

Introduction to Kyoto Products

Mikio Hirabayashi

<hirarin@gmail.com>

Kyoto Cabinet

- database library -

Features

- **straightforward implementation**

- **key / value database**

- e.g.) DBM, NDBM, GDBM, TDB, CDB, Berkeley DB

- **simple library = process embedded**

- Successor of QDBM, sibling of Tokyo Cabinet

- **C++03 (with TR1) and C++0x portable**

- Linux, FreeBSD, Solaris, Mac OS X

- Windows

- **high performance**

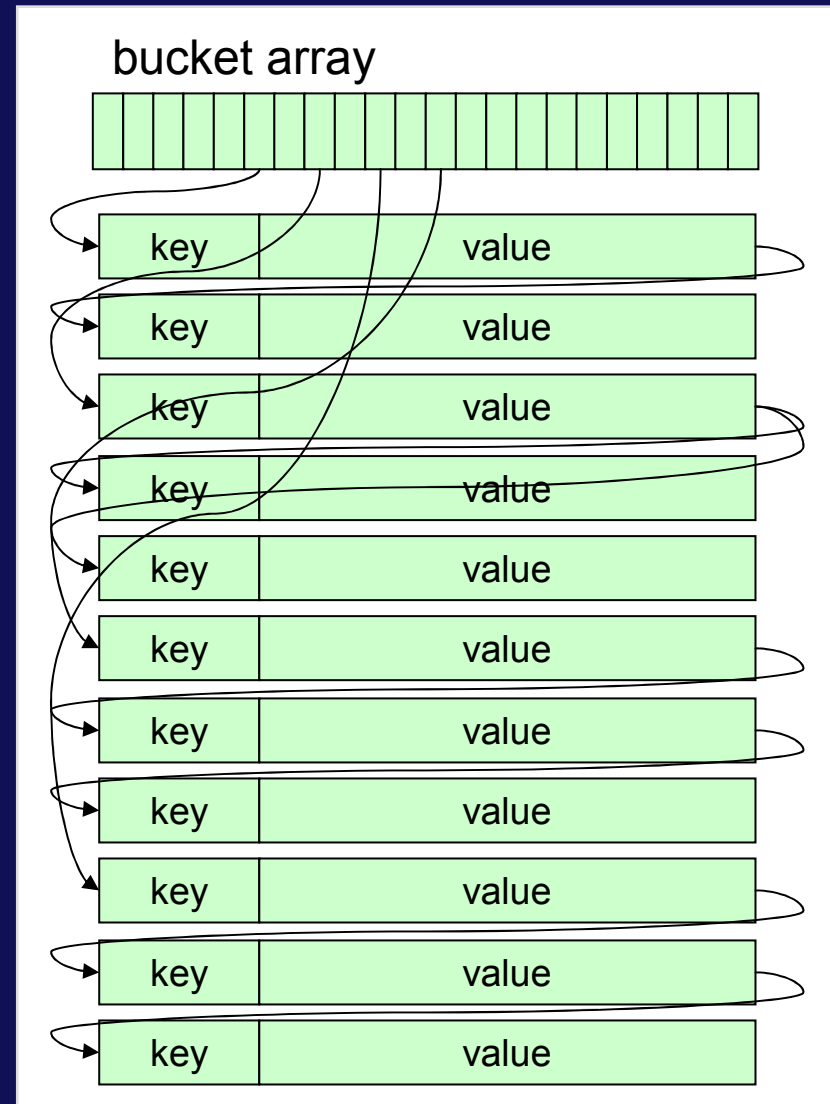
- **insert: 1.0 sec / 1M records (1,000,000 qps)**

- **search: 0.5 sec / 1M records (2,000,000 qps)**

- **high concurrency**
 - multi-thread safe
 - read/write locking by records
- **high scalability**
 - hash and B+tree structure = $O(1)$ and $O(\log N)$
 - no actual limit size of a database file (to 8 exabytes)
- **transaction**
 - write ahead logging and shadow paging
 - ACID properties
- **various APIs**
 - on-memory: hash table, binary search tree, LRU list
 - persistent file: hash table, B+ tree
- **script language bindings**
 - Ruby, and other languages (work in progress)
 - the "C" binding is also provided

