time Monitoring: Healthcare systems use real-time monitoring to track patient vitals continuously. Any abnormalities—such as irregular heart rate or oxygen levels—trigger immediate alerts to healthcare providers.

b) Streaming Analytics: Companies leverage streaming analytics to evaluate clickstream data from websites or mobile apps. This allows for real-time personalization and targeted marketing campaigns.

c) Financial Transactions: Banks and payment systems process thousands of transactions per second. Real-time fraud detection systems analyze transaction patterns instantly to flag suspicious activities.

d) IoT and Sensor Networks: Smart devices generate high-velocity data streams from sensors embedded in cars, homes, or industrial machines. This data is used for predictive maintenance, energy optimization, and real-time logistics tracking.

e) Social Media Sentiment Analysis: Velocity allows for real-time tracking of public sentiment across platforms like Twitter or Facebook. This is valuable for brand management, political campaigns, and customer service.

These use cases illustrate how big data velocity transforms static data systems into dynamic, responsive platforms that enhance decision-making, reduce risk, and improve user experiences across industries.

1.3. Redis Introduction

In today's real-time data landscape, organizations increasingly require systems capable of ingesting, processing, and responding to data with minimal latency. Redis (Remote

Dictionary Server), an open-source, in-memory data structure store, is purpose-built for such high-velocity scenarios. Redis bridges the gap between traditional databases—which may struggle with speed and concurrency—and modern demands for low-latency, high-throughput processing.

Redis combines the flexibility of key-value data storage with the performance of in-memory computing, allowing for real-time data processing across a wide range of use cases, from caching and message queuing to real-time analytics and machine learning pipelines.

Key features of Redis include:

- In-memory speed for rapid reads and writes.
- Support for multiple data structures such as strings, hashes, lists, sets, sorted sets, streams, and geospatial indexes.
- Pub/Sub and Streams for real-time messaging.
- High availability with Redis Sentinel and scalability via Redis Cluster.

Redis addresses many of the limitations found in traditional database systems when dealing with real-time applications:

- **Appending data at speed:** Redis allows seamless data writes with minimal latency, supporting high-frequency event ingestion.
- **Efficient updates and deletions:** Its data structures enable instant modifications without heavy I/O operations.
- **Real-time operations made easy:** Redis natively supports streaming data through Pub/Sub and Redis Streams, ideal for IoT, logging, and monitoring systems.
- **Low-latency querying:** As an in-memory store, Redis allows instant access to frequently queried data.
- **Handling ephemeral and time-sensitive data:** Redis supports TTL (Time-to-Live) for automatic expiration, useful for session storage, caching, or alerting systems.
- **Minimal storage overhead:** Redis is optimized for short-lived or frequently updated data, making it cost-effective for many real-time applications.

1.3.1. Core Features of Redis for High-Velocity Systems

Redis incorporates several core architectural features that make it highly suitable for real-time processing environments:

In-Memory Performance

Redis processes all operations in memory, offering sub-millisecond latency for reads and writes. This is essential for applications such as fraud detection, real-time dashboards, and recommendation engines.

Pub/Sub and Streams

Redis provides two powerful mechanisms for real-time data handling:

- **Pub/Sub**: Instantly pushes data to multiple subscribers for events like chat, notifications, and live tracking.

- **Streams**: Supports time-ordered log-like data ingestion with message IDs and consumer groups for durable and scalable stream processing.

Atomic Operations and Transactions

Redis supports atomicity, ensuring that operations are executed in order and without interference. Pipelining and transactions allow batch execution without risk of partial updates.

Data Expiration and TTL Management

Data can be configured to expire automatically, making Redis ideal for caching, temporary tokens, and real-time alerting systems that must auto-clean stale entries.

Scalability and High Availability

With Redis Cluster, data can be distributed across multiple nodes for horizontal scalability. Redis Sentinel provides automatic failover and monitoring to ensure uptime in production systems.

Built-in Replication and Persistence

Redis supports asynchronous replication and offers multiple persistence options (RDB

snapshots and AOF logs), allowing trade-offs between speed and durability based on application needs.

