ĐẠI HỌC QUỐC GIA HÀ NỘI
ĐẠI HỌC CÔNG NGHỆ

BÁO CÁO TỔNG KẾT ĐỀ TÀI NGHIÊN CỨU

Đề tài: "Nghiên cứu hệ Quản trị Cơ sở dữ liệu Redis"

Hà Nội, 01/2019

Danh sách những người thực hiện

TT	Họ và tên	Học hàm, học vị, chuyên môn	Chức vụ	Đơn vị
1	Phạm Tuấn Anh	Kỹ sư	Nhân viên	Tổng Công ty Mạng lưới VIETTEL
2	Nguyễn Tuấn Anh	Kỹ sư	Nhân viên	Tổng Công ty Mạng lưới VIETTEL
3	Trần Thọ Hoàng	Kỹ sư	Nhân viên	Tổng Công ty Viễn thông VIETTEL

Mục lục

TT **Tr**

Danh mục

I. LỜI MỞ ĐẦU

Ngày nay, khái niệm NoSQL trở nên không còn xa lạ trong giới Công Nghệ Thông Tin (CNTT). Đi kèm với đó là sự ra đời của hàng loạt hệ quản trị cơ sở dữ liệu (DBMS) phát triển dựa trên đặc thù của NoSQL: Non-relational (không quan hệ), Distributed (phân tán), Open-source (mã nguồn mở), Horizontally scalable (dễ dàng mở rộng theo chiều ngang). Có thể kể đến vài cái tên phổ biến hiện nay như Redis, MongoDB, Hbase, Cassandra. Bên cạnh những tri thức mới thì có thể thấy sự ra đời của các Hệ Quản trị Cơ sở Dữ liệu với cấu trúc NoSQL mang đến những lợi ích nhất định trong việc quy hoạch thiết kế cũng như triển khai các hệ thống phần mềm, ứng dụng mới ngày nay.

Vì vậy, bài Nghiên cứu này đưa ra nội dung kiến thức về một Hệ Quản trị Cơ sở Dữ liệu với cấu trúc NoSQL được ứng dụng rộng rãi ngày nay, đó là Redis.

II. NỘI DUNG CHÍNH

a. Mục đích và đối tượng chính

- Redis là một lưu trữ cấu trúc dữ liệu trên memory, được sử dụng làm cache, database, message broker (convert format bản tin của bên gửi thành format của bên nhận).
- Redis lưu trữ trên memory dữ liệu dưới dạng KEY-VALUE: là 1 tập các cặp (KEY,VALUE), trong đó KEY là duy nhất trong tập dữ liệu, còn VALUE có thể là bất cứ thứ gì.

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

- Redis ngoài tính năng lưu trữ KEY-VALUE trên RAM thì Redis còn hỗ trợ tính năng lưu trữ dữ liệu trên đĩa cứng cho phép bạn có thể phục hồi dữ liệu khi hệ thống gặp sự cố.
- Redis hỗ trợ tính năng replication(master-slave) cho phép bạn có thể sao chép, đồng bộ giữa 2 CSDL Redis với nhau. Ngoài ra còn có tính năng cluster cũng đang được phát triển cho phép load balancing(cân bằng tải).
- Redis có dữ liệu được nhân bản để đảm bảo dự phòng về mặt dữ liệu (replicaton) và chạy theo cluster để đảm bảo về mặt ứng dụng (với redis sentinel).
- Redis cũng cung cấp các toán tử để xử lý dữ liệu như cộng string, đẩy dữ liệu vào list. Do làm việc dữ liệu trên memory nên redis có hiệu năng cao.

b. Mô hình dữ liệu được sử dụng

- Khác với RDMS như MySQL, hay PostgreSQL, Redis không có table (bảng). Redis lưu trữ data dưới dạng key-value. Thực tế thì memcache cũng làm vậy, nhưng kiểu dữ liệu của memcache bị hạn chế, không đa dạng được

như Redis, do đó không hỗ trợ được nhiều thao tác từ phía người dùng. Dưới đây là sơ lược về các kiểu dữ liệu Redis dùng để lưu value.

