CS744 : DECS Project KVserver Final Report

Arpit Meher (25m0813)

Github Link : https://github.com/arpitmeher/CS744_Project_25m0813

SERVER:

The server component of KVServer is built using the `cpp-httplib` library, which provides a simple interface for defining HTTP routes and handling requests. It exposes four main endpoints: `/create` for storing key-value pairs, `/read` for retrieving values, `/delete` for removing keys, and `/stats` for reporting usage metrics. The server is configured with a thread pool sized to the number of logical CPU cores using `std::thread::hardware_concurrency()`, allowing it to handle multiple concurrent requests efficiently. Each incoming request is processed by a dedicated thread, which interacts with the cache and database layers in a thread-safe manner. The server validates inputs, updates statistics, and coordinates between the cache and database to ensure consistent and performant behavior.

CACHE:

The cache layer is implemented as an in-memory LRU (Least Recently Used) cache using a combination of `std::unordered_map` for fast key lookup and `std::list` for tracking usage order. This design allows constant-time access and eviction, ensuring that frequently accessed keys remain in memory while older entries are removed when capacity is exceeded. The cache supports `put`, `get`, and `remove` operations, and is protected by a `std::mutex` to ensure thread safety during concurrent access. On a GET request, the cache is checked first; if the key is found, the value is returned immediately and counted as a cache hit. If not, the request falls back to the database, and the result is inserted into the cache for future access. This caching strategy significantly reduces database load and improves response times for repeated queries.

DATABASE:

The database layer uses MySQL to persist key-value pairs in a table named `kv_table`, with columns `k` (key) and `v` (value). Access to the database is managed through the `DBConnector` class, which wraps the MySQL C++ Connector API and provides methods for `put`, `get`, and `remove` operations. To support concurrent access, the system includes a thread-safe

connection pool implemented using `std::queue`, `std::mutex`, and `std::condition_variable`. This pool pre-allocates a fixed number of connections and allows threads to acquire and release them as needed, avoiding the overhead of frequent connection creation and ensuring safe reuse. The `remove` method checks the number of affected rows to determine whether a key existed before deletion, allowing the server to return accurate responses. This modular and efficient database layer ensures reliable persistence and smooth integration with the rest of the system.

LOAD GENERATOR:

The load generator is implemented as a standalone C++ client using **cpp-httplib** to issue HTTP requests against the KVServer endpoints. It supports multiple workload patterns to simulate realistic usage scenarios:

