What is diskv?

Diskv (disk-vee) is a simple, persistent key-value store written in the Go language. It starts with an incredibly simple API for storing arbitrary data on a filesystem by key, and builds several layers of performance-enhancing abstraction on top. The end result is a conceptually simple, but highly performant, disk-backed storage system.

Build Status

Installing

Install Go 1, either from source or with a prepackaged binary. Then,

```
$ go get github.com/peterbourgon/diskv
```

Usage

```go
package main

import (
    "fmt"
    "github.com/peterbourgon/diskv"
)

func main() {
    // Simplest transform function: put all the data files into the base dir.
    flatTransform := func(s string) []string { return []string{} }

    // Initialize a new diskv store, rooted at "my-data-dir", with a 1MB cache.
    d := diskv.New(diskv.Options{
        BasePath:     "my-data-dir",
        Transform:    flatTransform,
        CacheSizeMax: 1024 * 1024,
    })

    // Write three bytes to the key "alpha".
    key := "alpha"
    d.Write(key, []byte{'1', '2', '3'})

    // Read the value back out of the store.
    value, _ := d.Read(key)
    fmt.Printf("%v\n", value)

    // Erase the key+value from the store (and the disk).
    d.Erase(key)
}
```

More complex examples can be found in the "examples" subdirectory.

Theory

Basic idea

At its core, diskv is a map of a key (`string`) to arbitrary data (`[]byte`). The data is written to a single file on disk, with the same name as the key. The key determines where that file will be stored, via a user-provided `TransformFunc` , which takes a key and returns a slice (`[]string`) corresponding to a path list where the key file will be stored. The simplest TransformFunc,

```
func SimpleTransform (key string) []string {
    return []string{}
}
```

will place all keys in the same, base directory. The design is inspired by Redis diskstore; a TransformFunc which emulates the default diskstore behavior is available in the content-addressable-storage example.

Note that your TransformFunc should ensure that one valid key doesn't transform to a subset of another valid key. That is, it shouldn't be possible to construct valid keys that resolve to directory names. As a concrete example, if your TransformFunc splits on every 3 characters, then

```
d.Write("abcabc", val) // OK: written to <base>/abc/abc/abcabc
d.Write("abc", val)    // Error: attempted write to <base>/abc/abc, but it's a
directory
```

This will be addressed in an upcoming version of diskv.

Probably the most important design principle behind diskv is that your data is always flatly available on the disk. diskv will never do anything that would prevent you from accessing, copying, backing up, or otherwise interacting with your data via common UNIX commandline tools.

Adding a cache

An in-memory caching layer is provided by combining the BasicStore functionality with a simple map structure, and keeping it up-to-date as appropriate. Since the map structure in Go is not threadsafe, it's combined with a RWMutex to provide safe concurrent access.

Adding order

diskv is a key-value store and therefore inherently unordered. An ordering system can be injected into the store by passing something which satisfies the diskv.Index interface. (A default implementation, using Google's btree package, is provided.) Basically, diskv keeps an ordered (by a user-provided Less function) index of the keys, which can be queried.

Adding compression

Something which implements the diskv.Compression interface may be passed during store creation, so that all Writes and Reads are filtered through a compression/decompression pipeline. Several default implementations, using stdlib compression algorithms, are provided. Note that data is cached compressed; the cost of decompression is borne with each Read.

Streaming

diskv also now provides ReadStream and WriteStream methods, to allow very large data to be handled efficiently.

Future plans

- Needs plenty of robust testing: huge datasets, etc...
- More thorough benchmarking
- Your suggestions for use-cases I haven't thought of