- **STRING**: Có thể là string, integer hoặc float. Redis có thể làm việc với cả string, từng phần của string, cũng như tăng/giảm giá trị của integer, float.
- **LIST**: Danh sách liên kết của các strings. Redis hỗ trợ các thao tác push, pop từ cả 2 phía của list, trim dựa theo offset, đọc 1 hoặc nhiều items của list, tìm kiếm và xóa giá trị.
- **SET**: Tập hợp các string (không được sắp xếp). Redis hỗ trợ các thao tác thêm, đọc, xóa từng phần tử, kiểm tra sự xuất hiện của phần tử trong tập hợp. Ngoài ra Redis còn hỗ trợ các phép toán tập hợp, gồm intersect/union/difference.
- **HASH**: Lưu trữ hash table của các cặp key-value, trong đó key được sắp xếp ngẫu nhiên, không theo thứ tự nào cả. Redis hỗ trợ các thao tác thêm, đọc, xóa từng phần tử, cũng như đọc tất cả giá trị.
- **ZSET (sorted set)**: Là 1 danh sách, trong đó mỗi phần tử là map của 1 string (member) và 1 floating-point number (score), danh sách được sắp xếp theo score này. Redis hỗ trợ thao tác thêm, đọc, xóa từng phần tử, lấy ra các phần tử dựa theo range của score hoặc của string.

c. Phương pháp và kỹ thuật xử lý các truy vấn/giao tác đồng thời

- **Redis Persistence**
  - Redis cung cấp 2 lựa chọn để lưu trữ dữ liệu xuống disk là snapshotting và append-only file để đảm bảo khi có lỗi xảy ra với dữ liệu trên memory, redis sẽ phục hồi lại được từ disk.
  - **Snapshotting**
    - Redis tạo 1 bản copy dữ liệu từ memory gọi là snapshot (được lưu trữ vào file "dump.rdb"). Bản snapshot này có thể được lưu tại tại đĩa hoặc copy sang các server khác để dự phòng. Mỗi lần tạo snapshot, dữ liệu sẽ được ghi vào file tạm. Sau khi hoàn thành nó sẽ thay thế file cũ. Nếu Redis có lỗi xảy ra thì các dữ liệu thay đổi trên memory sẽ bị mất kể từ snapshot cuối cùng được ghi trên đĩa. Có 5 cách để tạo snapshot:
      - BGSAVE (trừ window): Khi chạy BGSAVE, Redis sẽ dừng lại để tạo ra 1 tiến trình con (fork) để thực hiện Snapshotting, thời gian dừng lại phụ thuộc vào dung lượng dữ liệu của redis (10-20ms/1G dữ liệu). BGSAVE sẽ gây tốn tài nguyên RAM của Redis, trong trường hợp không đủ bộ nhớ, hệ điều hành sẽ kill tiến trình con này. Để tránh việc này, người ta sử dụng

tham số overcomit_memory. Redis vẫn tiếp nhận request và xử lý bình thường sau khi tiến trình con được tạo ra (chú ý, khi chạy BGSAVE, tiến trình con sẽ chiếm số lượng RAM gần bằng tiến trình cha)

- SAVE: Redis sẽ dừng tiếp nhận yêu cầu cho đến khi snapshot hoàn thành việc ghi lại, SAVE thực thi nhanh hơn BGSAVE.
- Save time number_of_write: Redis sẽ chạy BGSAVE khi đến ngưỡng số lần thay đổi trong vòng time giây kể từ snapshot trước.
- Shutdown: Khi nhận được lệnh shutdown, Redis sẽ thực hiện SAVE trước khi shutdown.
- Sync: khi nhân được lệnh sync, master redis server sẽ chạy BGSAVE.

- **Ưu điểm:**
  - RDB nó là 1 snapshot của dữ liệu hiện tại =>thuận tiện cho việc backup, lưu trữ và phục hồi (file backup được lưu sang server khác, tổng trạm khác)
  - Với BGSAVE, RDB tối đa hiệu năng của redis bởi vì tiến trình cha chỉ cần làm mỗi việc tạo tiến trình con.
  - Khôi phục nhanh đối với các dữ liệu lớn so với sử dụng AOF.
- **Nhược điểm:**
  - Có khả năng mất dữ liệu giữa các lần snapshot.
  - Với dữ liệu lớn, tiến trình con sẽ chiếm nhiều tài nguyên ảnh hưởng đến hiệu năng của Redis

- **Append-only File**
  - Redis sẽ ghi tất cả các dữ liệu được thay đổi dữ liệu vào file .aof. Khi Redis được restart lại, nó sẽ đọc từ đầu đến cuối file để dựng lại dữ liệu. Có 3 lựa chọn cho việc ghi dữ liệu xuống đĩa với Append-only file.

