Key-Value Databases

Chapter 14

Michael Tsiang

Stats 167: Introduction to Databases

UCLA

UCLA

Key-Value Databases

Redis

Key-Value Databases

Key-Value Databases

The simplest NoSQL database is the key-value database.

A **key-value database** (or **key-value store**) stores data in key-value pairs. In particular, a key-value database has:

- A set of identifying data objects called **keys**.
- For each key, there is exactly one associated data object, the **value** for that key.

Specifying a key queries the associated value in the database.

A key-value database is analogous to a dictionary in Python or a hash table in other programming languages.

Core Key-Value Operations

Key-value databases do not have a universal query language like SQL, but they all share a core set of three CRUD-style operations to add/update, remove, and retrieve key-value pairs:

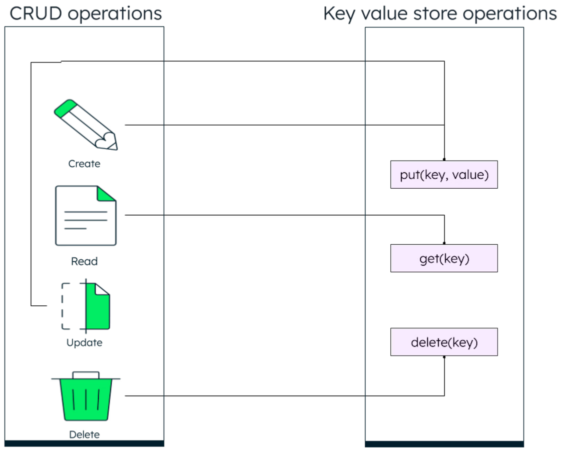
Operation	Purpose	Example
`put`	Insert or update a value for a given key	`put(key, value)`
`get`	Retrieve the value for a given key	`get(key)`
`delete`	Delete a key and its value	`delete(key)`

Some key-value databases, such as Redis, use **`set`** instead of `put`.

Side Note: The `put` operation is sometimes called an **upsert**, because it performs both an update (if the key already exists) and an insert (if the key does not).

CRUD Operations for Key-Value Databases

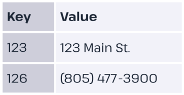
A schematic of CRUD-style operations in key-value databases:



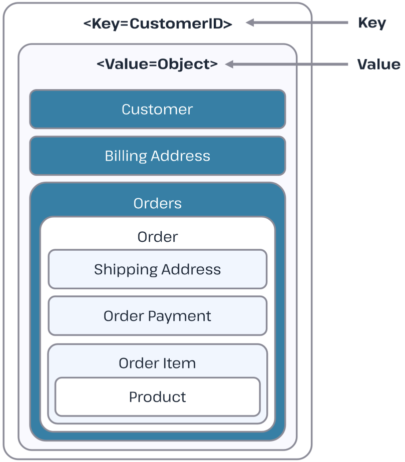
Source: MongoDB article "What is a Key Value Database?"

Examples of Key-Value Databases

Some examples of key-value databases:



Key	Value
123	123 Main St.
126	(805) 477-3900



Source: Hazelcast article “What is a key-value store?”

Flexible, Fast, and Scalable

Because of its simplicity, key-value databases is likely the most flexible type of NoSQL database.

The database is **schemaless**: There is no database-level restriction on the stored data objects. Values can be strings, blobs (i.e., raw binary data), JSON objects, or other arbitrary formats.

There are no relations between keys or values and thus no referential integrity to be checked, so reads and writes are extremely fast, and the database is massively scalable (easy to shard or partition).

Many key-value databases are stored in-memory (i.e., on RAM rather than a hard drive) for even faster access and processing speed.

When is this type of flexible, fast, and scalable database useful?

Key-Value Database Use Cases

Some examples of use cases for key-value databases:

- User session data (e.g., on web applications and massively multiplayer online games)
- User profile information (or other user-specific data)
- Shopping carts for online retailers
- Real-time leaderboards
- Real-time recommendations or ads
- Caching frequently accessed data

When Not to Use

Even though key-value databases are able to process large amounts of data quickly, the simple structure limits the queries and operations that can be performed on them.

Some scenarios in which a key-value database would not be suitable:

- Relational data

  There are no relationships stored between key-value pairs.

- Queries using data stored in values

  Data objects are stored as a whole in the value part of key-value pairs, so queries cannot be made based on the data.

- Complex queries (e.g., joins, filters, aggregations) or analytics

  Key-value databases only support basic queries (based on the key).

Redis

What is Redis?

Redis (https://redis.io/) is an open source, in-memory key-value database. In fact, Redis has been the most popular key-value database for over a decade.

Redis markets itself as a key-value “data structure store”: While Redis stores all data as key-value pairs, it allows for different types of values beyond plain strings, adding to its usability and popularity. It is often used as a database, cache, or message broker.

Side Note: Redis stands for “REmote DIctionary Server”.

Basic Redis Commands

Some basic Redis commands are shown below:

Command	Purpose	Example
`SET key val`	Set value of key	`SET user:1 "Leslie"`
`GET key`	Get value of key	`GET user:1`
`DEL key`	Delete key and value	`DEL user:1`
`EXISTS key`	Check if a key exists	`EXISTS user:1`
`INCR key`	Increment integer value (for counters)	`INCR page:views`
`EXPIRE key t`	Set key to expire after `t` seconds	`EXPIRE user:1 60`

The `MSET` and `MGET` commands (respectively) set and get multiple values at once.

