

Introduction to Kyoto Products

Kyoto Cabinet

- lightweight database library -

Features

- straightforward implementation of DBM
 - process-embedded database
 - persistent associative array = key-value storage
 - successor of QDBM, and sibling of Tokyo Cabinet
 - e.g.) DBM, NDBM, GDBM, TDB, CDB, Berkeley DB
 - C++03 (with TR1) and C++0x portable
 - supports Linux, FreeBSD, Mac OS X, Solaris, Windows
- high performance
 - insert 1M records / 0.8 sec = 1,250,000 QPS
 - search 1M records / 0.7 sec = 1,428,571 QPS

- **high concurrency**
 - multi-thread safe
 - read/write locking by records, user-land locking by CAS
- **high scalability**
 - hash and B+tree structure = $O(1)$ and $O(\log N)$
 - no actual limit size of a database file (up to 8 exa bytes)
- **transaction**
 - write ahead logging, shadow paging
 - ACID properties
- **various database types**
 - storage selection: on-memory, single file, multiple files in a directory
 - algorithm selection: hash table, B+ tree
- **other language bindings**
 - C, Java, Python, Ruby, Perl, Lua, and so on

Database Types

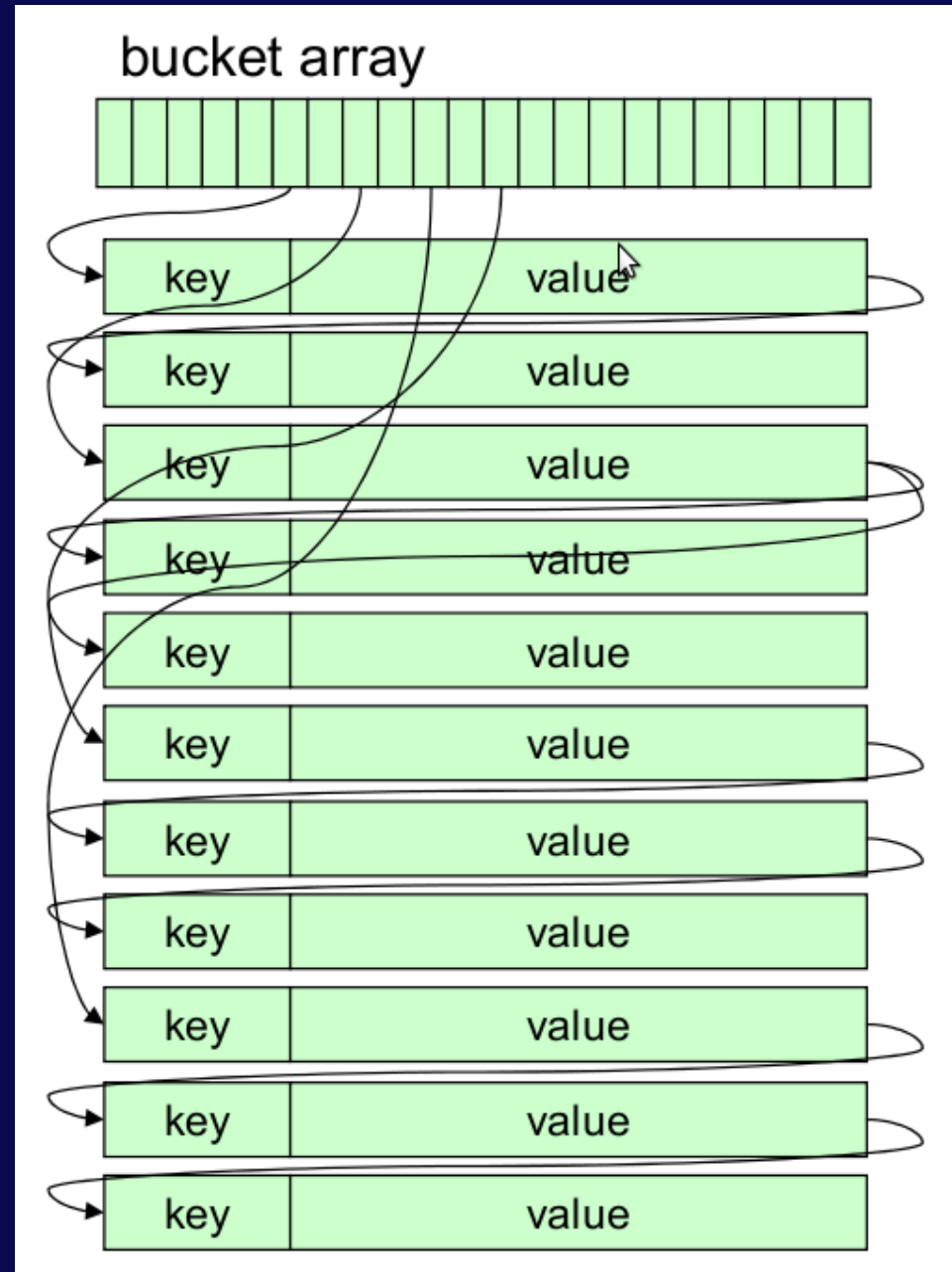
- **ProtoHashDB: prototype hash database**
 - on-memory database implemented with `std::unordered_map`
- **ProtoTreeDB: prototype tree database**
 - on-memory database implemented with `std::map`
- **CacheDB: cache hash database**
 - on-memory database featuring LRU deletion = typical cache
- **GrassDB: cache tree database**
 - on-memory database of B+ tree = cache with order
- **HashDB: file hash database**
 - file database of hash table = typical DBM
- **TreeDB: file tree database**
 - file database of B+ tree = DBM with order
- **DirDB: directory hash database**
 - respective files in a directory of the file system = for large records
- **ForestDB: directory tree database**
 - directory database of B+ tree = huge DBM with order