Option	How often syncing will occur
always	Every write command to Redis results in a write to disk. This slows Redis down substantially if used.
everysec	Once per second, explicitly syncs write commands to disk.
no	Lets the operating system control syncing to disk.

- Với lựa chọn no, OS sẽ thực hiện ghi xuống đĩa, default 30s với linux.
- Các bản ghi thay đổi liên tục được ghi xuống, cùng 1 bản ghi có thể có rất nhiều version. Redis cung cấp 1 tính năng là tạo lại file AOF mới. Khi thực hiện ghi, Redis tạo tiến trình con để thực hiện tạo 1 file tạm AOF mới ghi lại các lệnh ít nhất để thay đổi bản ghi. Tiến trình cha thực hiện tổng hợp tất cả các thay đổi trong buffer memory. Sau khi tiến trình con ghi lại file mới xong, tiến trình cha sẽ ghi phần thay đổi xuống vào cuối file của tiến trình con. Sau khi ghi xong nó sẽ thay thế file mới bằng file cũ. Mặc định Redis sẽ chạy rewrite log khi dữ liệu AOF lớn hơn 100% file rewrite log trước đó hoặc AOF lớn hơn 64MB.
- **Ưu điểm:**
  - Lưu thay đổi liên tục nên giảm thiểu việc mất dữ liệu khi restart Redis.
  - Dữ liệu ghi chỉ là file nên không có vấn đề về việc tìm kiếm và corruption.
  - Tự động rewrite lại AOL file để giảm lệnh thay đổi vào cùng 1 bản ghi.
  - AOF ghi lại các lệnh thay đổi theo thứ tự dễ dàng cho việc hiểu và phân tích.
- **Nhược điểm:**
  - Dung lượng AOF file lớn nhiều so với RDB của cùng 1 dataset.
  - Việc ghi AOF chậm hơn RDB (hiệu năng hệ thống do AOF liên tục ghi).
  - AOF có bug trong 1 số lệnh cụ thể, khi khôi phục dữ liệu sẽ không giống như ban đầu.
- Redis khuyến nghị nên sử dụng song song 2 phương pháp đồng thời. Bằng cách bật RDB và set lệnh "config set appendonly yes" (không sửa file config), nếu đặt config thì Redis sẽ ưu tiên load AOL hơn. Redis có cơ chế để BGREWRITEAOF và Snapshotting không diễn ra đồng thời.

- **Redis Replication**
  - Redis hoạt động theo cơ chế 1 master- nhiều slave. Các dữ liệu được ghi vào master cũng sẽ đồng bộ vào các slave để phục vụ việc truy vấn dữ liệu. Các client có thể được cấu hình chỉ đọc để

giảm tải cho master. Redis replica có 3 tác dụng: Giảm tải cho master, tăng độ dự phòng cho hệ thống, lưu trữ backup dữ liệu ngay trên memory mà không cần lưu xuống ổ cứng. Cơ chế hoạt động như sau:

- Khi kết nối giữa Master-slaves bình thường, Master gửi liên tục các lệnh vào slave để đảm bảo dữ liệu của master và slave giống nhau khi có yêu cầu ghi, key expired hoặc evicted, hoặc bất cứ thay đổi gì đến dataset của master.
- Khi Master-client bị mất kết nối hoặc quá thời gian timeout trên cả master/slave, slave sẽ kết nối lại và thực hiện xử lý đồng bộ lại 1 phần (lấy lại các comand bị thất thoát).
- Nếu việc đồng bộ 1 phần không được, slave sẽ yêu cầu đồng bộ toàn bộ. Master sẽ tạo 1 snapshop gửi cho slave và tiếp tục gửi các lệnh thay đổi sang slave.

