DiscoDB Documentation

Release

Disco Project

March 08, 2017

Contents

1 Install 3

2 Example 5

3 Notes 7

4 Python API 9

5 See Also 11

discodb is comprised of a low-level data structure implemented in C, and a high-level `discodb.DiscoDB` class which exposes a dict-like interface for using the low-level data structure from Python. In contrast to Python's builtin dict object, DiscoDB can handle tens of millions of key-value pairs without consuming gigabytes of memory.

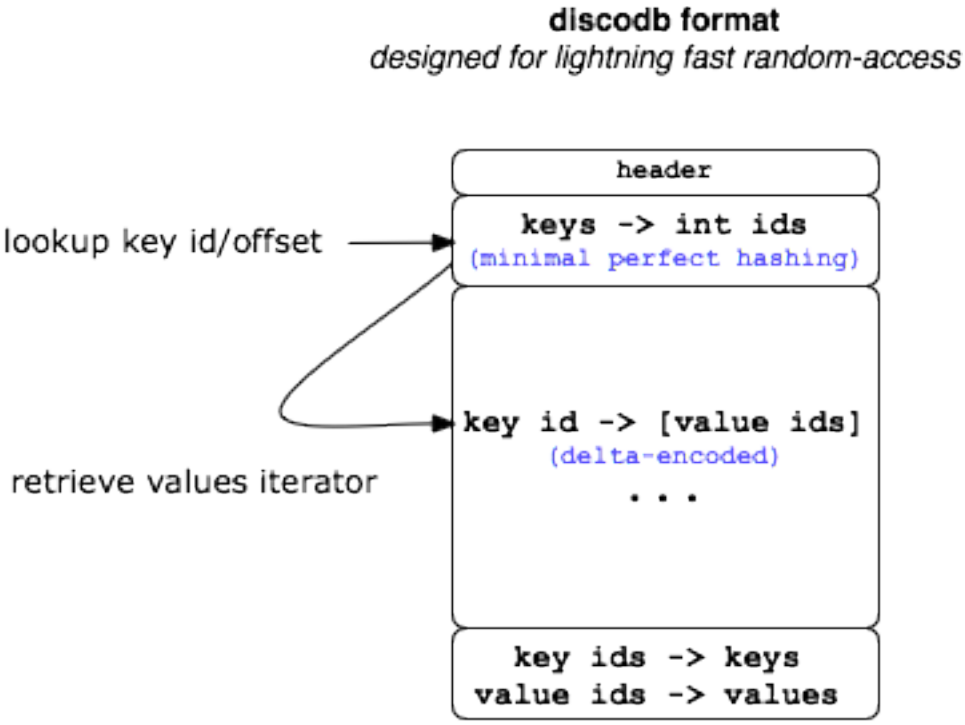
In addition to basic key-value mappings, DiscoDB supports evaluation of Boolean queries expressed in Conjunctive Normal Form (see `discodb.Q` below). All queries are evaluated lazily using iterators, so you can handle gigabytes of data in Python with ease.

DiscoDBs are persistent, which means that once created in memory, they can be easily compressed, serialized and written to a file. The benefit of this is that after they have been persisted, instantiating them from disk and key lookups are lightning-fast operations, thanks to memory mapping.

DiscoDBs are also immutable, which means that once they are created, they cannot be modified. A benefit of immutability is that the full key-space is known when DiscoDB is built, which makes it possible to use perfect hashing for fast *O(1)* key lookups. Specifically, DiscoDB relies on the CMPH library for building minimal perfect hash functions.

DiscoDB compresses values first by replacing duplicate entries with references to a singe unique entry and then by compressing unique entries with a fast compression algorithm based on Huffman Coding. The main benefit of this approach is that each value can be random accessed efficiently while achieving reasonable compression ratios, thanks to statistics collected from all the data. This means that you can have lots of redundancy, e.g. common prefixes, in your values without having to worry about space consumption.

The format of a DiscoDB file essentially looks like this:



The benefits of these properties are realized when you need repeated, random-access to data, especially when the dataset is too large to fit in memory at once. DiscoDBs are a key component in Disco's builtin distributed indexing system, discodex.

CHAPTER 1

Install

DiscoDB does not depend on Disco in any way, although it is a core component in discodex. You can use it without Disco as a general-purpose scalable, immutable datastructure for Python. To install only DiscoDB without rest of Disco, clone Disco, and run:

```
make install-discodb
```

or if you just want to build it locally:

```
cd contrib/discodb
python setup.py build
```

DiscoDB requires CMPH library v0.9 or newer (`libcmph-dev` in Debian).

CHAPTER 2

Example

Here is a simple example that builds a simple discodb and queries it:

```
from discodb import DiscoDB, Q

data = {'mammals': ['cow', 'dog', 'cat', 'whale'],
        'pets': ['dog', 'cat', 'goldfish'],
        'aquatic': ['goldfish', 'whale']}

db = DiscoDB(data) # create an immutable discodb object

print list(db.keys()) # => mammals, aquatic, pets
print list(db['pets']) # => dog, cat, goldfish
print list(db.query(Q.parse('mammals & aquatic'))) # => whale
print list(db.query(Q.parse('pets & ~aquatic'))) # => dog, cat
print list(db.query(Q.parse('pets | aquatic'))) # => dog, cat, whale, goldfish

db.dump(file('animals.db', 'w')) # dump discodb to a file
```

For more detailed information on querying, see `discodb.query`.

CHAPTER 3

Notes

- DiscoDB stores only one copy of each unique value. Thus using the same value multiple times is very cheap. Applications should utilize this feature to maximize space and time efficiency.
- DiscoDB sorts key-value pairs internally, so no pre-sorting is needed. You can request DiscoDB to remove duplicate key-value pairs automatically by setting `unique_items=True` in the DiscoDB constructor.
- DiscoDB does not compress values which are smaller than four bytes. DiscoDB may also decide to disable compression if it is not likely to be beneficial. This doesn't affect in DiscoDB behavior in any way. You can disable compression explicitly, e.g. if your values are already compressed, by setting `disable_compression=True` in the DiscoDB constructor.
- Value lists can be empty, in which case DiscoDB becomes an efficient set data structure.
- Keys and values can be arbitrary byte sequences or strings (see size limitations below). For instance, you can save a serialized DiscoDB, as produced by `discodb.DiscoDB.dumps()`, in another DiscoDB to build a tree of DiscoDBs!
- If any of the keys contains duplicate values, the whole DiscoDB is tagged to contain multisets. In the multiset mode `discodb.DiscoDB.query()` is not available, as it is not clear currently how duplicate values should be handled in queries. However, looking up a single key works as usual, so applications can freely utilize the multiset feature (which is very efficient, as noted above) if complex queries are not needed.
- **A single DiscoDB object has the following limitations:**
  - Maximum number of keys, 2^{32}
  - Maximum number of unique values, 2^{32}
  - Maximum number of values, 2^{64}
  - Maximum size of a key/value, 2^{32} bytes (4G)
  - Maximum size of a DiscoDB object, 2^{64} bytes

  In many cases it makes sense to have several (distributed) small or medium-size DiscoDBs with millions of keys at most, as created by discodex, instead of having a single DiscoDB with hundreds of millions of keys and values, although it is technically possible. Run your own benchmarks to find the optimal size for your application.

CHAPTER 4

Python API

CHAPTER 5

See Also