1.3.2. Advantages of Redis for Real-Time Applications

Redis stands out as a top-tier solution for high-velocity data workloads due to its simplicity, performance, and ecosystem support:

Real-Time Responsiveness

Redis allows systems to process and act on data in real-time, from anomaly detection to event-driven automation, offering immediate user feedback and improved decision-making.

Developer-Friendly Design

With intuitive commands and support for popular programming languages, Redis is easy to integrate and maintain, accelerating development cycles and deployment speed.

Versatile Use Cases

Redis is widely used for:

- Real-time analytics and counters.
- Session and token management.
- Leaderboards and rankings.
- Caching and precomputed query results.
- Machine learning feature stores.
- Real-time messaging and queuing.

Cost-Effective Performance

Due to its efficiency and low hardware requirements, Redis can deliver high throughput with minimal infrastructure, making it ideal for startups and large enterprises alike.

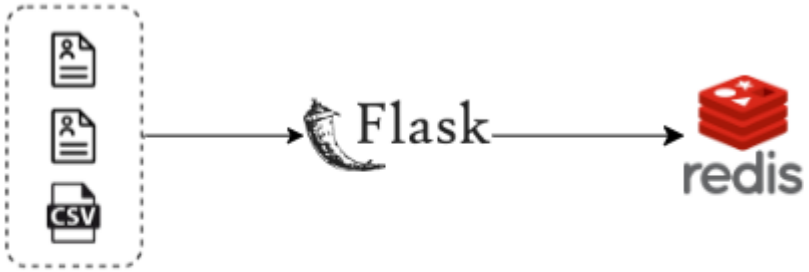
2. DATA PIPELINE ARCHITECTURE

2.1. Pipeline Overview

The data pipeline for this application is designed to support a responsive and scalable web service using Flask and Redis. The core architecture enables efficient request handling and rapid data access for real-time applications.

The front-end interacts with a Flask-based API server, which processes HTTP requests and manages application logic. Upon receiving a request, Flask communicates with Redis, an in-memory key-value store, to either retrieve or update data with ultra-low latency. Redis is used for its high-speed access and support for a variety of data structures such as strings, lists, and hashes.

This setup is ideal for scenarios that require caching, session management, or real-time analytics, significantly reducing the load on traditional databases and improving overall application performance. The lightweight nature of this pipeline ensures quick deployments and easy scalability in cloud or containerized environments.



Data sources

2.2. Component Details

2.2.1. Data Sources

In a hospital bed assignment system, Data Sources refer to the points or components that generate, provide and update data to serve the processing, analysis and decision making in real time. Specifically, the system exploits 2 main groups of data sources as follows:

- **Real-time Operational Data**: generated and updated continuously by the system during operation. This data describes the status of each hospital bed (in

use, empty), the list of patients waiting for bed assignment, as well as important events such as changes in the level of urgency when a patient's condition worsens, or when a patient completes treatment and is discharged. This is highly dynamic information, requiring the system to store and process it with extremely low latency. Redis is used as an intermediate storage, ensuring fast retrieval speed and instant updates.

- **User Input Data**: provided through the system interface by medical staff or the patient himself. This data includes patient information as well as necessary medical information such as medical condition, emergency level (from 1 to 5). This data is entered directly into the form on the web interface and sent to the server via API, serving the process of queuing and classifying patients.

2.2.2. Ingestion Layer

The Ingestion Layer, which acts as the first point in the pipeline, where data from different sources is received and fed into the processing system. In the bed assignment system, data is collected mainly through the user interface in the form. Users including medical staff or patients enter personal information, emergency indicators, medical conditions, and time of admission of the patient. These data are sent to the server through APIs built with Flask.

In addition to receiving administrative data, this layer is also responsible for recording real-time events such as: patients getting worse, beds becoming blank, or treatment status updates. These events are received continuously and immediately sent to Redis for further processing.