- Redis sử dụng việc đồng bộ khác thời kỳ (mặc định tức là ghi vào master xong mới ghi sang các slave) và các slave gửi xác nhận định kỳ về việc xử lý một lượng dữ liệu về master không cùng lúc. Master cũng không cần đợi slave xử lý xong 1 command.
- Master vẫn có thể xử lý query khi nó nhận được yêu cầu đồng bộ cho slave. Slave vẫn xử lý được query khi nó thực hiện đồng bộ 1 phần hoặc trong giai đoạn khởi tạo lại việc đồng bộ với dữ liệu cũ. Với redis 4.0 thì hỗ trợ trong cả trường hợp đang đồng bộ full.
- Ngoài việc các slave có thể kết nối với các slave khác trong cùng 1 master, nó còn có thể kết nối với các slave khác trong cụm.
- Lưu ý: Nếu tắt tính năng persitent thì phải turn off tính năng auto restart của master để đảm bảo không bị mất hết dữ liệu khi master khôi phục lại trước khi được phát hiện là down.
- **Chi tiết hoạt động của Replication**
  - Mỗi 1 master có 1 cặp 1(Replication ID, offset) để xác định version chính xác của data set.. Replication ID string ngẫu nhiên để mô tả lịch sử về dataset và offset được tăng lên cho mỗi byte được gửi tới slave (để đánh dấu đã gửi gì đi).
  - Khi Slave kết nối lại với master, nó sẽ gửi lệnh PSYNC để gửi lại cặp (Replication ID, offset) cũ mà nó đã xử lý. Dựa vào cặp cũ này, Master sẽ biết cần gửi những gì cho slave. Trong trường không đủ backlog trong buffer hoặc slave gửi

Replication ID mà master không biết, master sẽ thực hiện đồng bộ toàn bộ: Master thực hiện BGSAVE để tạo 1 file RDB đồng thời cũng buffer lại toàn bộ các thay đổi mới từ thời điểm này (backlog). Khi RDB được ghi xong, nó sẽ gửi file này cho slave lưu trữ vào đĩa. Slave sẽ thực hiện load dữ liệu này lên memory. Sau đó Master sẽ tiếp tục gửi toàn bộ các lệnh trên buffer sang client như bình thường (nếu nhiều slave cùng yêu cầu SYNC đồng thời, master chỉ chạy 1 BGSAVE cho tất cả slave).

- **Redis Cluster**
  - Redis cluster được thiết kế tự động phân chia dữ liệu giữa các instance. Redis cluster chỉ dùng 1 process để chạy nhưng sử dụng 2 port. Low port để xử lý request từ client, high port (= low port +10000) được sử dụng như là bus sử dụng binary protocal cho việc giao tiếp giữa các node trong cluster. Các node trong cluster có nhiệm vụ lưu trữ dữ liệu, nắm bắt trạng thái của cluster, bao gồm cả việc maping key vào đúng node. Cluster cũng tự động phát hiện ra các node mới, các node bị lỗi và promote slave thành master khi master lỗi. Các node sử dụng gossip để truyền các thông tin về cluster. Cluster hỗ trợ tối đa 15 master, 1000 node.
  - Các node trong cluster được kết nối full mesh với nhau qua TCP. Dữ liệu được phân chia ra các master và được nhân bản sang các slave. Một cluster thường gồm ít nhất 3 master, Mỗi master có ít nhất một slave để đảm bảo dự phòng. Với Redis cluster, khi master lỗi, slave được chuyển thành master, việc này không diễn ra ngay lập tức nên dịch vụ sẽ bị ảnh hưởng. Khi kết nối tới master, client cũng phải được chỉ rõ là kết nối đến cluster.
  - Master cho phép đọc ghi dữ liệu, slave chỉ cho phép đọc dữ liệu.
  - Phân chia dữ liệu trong cluster:
    - Hash slots
      - Cluster sử dụng phương thức hashslot để phân chia dữ liệu trong cluster. Mỗi cluster có 16384 slot. Mỗi một master sẽ là một tập hợp của hashslot này và được cấu hình bởi người dùng. Nếu một master không có slot nào, nó sẽ không lưu trữ dữ liệu, request sẽ được redirect đến các master khác. Vị trí của master lưu trữ data được xác định bởi công thức:

HASH_SLOT = CRC16(key) mod 16384 (CRC16(key) sẽ chuyển key thành integer)

- Các hash slot được di chuyển thủ công giữa các master mà không cần stop dịch vụ.

- Hash tags
  - Với multi-key, yêu cầu toàn bộ các key phải được lưu trữ trên cùng 1 node. Việc này sẽ do Hash tag thực thi trong cluster theo rule sau:
    1. Nếu key chứa ký tự {
    2. Key chứa ký tự } sau ký tự {
    3. Có ký giữa 2 ký tự "{" đầu tiên và "}" đầu tiên.
  - Các partern giống nhau sẽ được hash và xếp vào cùng 1 hashslot.
    1. {user1000}.following và {user1000}.followers được xếp vào cùng hashlot.
    2. foo{}{bar}: Không có dữ liệu nào giữa cặp {} đầu tiên => cả key được hash.
    3. foo{{bar}}: {bar được hash vì nó nằm giữa { đầu tiên và } đầu tiên.
    4. foo{bar}{zap}: bar sẽ được hash.
  - Hash sẽ dừng lại sau cặp "{" và "}" đầu tiên, vì vậy nếu không có dữ liệu giữa "{" và "}" đầu tiên thì cả key đó sẽ được hash.