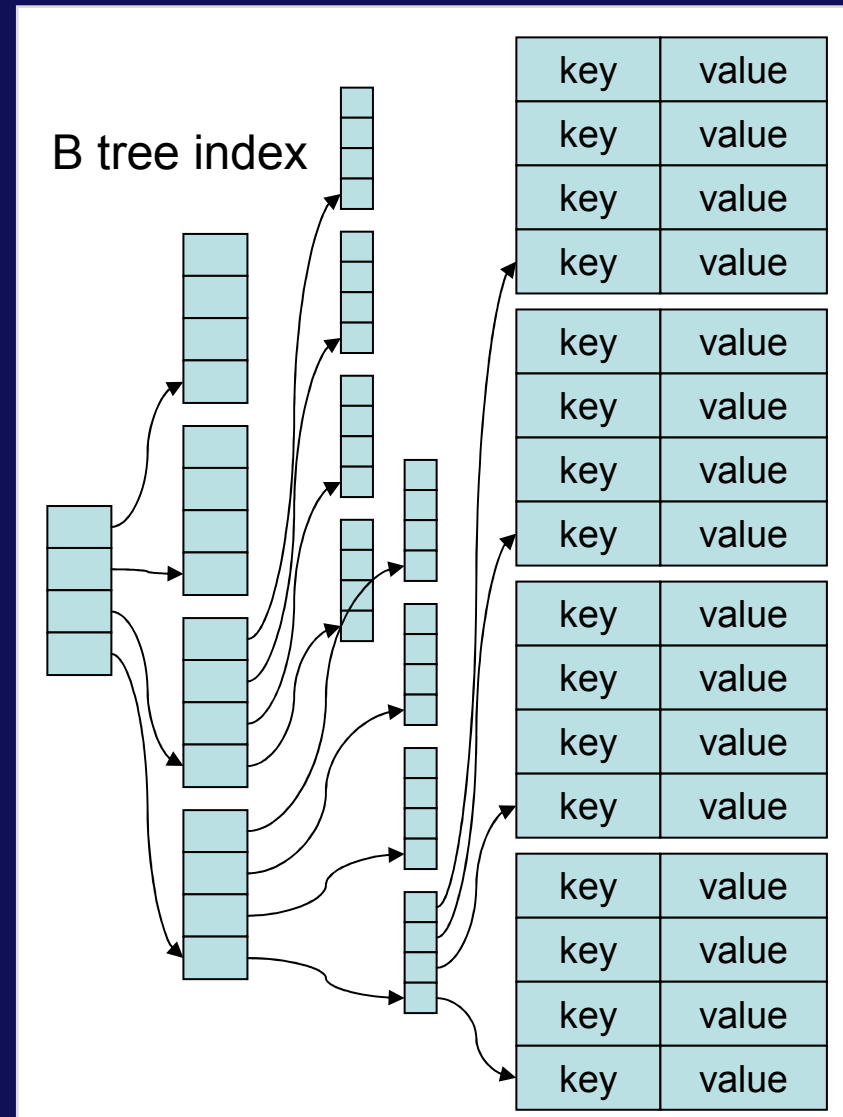
HashDB: File Hash DB

- **static hashing**
 - $O(1)$ time complexity
 - jilted dynamic hashing for simplicity and performance
- **separate chaining**
 - binary search tree
 - balances by the second hash
- **free block pool**
 - best fit allocation
 - dynamic defragmentation
- **combines mmap and pwrite/pread**
 - saves calling system calls
- **compression**
 - deflate(gzip) / custom



TreeDB: File B+ Tree DB

- **B+ tree**
 - $O(\log N)$ time complexity
- **page caching**
 - separated LRU lists
 - mid-point insertion
- **stands on hash DB**
 - records pages in hash DB
 - succeeds time and space efficiency
- **custom comparison function**
 - prefix / range matching
- **cursor**
 - jump / next / prev



On-memory Databases

- **ProtoDB: Prototype DB**

- DB wrapper for STL map
- any data structure compatible `std::map` are available
- ProtoHashDB: alias of `ProtoDB<std::unordered_map>`
- ProtoTreeDB: alias of `ProtoDB<std::map>`

- **CacheDB: Cache DB**

- hash table with double linked list
- constant memory usage
- LRU (least recent used) records are removed
- snapshot: dump/load current records with a file