2.2.3. Processing Layer

Processing Layer, which processes important information to ensure that patient coordination is carried out according to the correct procedures and priorities. At this layer, Flask not only receives requests but also performs related processing such as assessing the level of urgency of each patient (from 1 to 5), organizing patient queues in order of priority, and deciding on bed assignment when rooms are available.

Operations such as updating patients in the queue, rearranging when a patient's condition worsens, or freeing up beds when patients are discharged are all performed at this

processing layer. Processing takes place almost instantaneously (real-time) to ensure that the system can respond quickly to real-life emergency situations.

2.2.4. Storage Layer

Storage Layer, responsible for storing and retrieving data quickly and efficiently. The system uses Redis as the main storage, a database with extremely fast data retrieval speed, suitable for real-time applications such as hospital bed assignment. Redis helps store patient lists, allowing retrieval based on priority.

This storage layer balances between instant retrieval and durable data preservation, while creating a foundation for expanding analytics and reporting functions in the future.

3. REAL-WORLD CASE STUDY: BENEFITS IN ACTION

3.1. Problem Proposal

Background: In modern health care, the rational and efficient allocation of medical resources is a key factor to ensure survival and improve the quality of treatment for patients. Especially in hospitals with limited resources such as a fixed number of beds, the problem of patient admission and manage classification becomes extremely important.

Based on that need, the idea of the operation of a hospital have 20 bed was formed. Each bed represents a fixed resource, and the system needs to ensure that patients with urgent conditions (emergency levels from 1 to 5, with 1 being the most serious) will be assigned beds first. Patients who come to the clinic will fill in their personal information and health status, then be put in the queue for bed assignment. The system will automatically allocate beds based on this priority level.

Methodology:

- When a patient arrives at the hospital, they will be asked to fill in complete personal and medical information, this information is stored for the system to analyze and assign beds.
- Each patient is assigned a Priority Level (1-5), the system will ensure that: Patients with lower levels (more emergency conditions) will be treated first. In the case of multiple patients with the same priority level, the patient who arrives first will be served first.

- The system will check the number of beds available (up to 20), If there are any available beds, the patient with the highest priority will be assigned to a specific bed. After the bed is assigned, the patient's treatment time begins. When the treatment time ends, the bed will be available and the system continues to allocate it to the next patient.
- In the list of patients waiting for bed assignment, there is a function button, which will update the priority level of that patient to 1, and at the same time move the patient to the front of the list. The system will immediately check for available beds and assign them if possible.

Academic Relevance: The system operates based on continuous reception of patient information and decision making on bed allocation based on emergency priority. This process requires rapid data processing, immediate response to changes such as new patients or worsening cases.

This clearly reflects the characteristics of Big Velocity Management, a branch of Big Data that focuses on:

- High-speed data processing in real time.
- Analyze and react flexibly to continuous data streams.
- Make decisions instantly, without delay.

The system will help classify patients according to their level of urgency, similar to stream processing in Big Velocity Management. The feature of updating the priority level when the patient becomes more serious is also a typical example of Big Velocity Management. This simulation system is a practical application of Big Velocity Management in the medical industry, where processing speed and timely decision-making ability directly affect human life.

3.2. Dataset Description

Structure of the Dataset: Each column provides specific information about the patient, their admission, and the healthcare services provided, making this dataset suitable for various data analysis and modeling tasks in the healthcare domain.

- **Name**: This column represents the name of the patient associated with the healthcare record.