- **Hoạt động của cluster**
  - **Các thuộc tính của node**
    - Mỗi node trong cluster có một tên duy nhất và được diễn tả dưới dạng hexa của 160 bit ngẫu nhiên. Nó sẽ lưu lại tên này như là 1 ID vào file config. Chỉ bị thay đổi nếu bị xóa file config và hard reset cluster. Node ID này được dùng để xác định node mạng trong cluster (không sử dụng IP, vì vậy IP của node có thể thay đổi mà không ảnh hưởng gì). Ngoài node ID, mỗi node còn có các thông tin đi kèm khác như IP, port, tập hợp các cờ (cờ master/slave, last time ping/pong, tập hợp hashslot mà nó phục vụ).
  - **Cluster topology**
    - Các node được kết nối full mesh qua kết nối TCP. Mỗi node có N-1 TCP incoming và N-1 TCP outgoing (giả sử có N node

trong cluster). Các kết nối TCP này được duy trì. Nếu một bản tin ping gửi đi mà không nhận được bản tin pong gửi về trong 1 khoảng thơi gian, nó sẽ coi node mạng được ping là không kết nối được và sẽ thực hiện kết nối lại. Các node sử dụng Gossip để tránh việc trao đổi quá nhiều thông tin.

- Các node trong cluster có khả năng tự phát hiện các node khác trong cluster nếu node đó được cluster trusted thông qua gossip. Mỗi khi có bản tin nhận được nó sẽ phản hồi các trusted node.

- **Redirection**
  - Các client có thể truy vấn đến mọi node mạng trong cluster. Khi node nhận được query, nó sẽ kiểm tra node nào đang chứa hashslot. Nếu chính nó quản lý, nó sẽ xử lý và gửi lại thông tin. Nếu nó không quản lý, nó sẽ kiểm tra xem node mạng nào quản lý và trả về cho client với lỗi Move error cùng với địa chỉ IP/port mà đang chứa hashslot và cụ thể là hashlot nào. Dựa vào trả về lỗi này, client gửi đến đúng node đang chứa dữ liệu. Client cũng nên được cấu hình hashlot nào được phục vụ bởi node nào. và khi có nhiều bản tin MOVE thì cần cập nhật lại maping này.
- **Resharding**
  - Cluster hỗ trợ việc thêm mới, bỏ node mạng ra khỏi cluster bằng cách di chuyển hashslot từ node này sang node khác mà không ảnh hưởng đến hoạt động của cluster. Các lệnh được sử dụng trong resharding:
    - ADDSLOTS/ DELSLOTS: Được sử dụng để add/remove nhiều hashslot tới 1 node.
    - CLUSTER SETSLOT slot NODE node: Chỉ định 1 slot được gán vào node nào
    - CLUSTER SETSLOT slot MIGRATING node: Node nguồn sẽ tiếp nhận tất cả query về slot này. Nếu tồn tại key nó sẽ xử lý bình thường, nếu như không tồn tại nó sẽ trả về cho client mã –ACK với node đích được migrate. Trong trường hợp mutiple key nó sẽ trả về lỗi cho đến khi mutiple key hoàn thành chuyển sang node đích (MOVE và ACK: Khi nhận dược MOVE, Client sẽ update lại maping giữa hashslot và cặp IP mới. Khi nhận được ACK, client chỉ tạm thời chuyển câu

query đến node mới, query tiếp vào hashslot này vẫn chuyển vào node cũ).

- CLUSTER SETSLOT slot IMPORTING node: Node chỉ sử lý query nếu có thêm điều kiện ASKING. Nếu không có ASKING nó sẽ trả về bản tin có MOVE.