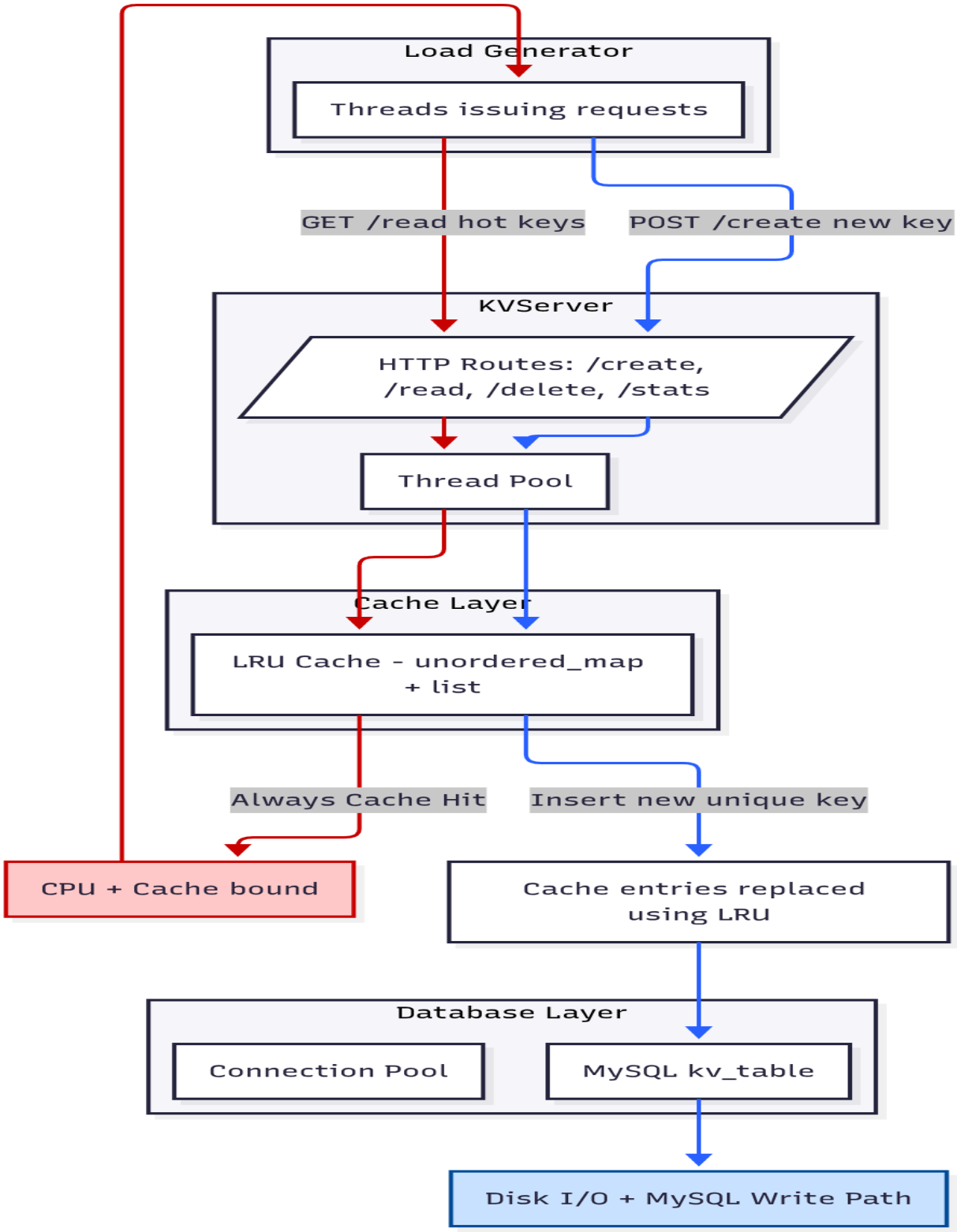
- **PutAll Workload**: continuously inserts new key-value pairs into the server.
- **GetAll Workload**: retrieves values for randomly generated keys.
- **GetPopular Workload**: repeatedly queries a fixed set of “hot” keys to simulate skewed access distributions. Number of keys is 100. And they can be feed into the database using feed_keys.sh script.
- **Mixed Workload**: combines create, read, and popular access patterns randomly.

Note: Only PutALL Workload and GetPopular Workload used for testing

Each workload is executed by multiple client threads, with the number of threads and test duration configurable via command-line arguments. The generator measures **throughput (requests per second)** and **average response time (latency)** by recording timestamps before and after each request. Results are aggregated across threads using atomic counters.

The load generator operates in a **closed-loop model**, meaning each thread waits for the server’s response before issuing the next request. This ensures that throughput reflects the server’s actual capacity under load rather than a fixed request rate

Flowchart of two Workloads along with system architecture:



LOAD TEST SETUP and RESULTS:

After running make:

1. **For GetPopular Workload** : It's CPU bound as all the same hundred keys are queried again and again and it ensures that it always results in cache hit.