- **Age**: The age of the patient at the time of admission, expressed in years.
- **Gender**: Indicates the gender of the patient, either "Male" or "Female."
- **Blood Type**: The patient's blood type, which can be one of the common blood types (e.g., "A+", "O-", etc.).
- **Medical Condition**: This column specifies the primary medical condition or diagnosis associated with the patient, such as "Diabetes," "Hypertension," "Asthma," and more.
- **Date of Admission**: The date on which the patient was admitted to the healthcare facility.
- **Doctor**: The name of the doctor responsible for the patient's care during their admission.
- **Hospital**: Identifies the healthcare facility or hospital where the patient was admitted.
- **Insurance Provider**: This column indicates the patient's insurance provider, which can be one of several options, including "Aetna," "Blue Cross," "Cigna," "UnitedHealthcare," and "Medicare."
- **Billing Amount**: The amount of money billed for the patient's healthcare services during their admission. This is expressed as a floating-point number.
- **Room Number**: The room number where the patient was accommodated during their admission.
- **Admission Type**: Specifies the type of admission, which can be "Emergency," "Elective," or "Urgent," reflecting the circumstances of the admission.
- **Discharge Date**: The date on which the patient was discharged from the healthcare facility, based on the admission date and a random number of days within a realistic range.
- **Medication**: Identifies a medication prescribed or administered to the patient during their admission. Examples include "Aspirin," "Ibuprofen," "Penicillin," "Paracetamol," and "Lipitor."

- **Test Results**: Describes the results of a medical test conducted during the patient's admission. Possible values include "Normal," "Abnormal," or "Inconclusive," indicating the outcome of the test.

With professions.

3.3. Implementation

3.3.1. Data Ingestion

Since Redis is a key-value based database system, it differs significantly from traditional relational databases like MySQL, which organize data in structured tables with predefined schemas. Redis does not support complex relational queries or require table definitions, making it more suitable for fast, in-memory operations with simpler data structures such as strings, hashes, lists, and sets. Due to this fundamental difference, it is not necessary—or even practical—to prepare a traditional dataset in the same way one would for MySQL or other relational databases.

To adapt to Redis's architecture and to streamline the development and demo process, our team chose to generate data programmatically within the code instead of relying on an external dataset. This approach allows us to simulate realistic data flow and system interactions dynamically, test edge cases with more flexibility, and maintain full control over the structure and volume of the data. It also reduces dependencies and setup time during testing and demonstration phases, making the development process more efficient and adaptable.

```
names = [
    "lYNn MaRtinez",
    "Bobby JacksOn",
    "tiMOthY CoLemaN",
    "LesLie TErRy",
    "DaNnY sMitH",
    "chRIsTOpHEr CHaPmAN",
    "hECTOR MAXweLL",
    "CHRis fRYe",
    "andrEw waTtS",
]
symptoms = ["Fever", "Chest Pain", "Bleeding", "Headache", "Breathing Problem"]
```

3.3.2. Storage and Preprocessing

Keys	Data Type	Fields	Description
Bed No.	Hash	patient_id, name, symptom, severity, bed_status	Store the data of the bed with patient ID using it and its status (occupied/available)
patient_stream	Stream	patient_id, name, symptom, severity, arrival_time	Store the data of incoming patients, but the data will be deleted after triaging
Patient ID	Hash	patient_id, name, symptom, severity, arrival_time	Store the data of incoming patients
triage:urgent	Ordered set	patient_id, time (score order)	Classify the patients having emergency severity
triage:normal	Ordered set	patient_id, time (score order)	Classify the patients having unpriority severity

After a specific amount of time (in this context is 0.5s), new patient will be generated and store his data in stream *patient_stream* and hash *patient ID*.

```
def simulate_patient():
    ID = 1
    while True:
        patient = {
            "id": ID,
            "name": random.choice(names),
            "symptom": random.choice(symptoms),
            "severity": str(random.randint(1, 5)),
            "arrival_time": str(time.time()),
        }
        r.xadd("patient_stream", {str(k): str(v) for k, v in patient.items()})
        r.hset(f"Patient {ID}", mapping=patient)
        ID += 1
        time.sleep(0.5)
```

We use two Redis data types—**hash** and **stream**—to store patient data. The reason for using **streams** is that they store a sequence of events or messages in the exact order they are

received, which helps us keep track of patient arrival times. However, when retrieving data from a stream, it is typically consumed and no longer available unless explicitly handled, so we also need to use **hashes** to persist patient information.

The reason we don't rely solely on hashes is because hashes do not maintain any inherent order, making it difficult to assign beds based on which patient arrived first.

