CHAPTER 9

pmemkv: A Persistent In-Memory Key-Value Store

Programming Persistent Memory is not easy. In several chapters we have described that application which takes advantage of persistent memory must take responsibility for atomicity of operations and consistency of data structures. PMDK libraries like `libpmemobj` are designed with flexibility and simplicity in mind. Usually, these are conflicting requirements, and one has to be sacrificed for the sake of the other. The truth is that in most cases API's flexibility increases its complexity.

In the current cloud computing ecosystem, there is an unpredictable demand for data. Consumers expect web services to provide data with predicable low-latency reliability. Persistent memory's byte-addressability and huge capacity characteristics make this technology a perfect fit for the broadly defined cloud environment.

Today, as greater numbers of devices with greater levels of intelligence are connected to various networks, businesses and consumers are finding the cloud to be an increasingly attractive option that enables fast, ubiquitous access to their data. Increasingly, consumers are fine with lower storage capacity on endpoint devices in favor of using the cloud. By 2020, we believe that more bytes will be stored in the public cloud than in consumer devices (Figure 9-1).

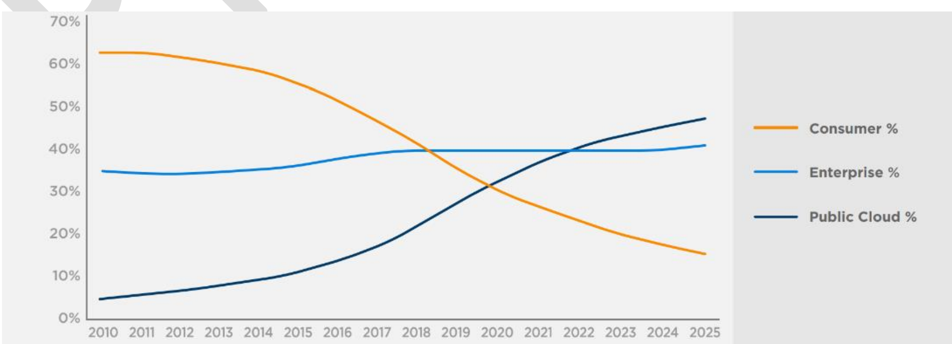

Figure 9-1. Where is data stored? Source: IDC White Paper – #US44413318

The cloud ecosystem, its modularity, and variety of service modes defines programming and application deployment as we know it. We call it cloud-native computing, and its popularity results in growing number of high level-languages, frameworks and abstraction layers. Figure 9-2 shows the fifteen most popular languages on GitHub based on pull requests.