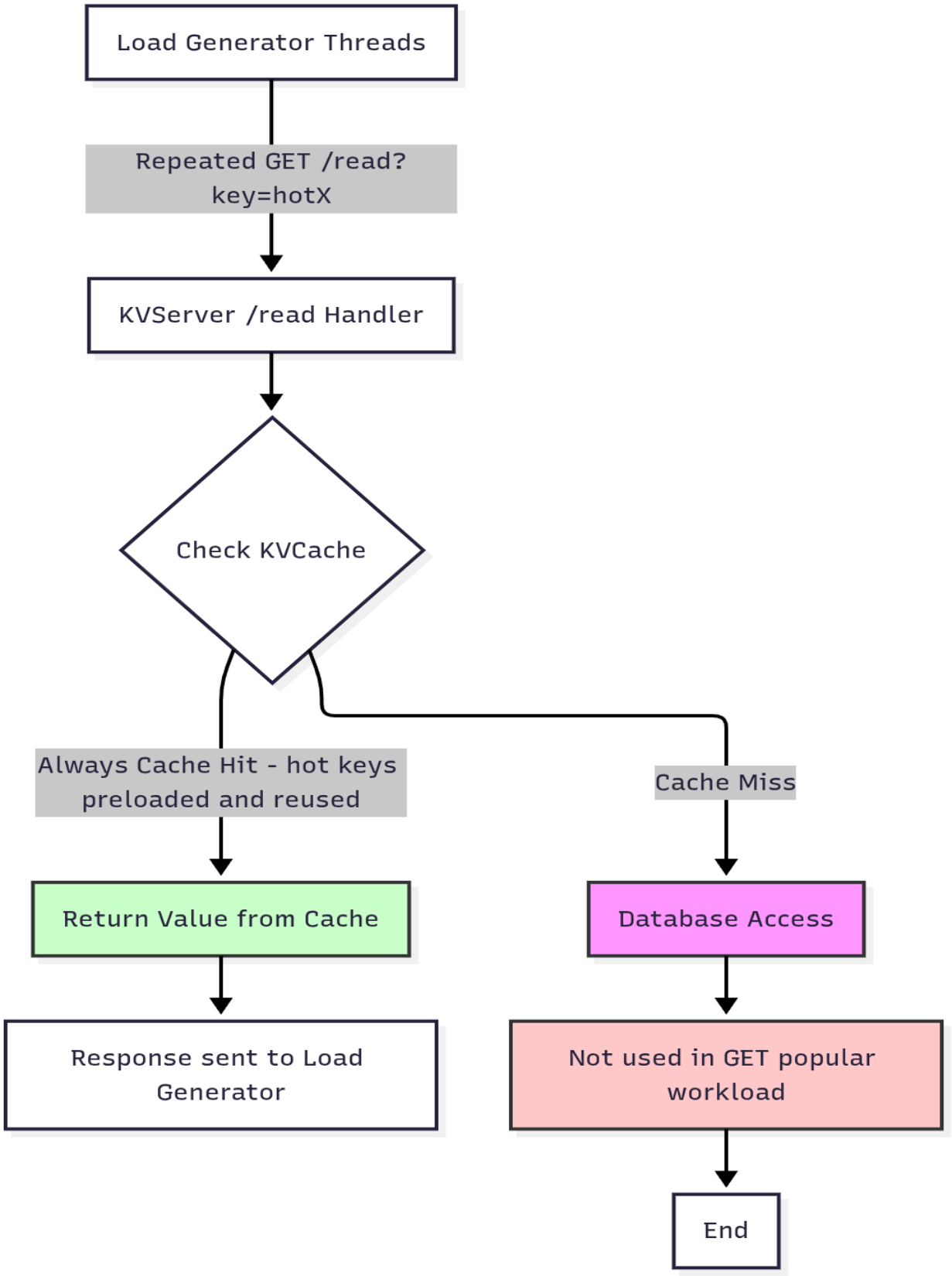
   Observation: if kv_server pinned on just one core cpu utilization reaches 100% for very low thread value.

   On more than one cores cpu utilization doesn't go beyond 92-93%.