performance characteristics

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

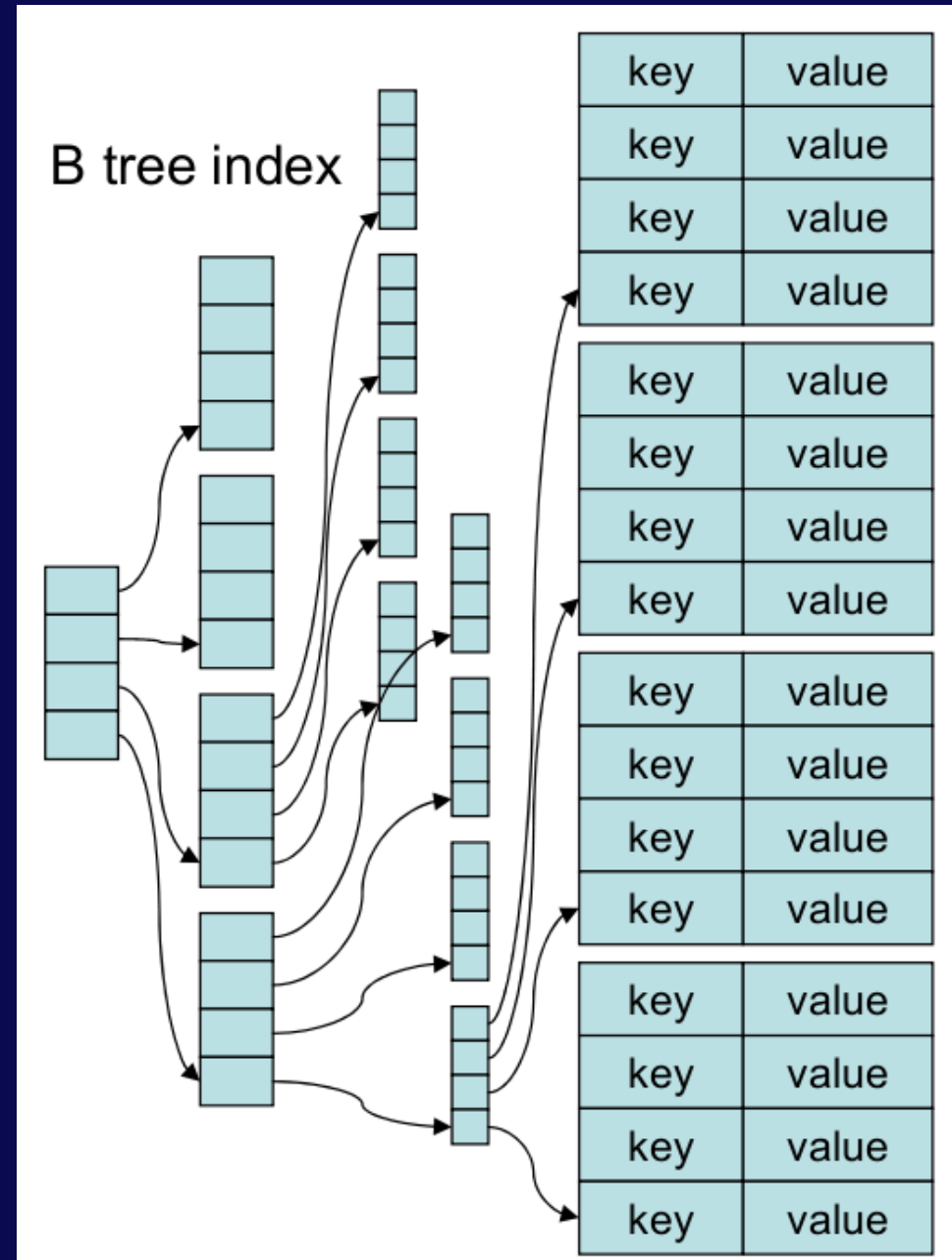
HashDB: File Hash Database

- **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 pread/pwrite**
 - saves calling system calls
- **tuning parameters**
 - bucket number, alignment
 - compression



TreeDB: File Tree Database

- **B+ tree**
 - $O(\log N)$ time complexity
 - custom comparison function
- **page caching**
 - separate LRU lists
 - mid-point insertion
- **stands on HashDB**
 - stores pages in HashDB
 - succeeds time and space efficiency
- **cursor**
 - jump, step, step_back
 - prefix/range matching



HashDB vs. TreeDB

	HashDB (hash table)	TreeDB (B+ tree)
time efficiency	faster, $O(1)$	slower, $O(\log N)$
space efficiency	larger, 16 bytes/record footprint	smaller, 3 bytes/record footprint
concurrency	higher, locked by the record	lower, locked by the page
cursor	undefined order jump by full matching, able to step back	ordered by application logic jump by range matching, unable to step back
fragmentation	more incident	less incident
random access	faster	slower
sequential access	slower	faster
transaction	faster	slower
memory usage	smaller	larger
compression	less efficient	more efficient
tuning points	number of buckets, size of mapped memory	size of each page, capacity of the page cache
typical use cases	session management of web service user account database document database access counter cache of content management system graph/text mining	job/message queue sub index of relational database dictionary of words inverted index of full-text search temporary storage of map-reduce archive of many small files

Directory Databases

- **DirDB: directory hash database**
 - respective files in a directory of the file system
 - suitable for large records
 - strongly depends on the file system performance
 - ReiserFS > EXT3 > EXT4 > EXT2 > XFS > NTFS
- **ForestDB: directory tree database**
 - B+ tree stands on DirDB
 - reduce the number of files and alleviates the load
 - most scalable approach among all database types