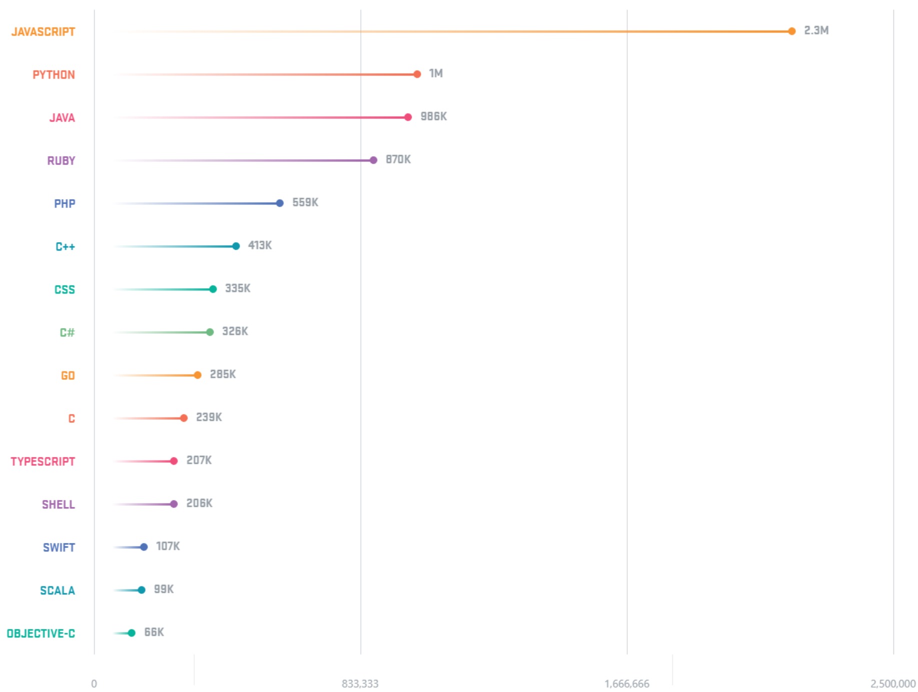

Figure 9-2. The fifteen most popular languages on GitHub by opened pull request (2017). Source: https://octoverse.github.com/2017/

In cloud environments, the platform is typically virtualized, and applications are heavily abstracted as to not make explicit assumptions about low-level hardware details. The question is: how to make programming persistent memory easier in cloud-native environment given the physical devices are local only to a specific server?

One of the answers is a key-value store. This data storage paradigm designed for storing, retrieving, and managing associative arrays with straightforward

API can easily utilize the advantages of persistent memory. This is why `pmemkv` was created.

pmemkv Architecture

There are many key-value datastores available on the market. They have different features, licenses, and their APIs are targeting different use cases. However, their core API remains the same. All of them provide methods like put, get, remove, exists, open and close. At the time we published this book, the most popular key-value data store is Redis. It is available in open source (https://redis.io/) and enterprise (https://redislabs.com) versions. DB-Engines (https://db-engines.com) shows that Redis has a significantly higher rank than any of its competitors in this sector.

Rank	Name	Score
1.	Redis	144,26
2.	Amazon DynamoDB	56,42
3.	Microsoft Azure Cosmos DB	29,08
4.	Memcached	27,07
5.	Hazelcast	8,27
6.	Aerospike	6,59
7.	Ehcache	6,56
8.	Riak KV	6,06
9.	OrientDB	5,69
10.	ArangoDB	4,66
11.	Ignite	4,26
12.	Oracle NoSQL	3,46
13.	InterSystems Caché	3,30
14.	LevelDB	3,29
15.	Oracle Berkeley DB	3,04

Figure 9-3. DB-Engines Ranking of Key-value Stores (July 2019). Scoring method: https://db-engines.com/en/ranking_definition Source: https://db-engines.com/en/ranking/key-value+store

Pmemkv was created as a separate project not only to complement PMDK's set of libraries with cloud-native support, but also to provide key-value API built for persistent memory. One of the main goals for pmemkv developers was to create friendly environment for open source community to develop new engines with the help of PMDK and to integrate it with other programming languages. Pmemkv uses the same BSD 3 Clause permissive license as PMDK. The native API of pmemkv is

C and C++. Other programming language bindings are available such as Javascript, Java, and Ruby. Additional languages can easily be added.