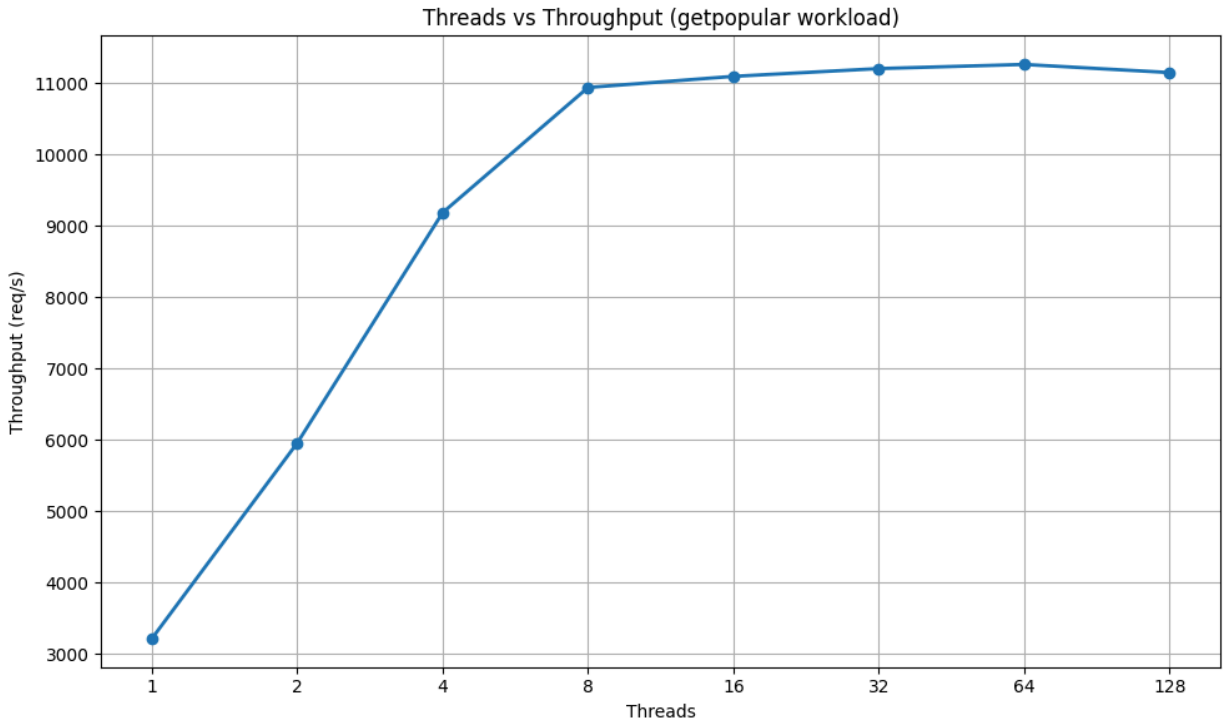
   Using thread counts 1, 2, 4, 8, 16, 32, 64

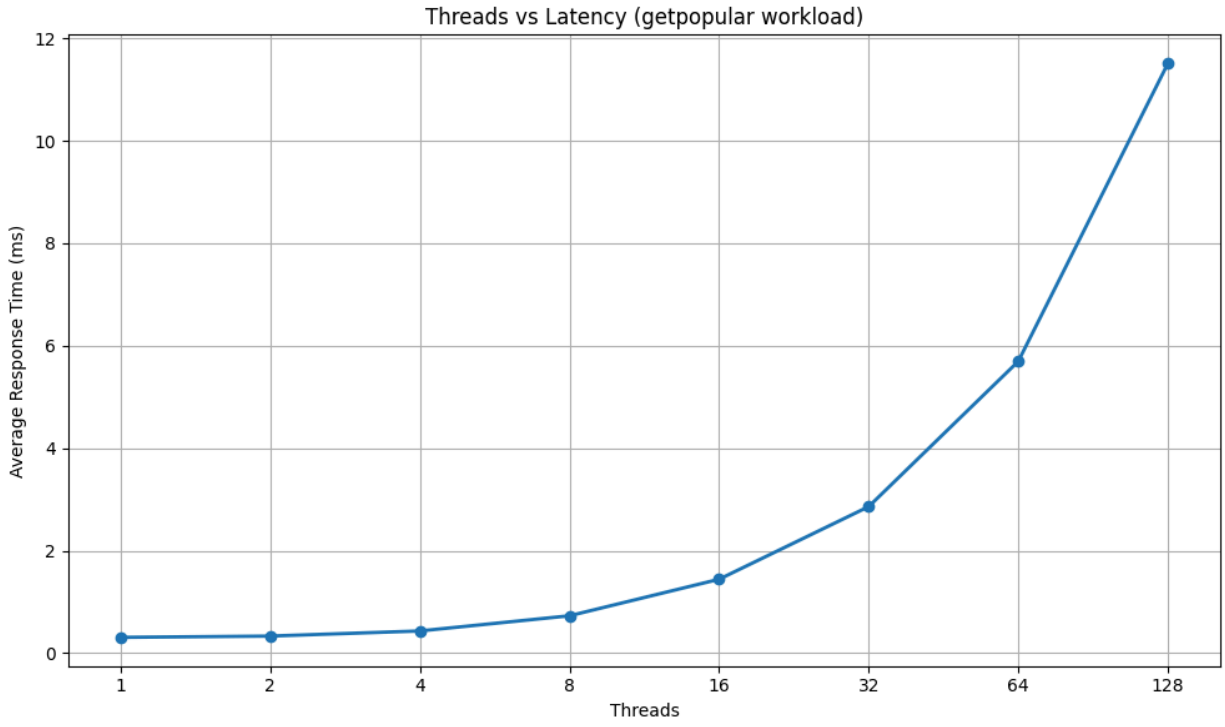
```shell
# pin kv server on core 1 and 2 run following in build directory
taskset -c 1-2 ./kv_server

# To put the 100 popular keys in database (Uses CURL in for loop)
./feed_keys.sh

# To get process id of mysql server
pgrep mysql
# To pin mysql server on core 0
sudo taskget -cp 0 <pid>

# To send request using loadgen which is pinned on core 3-15
taskset -c 3-15 ./loadgen --workload getpopular --threads 2 --duration 180 >>
results_get_pop.csv

# run top to see cpu and mem usage
top

# run htop to see cpu usage of each core
htop

# optionally log disk usage using
iostat -x 1 > disk_usage.log &
```