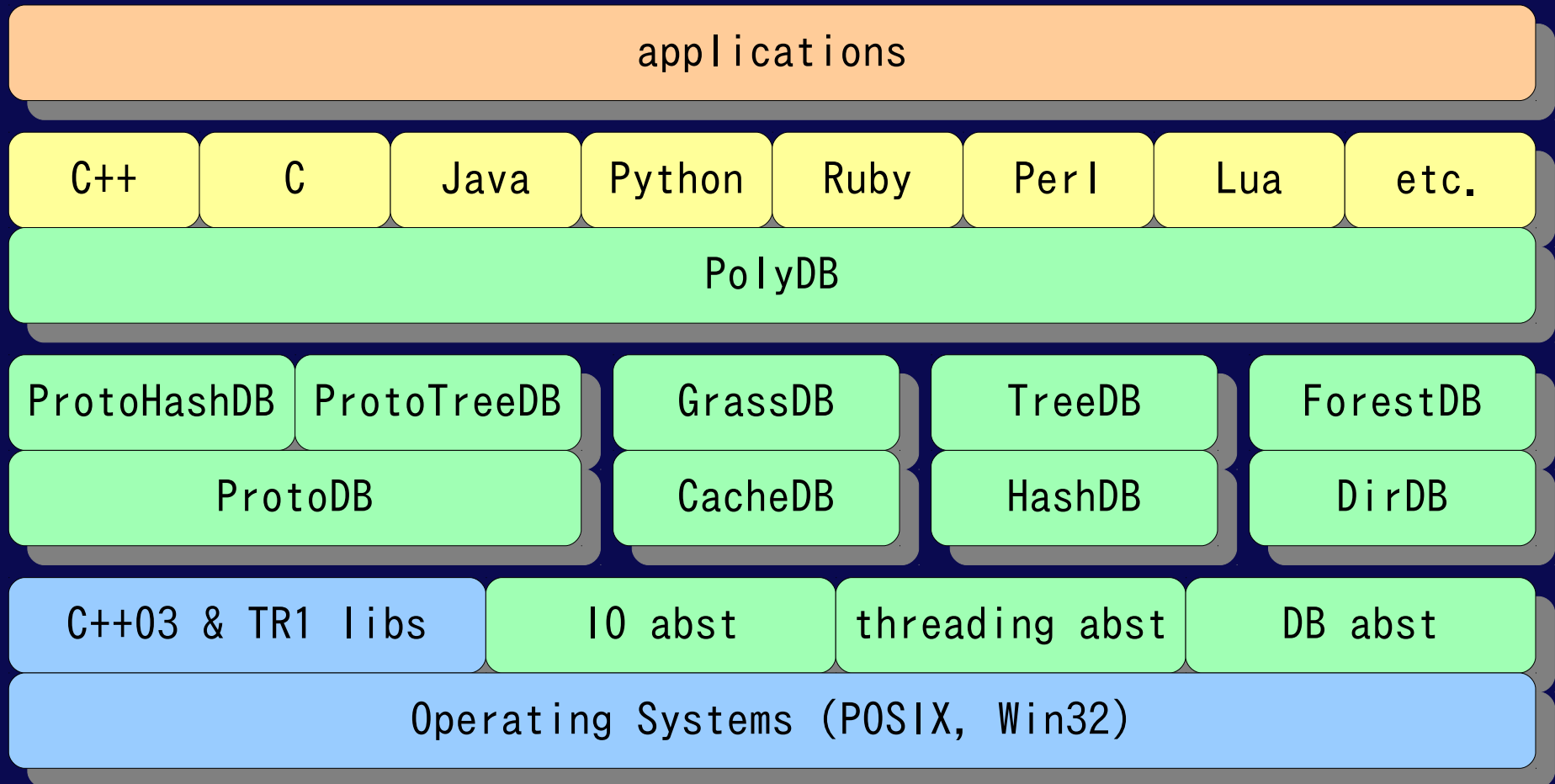
On-memory Databases

- **ProtoDB: prototype database template**
 - 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 hash database**
 - hash table with double linked list
 - constant memory usage
 - LRU (least recent used) records are removed
 - much less memory usage than `std::map` and `std::unordered_map`
- **GrassDB: cache tree database**
 - B+ tree stands on CacheDB
 - much less memory usage than CacheDB

Class Hierarchy

- **DB = interface of record operations**
 - BasicDB = interface of storage operations, mix-in of utilities
 - ProtoHashDB, ProtoTreeDB, CacheDB, HashDB, TreeDB, ...
 - PolyDB
- **PolyDB: polymorphic database**
 - dynamic binding to concrete DB types
 - "factory method" and "strategy" patterns
 - the concrete type is determined when opening
 - naming convention
 - ProtoHashDB: "-", ProtoTreeDB: "+", CacheDB: "*", GrassDB: "%"
 - HashDB: ".kch", TreeDB: ".kct", DirDB: ".kcd", ForestDB: ".kcf"

