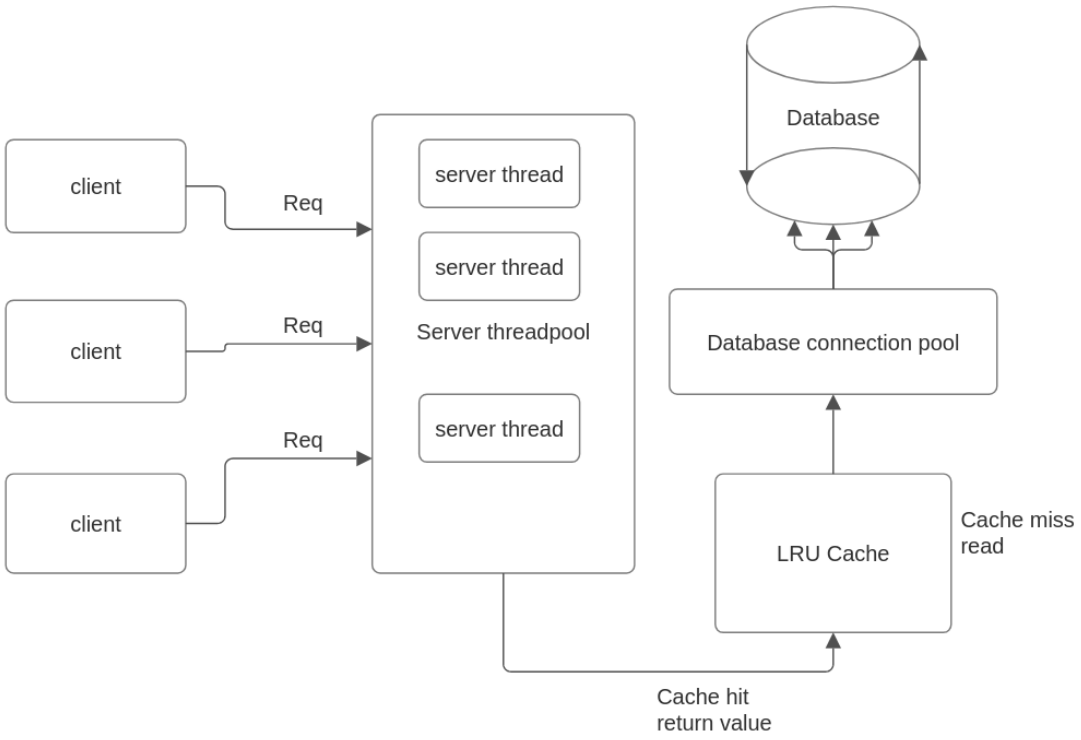
CS 744 - Phase 2 Project Report

Name : Mohit Chinchili
Roll no. 25m0767

1. System Architecture



The system consists of three main components:

1. Kv server
2. LRU cache
3. Postgres Databse

The KV server exposes RESTful create, read, and delete operations. Each request follows a structured path designed for both correctness and performance.

KV Server

Implemented using **cpp-httplib** for lightweight HTTP handling.

Uses a **thread pool** so multiple client requests can be served concurrently.

Each incoming request is processed through handlers for:

```
PUT /kv/<key>

GET /kv/<key>

DELETE /kv/<key>
```

LRU Cache

Server maintains a fixed-size in-memory cache.

Implements **Least Recently Used (LRU)** eviction.

- Serve frequently accessed reads directly from memory.
- Reduce load on PostgreSQL.
- Improve average latency for get requests..

A read request proceeds as:

1. Check cache → return immediately if present.
2. If absent → fetch from DB → insert into cache.

A write request:

1. Updates both cache and database.

A delete request:

1. Removes from database (primary copy).
2. If present, also remove from cache to maintain consistency.

Database Connection Pool

- Instead of creating a new database connection for each request, a **connection pool** maintains several open connections.
- Each server thread borrows a connection, performs the query, and returns it to the pool.
- This reduces connection overhead and improves throughput under concurrency.

PostgreSQL Storage

- Persistent key-value storage (`key INT PRIMARY KEY`, `value TEXT`). Handles durability, crash recovery, and storage management.
- Supports millions of keys for large-scale load testing.

---

2. Key Changes From Phase 1

1. Changed `INSERT` to `UPSERT`

Phase 1 used plain `INSERT`, which failed on duplicate keys and complicated testing.
Phase 2 replaces this with:

```
INSERT INTO kv (key, value)
VALUES ($1, $2)
ON CONFLICT (key) DO UPDATE SET value = EXCLUDED.value;
```

Benefits:

- Simplifies load testing (unique-key coordination not required)
- Behaves like real KV stores (PUT = create or update)
- Removes unnecessary insert failures

---

2. Added Thread Pool for Server

Phase 1 relied on httplib's default threading model, which is limited.
Phase 2 introduces a configured **thread pool**, allowing the server to handle significantly more simultaneous requests.

Benefits:

- Higher throughput under load
- Better parallelism
- Prevent request backlog delays

---

3. Added Database Connection Pool

Phase 1 opened new PostgreSQL connections per request, which was slow and created backend churn.
Phase 2 introduces a persistent pool of connections reused across requests.