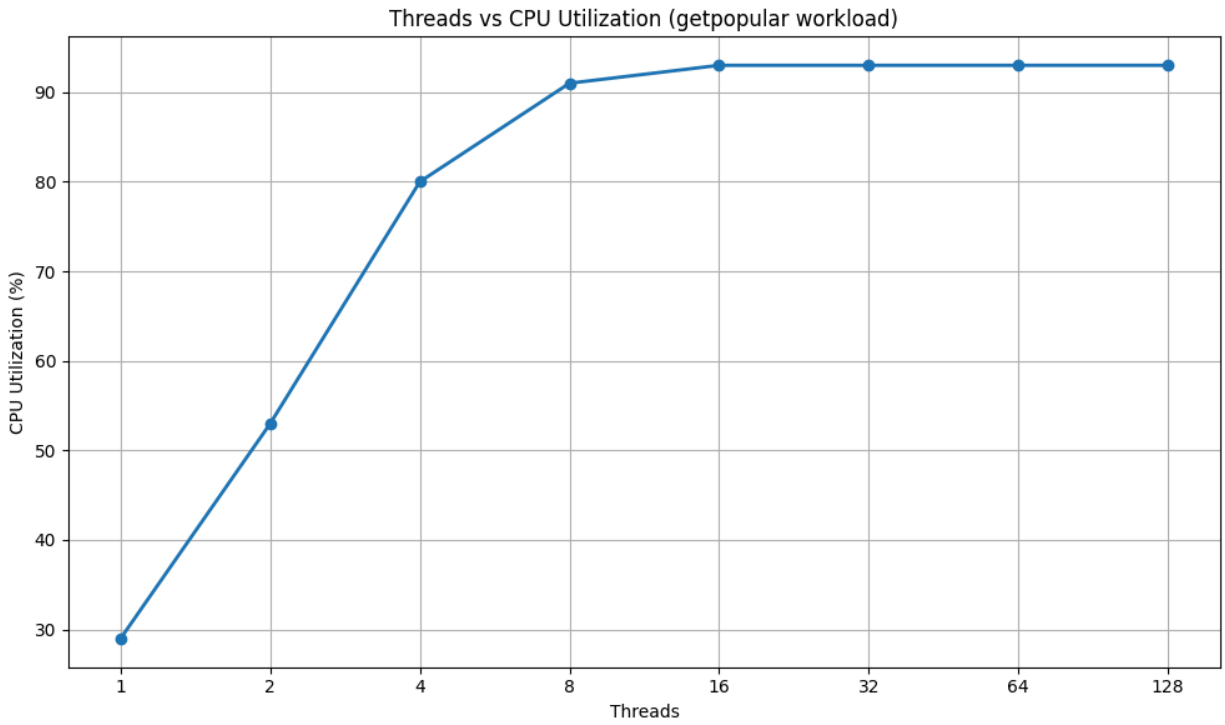
Flow chart for GetPopular:



EXPERIMENT RESULT FOR WORKLOAD 1 CPU - BOUND







CPU USAGE - 93% approx
Throughput after hitting bottleneck - 11200 req/sec approx