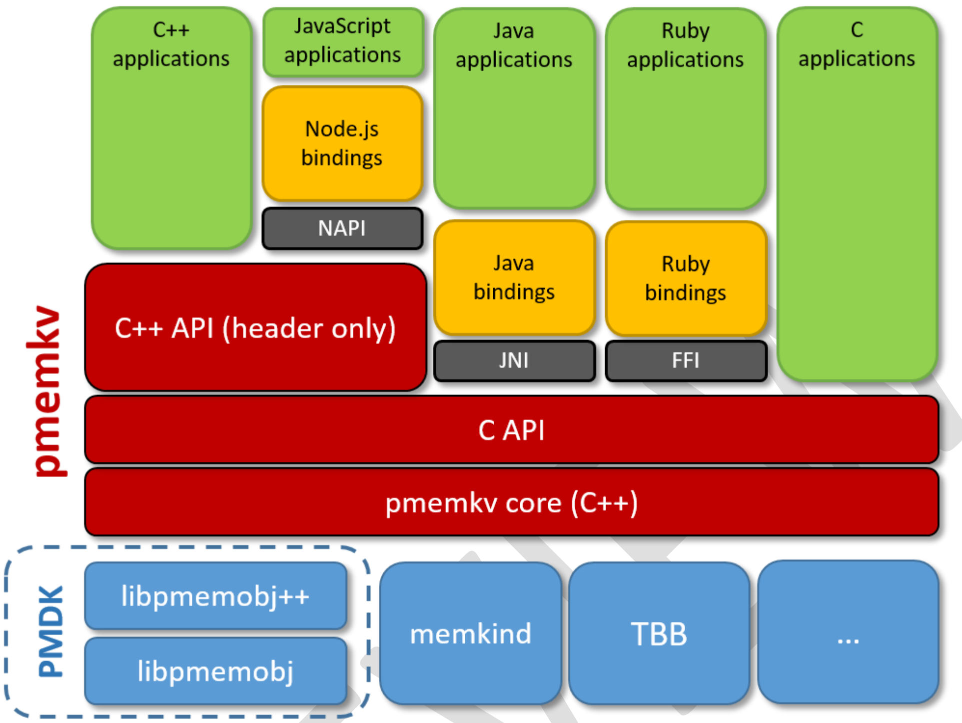

Figure 9-4. The architecture of pmemkv and programming languages support

The pmemkv API is similar to most key-value databases. Several storage engines are available for flexibility and functionality. Each engine has different performance characteristics and aims to solve different problems. Because of that, the functionality provided by each engine differs. They can be described by following characteristics:

- Persistence – persistent engines guarantees modifications are retained and power fail safe, while volatile ones keep its content only for the application lifetime.
- Concurrency – concurrent engines guarantees that some methods such as `get()`/`put()`/`remove()`, are thread-safe.
- Keys' ordering – "sorted" engines provide range query methods (like `get_above()`).

What makes pmemkv different from other key-value databases is that it provides direct access to the data. This means reading data from persistent memory does not require a copy into DRAM. This was already mentioned in Chapter 1 and is presented again in Figure 9-5.

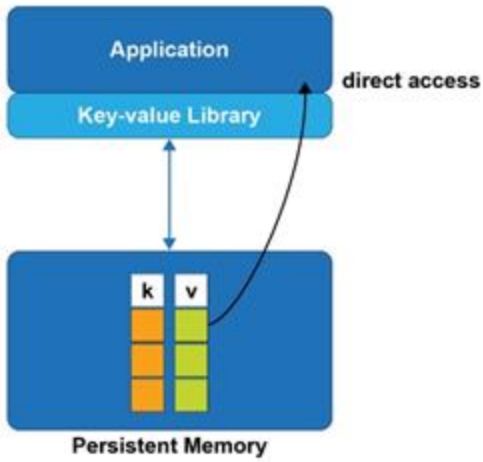

Figure 9-5. Applications directly accessing data in-place using pmemkv

Having direct access to the data significantly speeds up the application. This benefit is most noticeable in situations where the program is only interested in a part of the data stored in the database. In conventional approaches, this would require copying the whole data in some buffer and returning it to the application. With pmemkv, we provide the application a direct pointer and the application reads only as much as it is needed.

To make the API fully functional with various engine types, a flexible `pmemkv_config` structure was introduced. It stores engine's configuration options and allows to tune its behavior. Every engine has documented all supported config parameters. The pmemkv library was designed in a way that engines are pluggable and extendable to support the developers own requirements. Developers are free to modify existing engines or contribute new ones (https://github.com/pmem/pmemkv/blob/master/CONTRIBUTING.md#engines)

Listing 9-1 shows a basic setup of the `pmemkv_config` structure using the native C API. All the setup code is wrapped around the custom function, `config_setup()`, which will be used in a phonebook example in the next section. You can see also, how error handling is solved in pmemkv – all methods, except for `pmemkv_close()` and `pmemkv_errormsg()`, return a status. We can obtain error message using the `pmemkv_errormsg()` function. A complete list of return values can be found in pmemkv man page.