3.3.3. Source code

For the source code of the project, please visit the following links:

https://github.com/KienKhuu/DBMS-project.git

3.4. Advantages of our DBMS solution

The integration of Flask and Redis in this project brings several distinct advantages, particularly for applications that demand real-time responsiveness, scalability, and simplicity.

- **Fast Data Access:** Redis, as an in-memory key-value store, provides extremely low-latency data access, making it ideal for caching, session storage, and scenarios requiring high-speed reads/writes.
- **Lightweight and Modular Architecture:** The Flask-Redis setup is lightweight and easy to deploy, especially in containerized environments. The modular nature of the system allows rapid development, testing, and maintenance.
- **Improved Application Performance:** By offloading temporary data and session state management to Redis, the system reduces the load on traditional databases, enhancing the performance and responsiveness of the web service.
- **Flexibility in Data Structures:** Redis supports multiple data types (strings, lists, sets, hashes), which makes it highly flexible for storing various kinds of data structures efficiently without additional schema definitions.
- **Open-source and Cost-effective:** Both Flask and Redis are open-source technologies, minimizing licensing costs and making the solution accessible and sustainable for student and small-scale production environments.

- **Real-time Data Handling:** This DBMS solution is well-suited for real-time analytics, live updates, and interactive applications where speed and minimal latency are critical.

Cost Overrun

Some auxiliary tools used for monitoring, testing, or CI/CD pipelines had usage limits under free plans. Upgrading to premium tiers or temporarily subscribing to external APIs contributed to a slight increase in project expenses.

Due to debugging challenges and multiple iterations of API-Redis communication logic, the development time was longer than initially estimated. While there were no direct monetary costs for human labor in a student-led project, this time overrun indirectly reflected in opportunity cost and delayed testing or documentation phases.

The team experimented with additional libraries or modules (e.g., RedisGraph, Flask extensions), some of which required more RAM or special deployment settings, resulting in increased infrastructure resource allocation.

4. CHALLENGES AND LIMITATIONS

Despite the simplicity and responsiveness of the Flask-Redis architecture, this project still faces several challenges and limitations that are common in lightweight and in-memory data systems.

- **Data Persistence and Durability:** Redis is an in-memory key-value store, which means that while it provides ultra-low latency access, data can be lost in the event of a system failure unless explicit persistence configurations (like RDB snapshots or AOF logs) are enabled. This introduces trade-offs between speed and reliability.

- **Scalability Constraints Although:** Redis supports clustering and sharding, the current setup uses a single-instance Redis server. This can become a

bottleneck under heavy traffic or high concurrency. Similarly, Flask is single-threaded by default and may not efficiently handle a large number of simultaneous requests without additional deployment strategies (e.g., using Gunicorn or uWSGI behind a reverse proxy).

- **Limited Analytical Capability:** The current architecture is optimized for fast read/write operations but lacks built-in analytical functions or historical data analysis support. For long-term storage, backup, or complex querying, integration with a relational or analytical database (e.g., PostgreSQL, ClickHouse) is required.

- **Security Considerations:** By default, Redis does not use authentication and operates on open ports, which may lead to security vulnerabilities if deployed in public-facing environments without proper protection measures like firewalls, authentication, or encryption.

- **No Monitoring or Observability Tools**: Currently, there is no monitoring layer integrated (e.g., Grafana, Prometheus, or logging systems) to track system performance or identify bottlenecks in real-time.

REFERENCE

[1] Redis Documentation. URL: https://redis.io/docs/. Official Redis documentation providing detailed guidance on data structures, commands, and high-performance use cases.

[2] Flask Documentation. URL: https://flask.palletsprojects.com/en/latest/. *Comprehensive documentation for Flask, a lightweight web framework for Python used to build APIs and web applications.*

[3] Prasad, P. (2022). *Healthcare Dataset*. Kaggle. URL: https://www.kaggle.com/datasets/prasad22/healthcare-dataset

[4] IBM Cloud Education. (2020). The Three V's of Big Data: Volume, Variety and Velocity. IBM. URL: https://www.ibm.com/cloud/learn/big-data