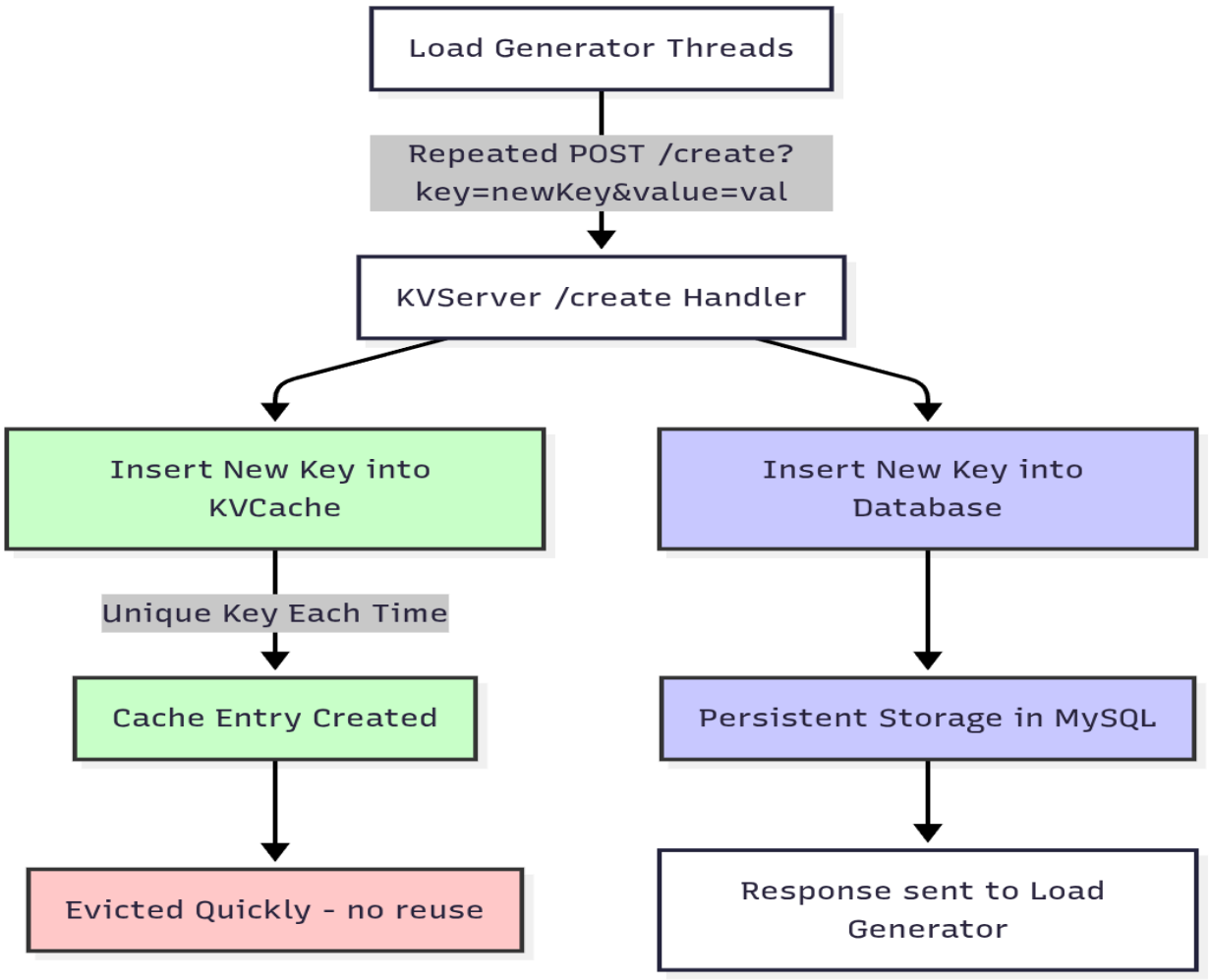
2. **For PutAll Workload** : It's disk bound as a new key is inserted each time so the server has to put it into the database ensuring disk access.

   Disk Usage doesn't exceed about 78% no matter how many threads used even when server cores are not saturated.