- **Client Redis**
  - Dummy client: Lựa chọn ngẫu nhiên 1 trong các node của cluster để thực hiện truy vấn. Sẽ redirect nếu có yêu cầu.
  - Smart client**:** Sẽ cache lại thông tin hashslot-node để truy vấn đúng node mạng chứa key. Client sẽ cập nhật thông tin qua 3 cách:
    - Khi start up, nó sẽ khởi tạo cấu hình hashslot –node
    - Khi client nhận được MOVE sẽ cập nhật lại hashslot-node
    - Client thực hiện lệnh CLUSTER SLOTS để khởi tạo lại cấu hình hashlot-node.

d. So sánh và đánh giá hiệu năng (Redis và MySQL)

- Cấu hình máy ảo: 16 vCPU, 16GB RAM
- Cấu trúc bảng trong MySQL (sử dụng code để generate ra 100 triệu bản ghi sau đó import vào MySQL bằng câu lệnh “load data local infile”)

```
mysql> load data local infile '/var/lib/mysql/customer.tsv' into table customer;
Query OK, 100000001 rows affected, 65535 warnings (22 min 5.37 sec)
Records: 200000000  Deleted: 0  Skipped: 99999999  Warnings: 300000000

mysql> desc customer;
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| cust_id    | varchar(20) | NO   | PRI | NULL    |       |
| cust_name  | varchar(20) | NO   |     | NULL    |       |
| birth_date | varchar(20) | NO   |     | NULL    |       |
| address    | varchar(20) | NO   |     | NULL    |       |
+------------+-------------+------+-----+---------+-------+
4 rows in set (0.01 sec)
```

- Sử dụng lệnh để import bảng từ MySQL vào Redis:
  - *mysql -uroot -p test *
    --skip-column-names --raw < data.sql | redis-cli -p 6379 –pipe

```
[root@localhost ~]# redis-cli -p 6379 info | grep -A 1 Keyspace
# Keyspace
db0:keys=100000001,expires=0,avg_ttl=0
```

  - Nội dung file data.sql:
    SELECT CONCAT(
    *"*3\r\n",*
    '$',LENGTH(redis_cmd),'\r\n',redis_cmd,'\r\n',

```
    '$',LENGTH(redis_key),'\r\n',redis_key,'\r\n',
    '$',LENGTH(redis_birth),'\r\n',redis_birth,'\r'
  ) FROM (
    SELECT 'SET' as redis_cmd,
    cust_id as redis_key,
    birth_date as redis_birth
    FROM customer
  ) AS cust
```

- So sánh thời gian:

	MySQL	Redis
Thời gian import 100tr bản ghi	**22 phút**	**16 phút**
Thời gian select 1 bản ghi	**5ms**	**4ms**
Thời gian insert 1 bản ghi	**237ms**	**8ms**
Thời gian delete 1 bản ghi	**147ms**	**5ms**
Thời gian select 100000 bản ghi	**427ms**	**3.52s**

- Hình ảnh chạy thực tế khi chạy chương trình code:
  - MySQL

```
enter Db_Port:
33060
enter Db_Name:
test
enter Db_User:
root
enter Db_Pass:
Viettel@123
enter SQL query:
select * from tbl_test where 1 = 1 limit 100000
Executed time of query : 429 ms
enter Db_Port:
33060
enter Db_Name:
test
enter Db_User:
root
enter Db_Pass:
Viettel@123
enter SQL query:
select * from tbl_test limit 1
Executed time of query : 5 ms
```

```
enter Db_Port:
33060
enter Db_Name:
test
enter Db_User:
root
enter Db_Pass:
Viettel@123
Enter number to choose option:
0. Exit
1. Select
2. Insert
3. Delete
2
enter SQL query:
insert into customer (cust_id, name, birth_date, address) values ('100000004', 'Nguyen Van B', '19750430', 'Ho Chi Minh')
Executed time of query : 237 ms
Enter number to choose option:
0. Exit
1. Select
2. Insert
3. Delete
3
enter SQL query:
delete from customer where cust_id = '100000002'
Executed time of query : 147 ms
```

- Redis

```
-> Connection to server sucessfully.
Enter number to choose option:
0. Exit
1. Get
2. Set
3. Delete
1
enter key:
1
key:1 - value : Nguyen Tuan Anh 19930313 Ha Noi
Executed time of query : 4 ms
Enter number to choose option:
0. Exit
1. Get
2. Set
3. Delete
2
enter key:
2
enter value:
Pham Tuan Anh 19920604 Ha Noi
Set key:2 -> value :Pham Tuan Anh 19920604 Ha Noi success.
Executed time of query : 8 ms
Enter number to choose option:
0. Exit
1. Get
2. Set
3. Delete
3
enter key:
2
Executed time of query : 5 ms
```