Getting Started with Redis

There are a couple ways to get started with trying Redis for yourself.

Install Redis locally

- Installation guide (depends your on operating system)

Create a Redis Cloud account

- Sign-up link
- Quick start guide

Redis CLI

If Redis is installed and running locally, you can run **`redis-cli`** in a command line terminal to start the Redis command line interface:

```
redis-cli
```

The default host is `127.0.0.1` (localhost) and the default port is 6379, so `redis-cli` is equivalent to the command:

```
redis-cli -h 127.0.0.1 -p 6379
```

To log into a remote Redis server, use the command:

```
redis-cli -h host -p port -a password --user username
```

Use the keyword **`exit`** to close the CLI and return to the terminal.

Examples

```
redis-cli

SET employee:1:name "Leslie"
SET employee:1:job "Deputy Director"
SET employee:1:age 36
SET employee:1:dislikes "salads"

GET employee:1:name
GET employee:1:age

GET employee:1:dislikes
DEL employee:1:dislikes
GET employee:1:dislikes

INCR employee:1:age
GET employee:1:age
```

Key Naming

For clear and maintainable design, it is usually recommended to have a consistent system for naming keys.

A common convention is **`entity:id:attribute`**, with best practices:

- Use colons `:` as **namespace** (i.e., logical grouping) separators
- Use lowercase and `snake_case`
- Do not use overly long or time/random-based identifiers

SCAN and KEYS

The naming convention `entity:id:attribute` works well with **`SCAN`**, which uses a cursor to search over the keys, and **`KEYS`**, which lists all keys.

```
# Finds all keys that start with employee:1:
SCAN 0 MATCH employee:1:*
# Lists all keys that start with employee:1:
KEYS employee:1*
# Lists all keys (inefficient for large databases)
KEYS *
```

Some articles on naming conventions:

- Medium article y Yaman Nasser, “Unveiling the Art of Redis Key Naming Best Practices”, August 2023
- TheCodeBuzz article, “Redis Cache Best Practices for Development”
- Redis article, “Redis Namespace and Other Keys to Developing with Redis”

Redis Data Types

Since Redis is a key-value data structure store, it can support structured data types other than just strings.

Some common Redis data types:

- String
- Hash
- List
- Set
- Sorted Set

A full list of data types and basic commands for each can be found here: https://redis.io/docs/latest/develop/data-types/

Redis Hash Commands

Redis hashes are record types structured as collections of field-value pairs.

A collection of field-value pairs is stored under a single key, keeping related information together. For example, user profiles often have multiple fields (i.e., username, contact information, etc.) that are all under the same user.

Basic Redis hash commands:

- `HSET`: Sets multiple fields of the hash
- `HGET`: Retrieves a single field
- `HMGET`: Retrieves multiple fields
- `HGETALL`: Retrieves all fields
- `HINCRBY`: Increments a given field by a specified integer
- `HDEL`: Deletes specified fields

Redis Hash Examples

```
HSET employee:1 name "Leslie" job "Deputy Director" age 36
HGET employee:1 name
HMGET employee:1 name job
HGETALL employee:1
HINCRBY employee:1 age 5
HGET employee:1 age
HDEL employee:1 age
```

Notice that all field-value pairs are stored under the same `employee:1` key.

Connecting to Redis with R and Python

Redis supports multiple APIs to connect with its databases.

In particular:

- The `redux` package in R
- The `redis` library in Python

Redis with R

```r
library(redux)

# Connect to local or remote Redis
r <- hiredis()
# r <- hiredis(host = "hostname",
#              port = 6379,
#              password = "password")

r$SET("employee:1:name", "Leslie")
r$GET("employee:1:name")

r$HSET("employee:2", "name", "Ron")
r$HSET("employee:2", "age", "42")
r$HGET("employee:2", "name")
r$HGETALL("employee:2")
```

Note: If connecting to a local Redis database, make sure the database is running first (in Mac, run `brew services start redis`).

Redis with Python

```python
import redis

# Connect to local or remote Redis
r = redis.Redis(
   host='localhost',
   port=6379,
   decode_responses=True
)
# r = redis.Redis(
#    host='hostname',
#    port=6379,
#    password='password',
#    decode_responses=True
# )

r.set("employee:1:name", "Leslie")
r.get("employee:1:name")

r.hset("employee:2", mapping={"name": "Ron", "age": "42"})
r.hget("employee:2", "name")
r.hgetall("employee:2")
```

Further Learning

As simplistic as key-value databases are, there is still a lot more to Redis than we can cover here.

For more in-depth information on Redis:

- Redis documentation: https://redis.io/docs/latest/
- Redis learning tutorials: https://university.redis.io/academy

Tutorials:

- https://www.geeksforgeeks.org/introduction-to-redis-server/
- https://www.datacamp.com/tutorial/python-redis-beginner-guide

Newer extensions of Redis:

- RedisJSON turns Redis into a document database: https://redis.io/json/
- Redis as a vector database: https://redis.io/solutions/vector-database/