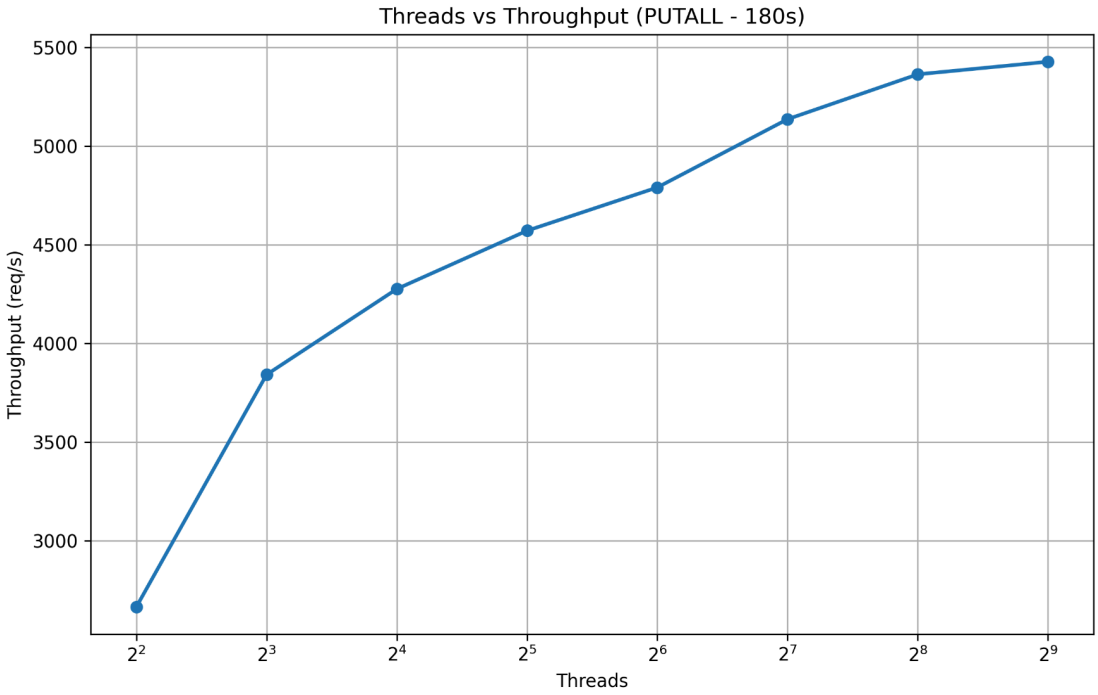
```shell
# pin kv server on core 1 and 2 run following in build directory
taskset -c 1-2 ./kv_server

# To send request using loadgen which is pinned on core 3-15
taskset -c 3-15 ./loadgen --workload putall --threads 2 --duration 180 >>
results_putall.csv

# run top to see cpu and mem usage
top
```

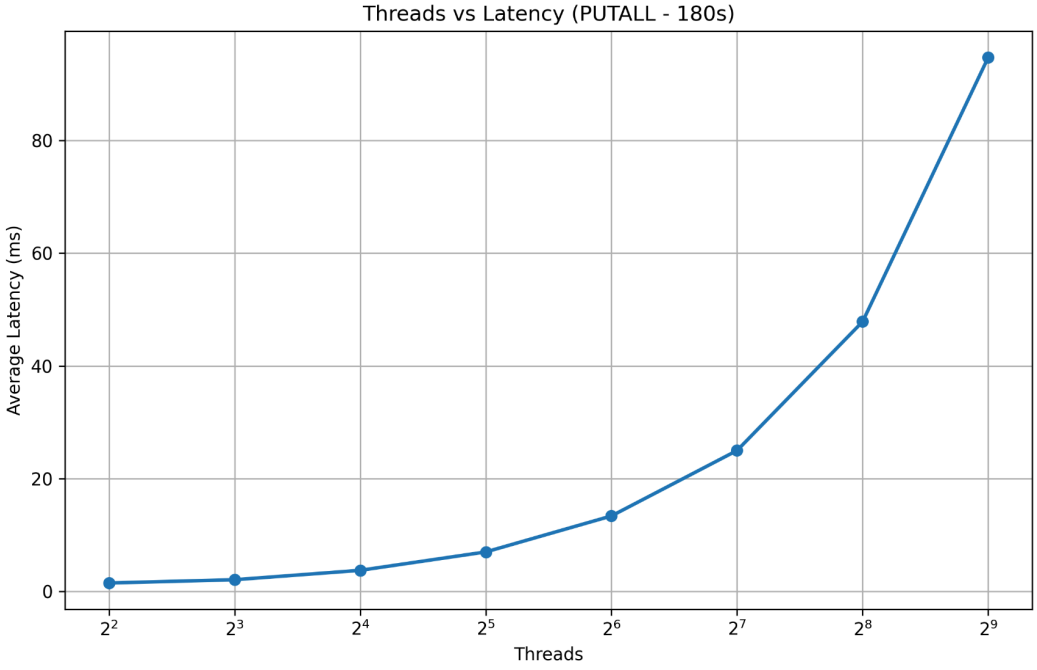
```
# run htop to see cpu usage of each core
htop

# log disk usage using every 3 sec logged in background
sar -d 3 > disk_usage.log &

# to see disk usage printed every 2 sec
iostat -d -x 2
```