Comparison among DB Types

class	ProtoHashDB	ProtoTreeDB	CacheDB	HashDB	TreeDB
persistence	volatile	volatile	volatile	persistent	persistent
algorithm	hash table	red black tree	hash table	hash table	B+ tree
complexity	$O(1)$	$O(\log N)$	$O(1)$	$O(1)$	$O(\log N)$
sequence	undefined	lexical order	undefined	undefined	custom order
lock unit	whole (rwlock)	whole (rwlock)	record (mutex)	record (rwlock)	page (rwlock)

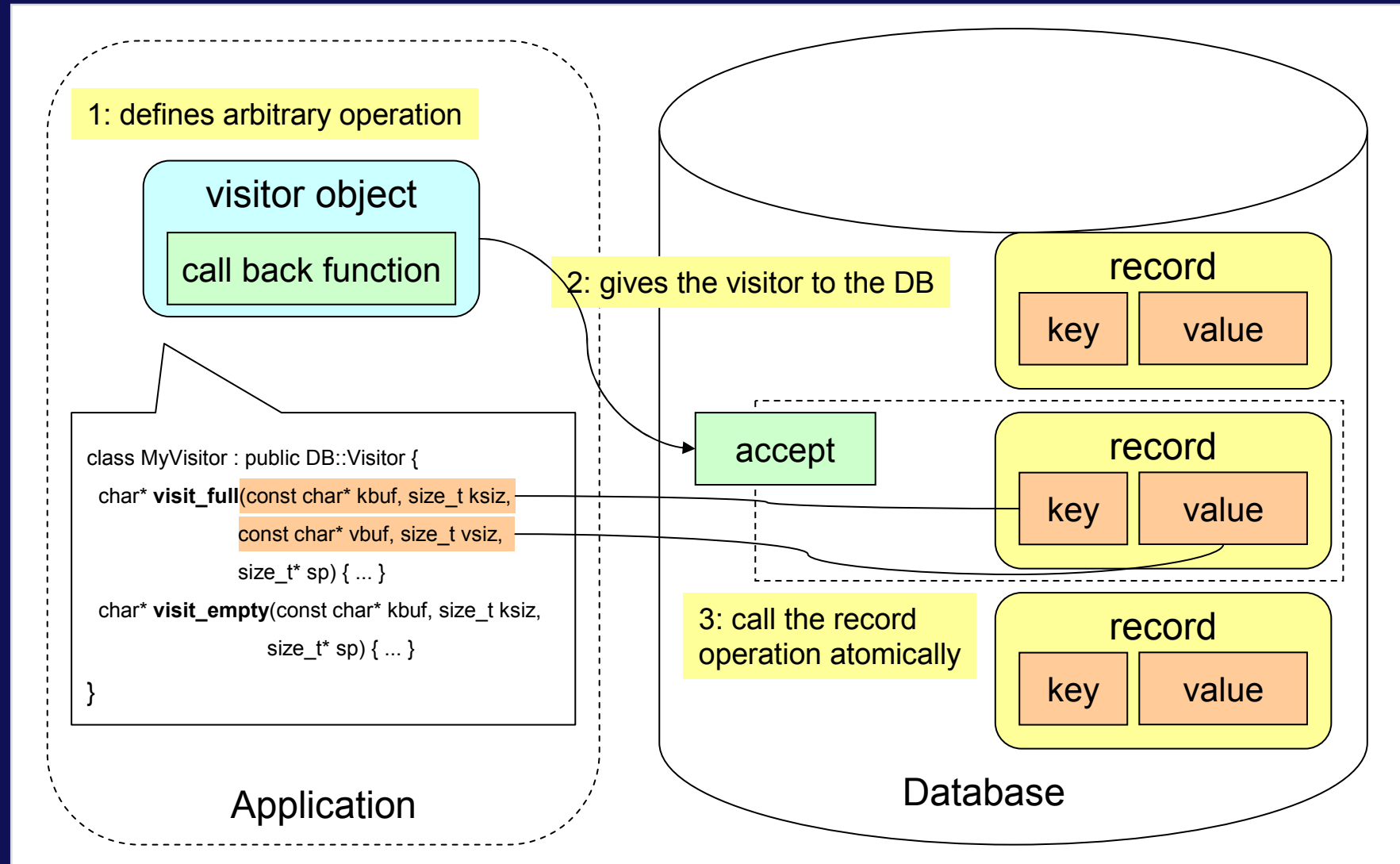
Class Hierarchy

- **DB = interface of record operations**
 - **FileDB** = interface of file operation, mix-in of utilities
 - ProtoHashDB, ProtoTreeDB, HashDB, TreeDB
 - PolyDB
- **PolyDB: polymorphic database**
 - dynamic binding to four DB types
 - "factory method" and "strategy" patterns
 - the concrete type is determined when opening
 - naming convention
 - ProtoHashDB: "-", ProtoTreeDB: "+", CacheDB: "*"
 - HashDB: "__kch", TreeDB: "__kct"

Abstraction of KVS

- **what is "Key Value Storage" ?**
 - each record consists of one key and one value
 - atomicity is assured for only one record
 - records are stored in persistent storage
- **so, what?**
 - every operation can be abstracted by "visitor" pattern
 - the database accepts one visitor in a record at the same time
 - lets him read/write the record arbitrary
 - saves the operated value
- **flexible and useful interface**
 - provides the "DB::accept" method which realizes whatever
 - "DB::set", "DB::get", "DB::remove", "DB::increment" are built-in as wrappers of the "DB::accept"

Visitor Interface



Comparison with Tokyo Cabinet

• Pros

- **space efficiency: smaller size of DB file**
 - footprint/record: TC=22B → KC=16B
- **parallelism: higher performance in multi-thread environment**
 - uses atomic operations such as CAS
- **portability: non-POSIX platform support**
 - supports Win32