Listing 9-1. pmemkv_config.h – An example of the pmemkv_config structure using the C API

```
1    #include <cstdio>
2    #include <cassert>
3    #include <libpmemkv.h>
4
5    pmemkv_config* config_setup(const char* path, const uint64_t fcreate,
const uint64_t size) {
6        pmemkv_config *cfg = pmemkv_config_new();
7        assert(cfg != nullptr);
8
9        if (pmemkv_config_put_string(cfg, "path", path) !=
PMEMKV_STATUS_OK) {
10            fprintf(stderr, "%s", pmemkv_errormsg());
11            return NULL;
12        }
13
14        if (pmemkv_config_put_uint64(cfg, "force_create", fcreate) !=
PMEMKV_STATUS_OK) {
15            fprintf(stderr, "%s", pmemkv_errormsg());
16            return NULL;
17        }
18
19        if (pmemkv_config_put_uint64(cfg, "size", size) !=
PMEMKV_STATUS_OK) {
20            fprintf(stderr, "%s", pmemkv_errormsg());
21            return NULL;
22        }
23
24        return cfg;
25    }
```


- Line 5: We define custom function to prepare config and set all required params for engine(s) to use.
- Line 6: We create an instance of C config class. It returns `nullptr` on failure.
- Line 9-22: All params are put into config (the `cfg` instance) one after another (using function dedicated for the type) and each is checked if was stored successful (`PMEMKV_STATUS_OK` is returned when no errors occurred).

A Phonebook Example

Listing 9-3 shows a simple phonebook example implemented using the pmemkv C++ API v0.9. One of the main intentions of pmemkv is to provide a familiar API similar to the other key-value stores. This makes it very intuitive and easy to use. We will reuse the function from listing 9-1, which wraps configuration setup.

Listing 9-3. A simple phonebook example using the pmemkv C++ API

```
#include <iostream>
#include <cassert>
#include <libpmemkv.hpp>
#include <string>
#include "pmemkv_config.h"

using namespace pmem::kv;

auto PATH = "/daxfs/kvfile";
const uint64_t FORCE_CREATE = 1;
const uint64_t SIZE = 1024 * 1024 * 1024; // 1 Gig

int main() {
    // Prepare config for pmemkv database
    pmemkv_config *cfg = config_setup(PATH, FORCE_CREATE, SIZE);
    assert(cfg != nullptr);

    // Create a key-value store using the "cmap" engine.
    db kv;

    if (kv.open("cmap", config(cfg)) != status::OK) {
        std::cerr << db::errormsg() << std::endl;
        return 1;
    }

    // Add 2 entries with name and phone number
    if (kv.put("John", "123-456-789") != status::OK) {
        std::cerr << db::errormsg() << std::endl;
        return 1;
    }
    if (kv.put("Kate", "987-654-321") != status::OK) {
        std::cerr << db::errormsg() << std::endl;
```

```
            return 1;
        }

        // Count elements
        size_t cnt;
        if (kv.count_all(cnt) != status::OK) {
            std::cerr << db::errormsg() << std::endl;
            return 1;
        }
        assert(cnt == 2);

        // Read key back
        std::string number;
        if (kv.get("John", &number) != status::OK) {
            std::cerr << db::errormsg() << std::endl;
            return 1;
        }
        assert(number == "123-456-789");

        // Iterate through the phonebook
        if (kv.get_all([](string_view name, string_view number) {
                std::cout << "name: " << name.data() <<
                ", number: " << number.data() << std::endl;
                return 0;
                }) != status::OK) {
            std::cerr << db::errormsg() << std::endl;
            return 1;
        }

        // Remove one record
        if (kv.remove("John") != status::OK) {
            std::cerr << db::errormsg() << std::endl;
            return 1;
        }

        // Look for removed record
        assert(kv.exists("John") == status::NOT_FOUND);

        // Try to use one of methods of ordered engines
        assert(kv.get_above("John", [](string_view key, string_view value) {
            std::cout << "This callback should never be called" << std::endl;
            return 1;
        }) == status::NOT_SUPPORTED);

```

```
113         // Close database (optional)
114         kv.close();
115
116         return 0;
117     }
```