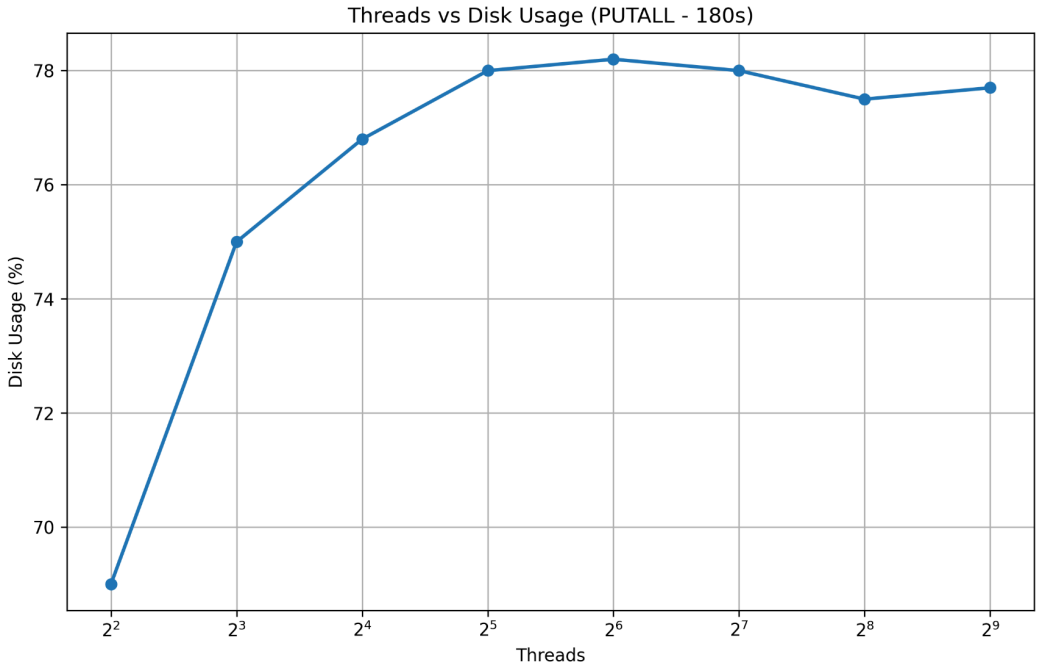
Flowchart for putall:



EXPERIMENT RESULT FOR WORKLOAD 2 Disk - BOUND







Disk Usage - 78 % approx

Throughput after hitting bottleneck - 5300 req/sec approx

To install dependencies:

```shell
sudo apt update
sudo apt install g++ cmake libmysqlcppconn-dev libssl-dev mysql-server
```

Creating Database:

```sql
CREATE DATABASE kvstore;
USE kvstore;
CREATE TABLE kv_table (k VARCHAR(64) PRIMARY KEY, v TEXT);
```

Build using Cmake:

```shell
mkdir build && cd build
cmake ..
make
```

Running server: if you are already in project/build directory

```shell
./kv_server
```

Curl syntax to create, get, remove key val pairs:

```shell
curl -X POST http://localhost:8080/create -d "key=key&value=val"
curl "http://localhost:8080/read?key=key"
curl -X DELETE "http://localhost:8080/delete?key=key"
```

If your MySQL password or user is different, update it in `db_connector.cpp`

```cpp
user = "your_user";
password = "your_password";
```