- **usability: object-oriented design**
 - external cursor, generalization by the visitor pattern
- **robustness: auto transaction and auto recovery**

• Cons

- **time efficiency per thread: due to grained lock**
- **dependency on modern C++ implementation**

Example Code

```
#include <khashdb.h>
#include <iostream.h>

using namespace std;
using namespace kyotocabinet;

int main(int argc, char** argv) {

    // create the database object
    HashDB db;

    // open the database
    if (!db.open("casket.kch",
        HashDB::OWRITER | HashDB::OCREATE)) {
        cout << "open error: " << db.error().string() << endl;
    }

    // store records
    if (!db.set("foo", "hop") ||
        !db.set("bar", "step") ||
        !db.set("baz", "jump")) {
        cout << "set error: " << db.error().string() << endl;
    }

    // retrieve records
    string* value = db.get("foo");
    if (value) {
        cout << *value << endl;
        delete value;
    } else {
        cout << "get error: " << db.error().string() << endl;
    }
}
```

This example is available for all database types by switching the class name.

```
// traverse records
class Traverser : public DB::Visitor {
    const char* visit_full(const char* kbuf, size_t ksiz,
        const char* vbuf, size_t vsiz,
        size_t *sp) {
        cout << string(kbuf, ksiz) << ":" <<
            string(vbuf, vsiz) << endl;
        return NOP;
    }
} traverser;
db.iterate(&traverser, false);

// close the database
if (!db.close()) {
    cout << "close error: " << db.error().string() << endl;
}

return 0;
}
```

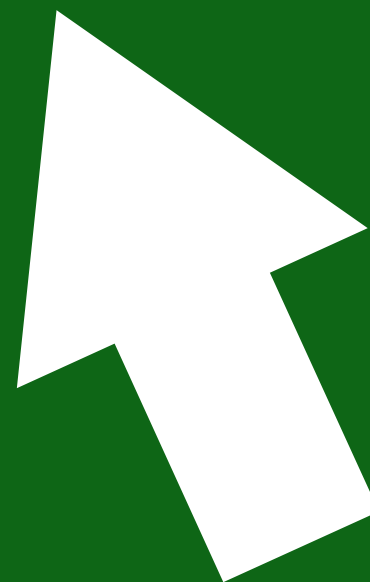
Other Kyoto Series?

- **Now, planning.**
- **Kyoto Tyrant?**
 - network service of KC
- **Kyoto Dystopia?**
 - full-text search engine on KC

maintainability is my paramount concern...

<http://1978th.net/>

京都



キャビネット

8 EiB