- Line 51: We set the `pmemkv_config` structure by calling `config_setup()` function introduced in previous section and listing (imported with `#include "pmemkv_config.h"`).
- Line 55: Creates a volatile object instance of the class `pmem::kv::db` which provides interface for managing persistent database.
- Line 57: Here, we open the key-value database backed by the *cmap* engine using the config parameters. The cmap engine is a persistent concurrent hashmap engine, implemented in `libpmemobj-cpp`. You can read more about *cmap* engine internal algorithms and data structures in Chapter 13.
- Line 58: The `pmem::kv::db` class provides a static `errormsg()` method for extended error messages. In this example, we use the `errormsg()` function as a part of the error handling routine.
- Line 63 & 67: The `put()` method inserts a key-value pair into the database. This function is guaranteed to be implemented by all engines. In this example, we are inserting two key-value pairs into database and compare returned statuses with `status::OK`. It's recommended way to check if function succeeded.
- Line 74: The `count_all()` has a single argument of type `size_t`. The method returns the number of elements (phonebook entries) stored in the database by the argument variable (`cnt`).
- Line 82: Here, we use the `get()` method to return the value of the "John" key. The value is copied into the user-provided `number` variable. The `get()` function returns `status::OK` on success, or an error on failure. This function is guaranteed to be implemented by all engines.
- Line 86: For this example, the expected value of variable `number` for "John" is "123-456-789". If we do not get this value, an assertion error is thrown.

- Line 89: The `get_all()` method used in this example gives the application direct, read-only access to the data. Both key and value variables are references to data stored in persistent memory. In this example, we simply print the name and the number of every visited pair.
- Line 99: Here, we are removing "John" and his phone number from the database by calling the `remove()` method. It is guaranteed to be implemented by all engines.
- Line 105: After removal of the pair "*John*, *123-456-789"*, we verify if the pair still exists in database. The API method `exists()` checks the existence of an element with given key. If the element is present, `status::OK` is returned, otherwise `status::NOT_FOUND` is returned.
- Line 108: Not every engine provides implementations of all the available API methods. In this example, we used the *cmap* engine, which is unordered engine type. This is why *cmap* does not support the `get_above()` function (and similarly: `get_below()`, `get_between()`, `count_above()`, `count_below()`, `count_between()`). Calling these functions will return `status::NOT_SUPPORTED`.
- Line 114: Finally, we are calling the `close()` method to close database. Calling this function is optional because `kv` was allocated on the stack and all necessary destructors will be called automatically, just like for the other variables residing on stack.

Bringing persistent memory closer to the cloud

We will rewrite the phonebook example using the Javascript language bindings. There are several language bindings available for pmemkv - JavaScript, Java, Ruby, Python. However, not all provide the same API functionally equivalent to the native C and C++ counterparts. Listing 9-4 shows an implementation of the phonebook application written using the pmemkv v0.9 using JavaScript language bindings API.

Listing 9-4. A simple phonebook example written using the JavaScript bindings for pmemkv v0.8

```
const Database = require('./lib/all');
```

```

function assert(condition) {
    if (!condition) throw new Error('Assert failed');
}

console.log('Create a key-value store using the "cmap" engine');
const db = new Database('cmap', '{"path":"/daxfs/kvfile","size":1073741824, "force_create":1}');

console.log('Add 2 entries with name and phone number');
db.put('John', '123-456-789');
db.put('Kate', '987-654-321');

console.log('Count elements');
assert(db.count_all == 2);

console.log('Read key back');
assert(db.get('John') === '123-456-789');

console.log('Iterate through the phonebook');
db.get_all((k, v) => console.log(`  name: ${k}, number: ${v}`));

console.log('Remove one record');
db.remove('John');

console.log('Lookup of removed record');
assert(!db.exists('John'));

console.log('Stopping engine');
db.stop();
```


The goal of higher-level pmemkv language bindings is to make programming persistent memory even easier and to provide a convenient tool for developers of cloud software.

Summary

In this chapter, we have shown how a familiar key-value datastore is an easy way for the broader cloud software developer audience to use persistent memory and directly access the data in-place. The modular design, flexible engine API, and integration with many of the most popular cloud programming languages makes pmemkv an intuitive choice for cloud-native software developers. As an open-source and lightweight library, it can easily be integrated into existing applications to immediately start taking advantage of persistent memory.

Some of the pmemkv engines are implemented using `libpmemobj-cpp` that we described in Chapter 8. The implementation of such engines provides real-world examples for developers to understand how to use PMDK (and related libraries) in applications.