Benefits:

- Substantially lower request latency
- Higher concurrency (many DB operations in parallel)
- Reduced overhead and fewer connection storms

---

4. Moved Database Logic Directly into Server Handlers

Earlier logic was split across helper functions and scattered error paths.
Phase 2 performs all DB operations directly inside request handlers with a unified `try-catch` structure.

Benefits:

- Cleaner and more maintainable code
- Simpler error reporting
- Easier debugging of request-to-DB flow
- Clear relationship between REST operations and SQL execution

---

3. Load Generator Design

The load generator is designed to simulate realistic client behavior under controlled conditions and to discover system bottlenecks such as CPU saturation, database contention, or disk I/O limits.

Thread Model

Multi-threaded: each thread represents one client.
Closed-loop operation:

Send request, Wait for response, Immediately send the next request, No built-in think time unless specified.

This design ensures the generator *always* pushes the server as fast as the server can respond, revealing real bottlenecks.

Workload Modes

The generator supports several workload patterns:

Get All

Large key space (e.g., 10M–50M keys).

Every request becomes a random key lookup.

Produces primarily **disk-bound** behavior when dataset exceeds RAM.

Put All

Continuous upserts on random keys.

Stresses:

DB write path, WAL generation, Locking behavior

Mixed Read/Write

Random mixture of GET, PUT, DELETE. Produces a combination of CPU, lock contention, and IO load.

Metrics Collected

Each client thread records response start–end times.
The generator aggregates them to compute:

- **Throughput (requests per second)**
- **Average response time**
- **Tail latencies (optional)**

- **Success vs failure rates**

These metrics are correlated with system monitoring tools:

- `iostat` for disk utilization
- `htop` or `mpstat` for CPU
- PostgreSQL stats (optional)

These help identify whether a workload is CPU-bound, disk-bound, or limited by DB connection pool or server thread pool.

---

4. Load Testing

All load testing was following standard conditions:

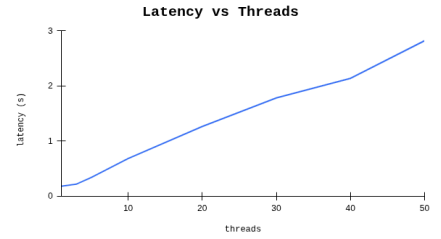
- server pinned to core 2,3
- Postgres pinned to core 0,1
- loadgen pinned to core 5,6
- duration of 5 mins
- cpu utilisation, io utilisation monitored for each process

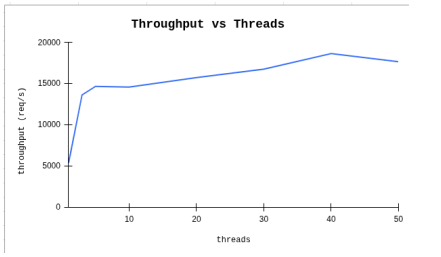
Get All (bottleneck - cpu)

The first workload issues only read operations. Due to the LRU cache, a large portion of these requests result in cache hits, meaning the frontend server handles most of the load while backend database traffic remains comparatively low. The table below summarizes the metrics collected under different load levels.

Threads	1	3	5	10	20	30	40	50
Cpu core 0	20.5	59.9	77.8	85.7	96.2	93.1	95.2	96.2
Cpu core 1	28.2	60.5	80.5	87.3	96.2	96.7	95.2	96.8
Cpu core 2	26.8	65.7	82.1	88.4	90.8	98.7	91.4	90.6
Cpu core 3	28.3	59.3	77.2	84.7	87	99.4	87.3	86.5
Cpu core 5	18.5	44.3	58.1	68.2	70.6	80.5	75.2	71.8
Cpu core 6	31.3	45.4	58.6	70.3	70.5	85.4	75.2	73

iostat	3.4	4.52	4.04	4.16	5.08	7.8	4.18	4.47
Throughput	5419.55	13634.87	14678.9	14596.13	15750.85	16776.88	18649.05	17671.95
Latency	0.184	0.219	0.34	0.683	1.266	1.784	2.139	2.822





Put All (bottleneck - disk io)

The second workload is purely write requests. In this load, continuous write requests are sent out to the backend that handles the database. That would be the case but the read writes are still quite fast in NVMes. Hence the bottleneck is still the cpu. Tabulated data of various metrics across different loads is as follows:

Threads	**1**	**3**	**5**	**10**	**20**	**30**	**40**	**50**
Cpu core 0	26.4	30.1	48	56.8	64.9	56.2	56.2	56.2
Cpu core 1	23.3	25.8	41	56.1	65.2	56.8	56.2	56.7
Cpu core 2	15.3	21.4	20.1	46.1	48.3	45.6	45.9	44
Cpu core 3	16.9	14.9	28.5	41	45.2	42.9	43	39.8
Cpu core 5	16.4	14.4	19.7	32.1	43.2	31.3	33.5	32.2
Cpu core 6	11.5	15.1	22.3	32.1	44.3	32.7	34.8	32.4

iostat	60.38	84.2	79.8	75.9	72.2	74.8	76.5	76.1
Throughput	1423.98	3131.66	4710.81	6968.15	6940.43	7065.37	7287.69	6715.66
Latency	0.7	0.955	1.059	1.432	1.438	4.242	5.484	7.099