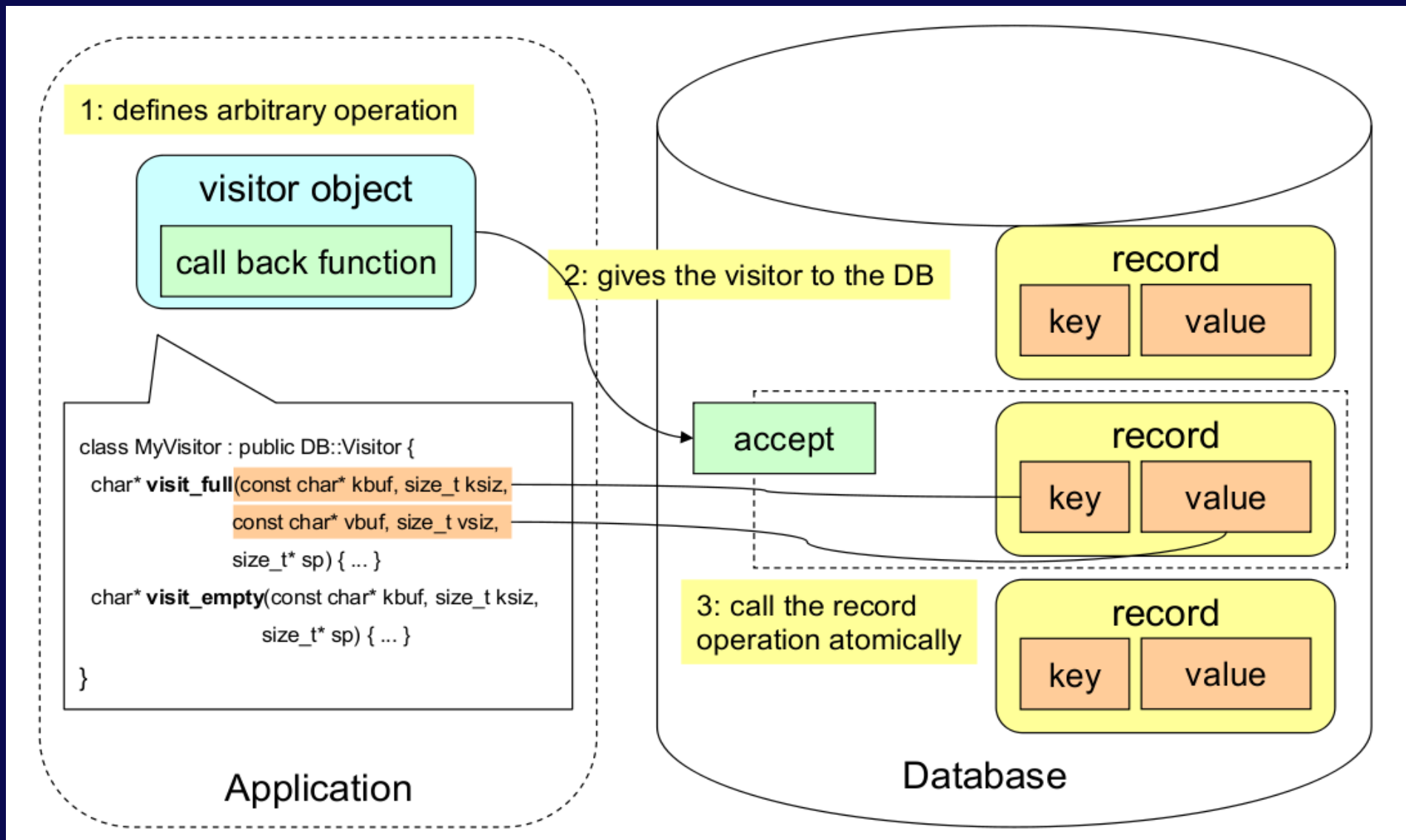
Components



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 the "visitor" pattern
 - the database accepts one visitor for a record at the same time
 - lets it read/write the record arbitrary
 - saves the computed value
- flexible and useful interface
 - provides the "DB::accept" method realizing whatever operations
 - "DB::set", "DB::get", "DB::remove", "DB::increment", etc. 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
 - new database types: GrassDB, ForestDB
- robustness: auto transaction and auto recovery

- **cons**

- time efficiency per thread: due to grained lock
- discarded features: fixed-length database, table database
- dependency on modern C++ implementation: jilted old environments

Example Codes

```
#include <kcpolydb.h>

using namespace std;
using namespace kyotocabinet;

// main routine
int main(int argc, char** argv) {
    // create the database object
    PolyDB db;
    // open the database
    if (!db.open("casket.kch", PolyDB::OWRITER | PolyDB::OCREATE)) {
        cerr << "open error: " << db.error().name() << endl;
    }
    // store records
    if (!db.set("foo", "hop") ||
        !db.set("bar", "step") ||
        !db.set("baz", "jump")) {
        cerr << "set error: " << db.error().name() << endl;
    }
    // retrieve a record
    string* value = db.get("foo");
    if (value) {
        cout << *value << endl;
        delete value;
    } else {
        cerr << "get error: " << db.error().name() << endl;
    }
    // traverse records
    DB::Cursor* cur = db.cursor();
    cur->jump();
    pair<string, string>* rec;
    while ((rec = cur->get_pair(true)) != NULL) {
        cout << rec->first << ":" << rec->second << endl;
        delete rec;
    }
    delete cur;
    // close the database
    if (!db.close()) {
        cerr << "close error: " << db.error().name() << endl;
    }
    return 0;
}
```

```
#include <kcpolydb.h>

using namespace std;
using namespace kyotocabinet;

// main routine
int main(int argc, char** argv) {
    // create the database object
    PolyDB db;
    // open the database
    if (!db.open("casket.kch", PolyDB::OREADER)) {
        cerr << "open error: " << db.error().name() << endl;
    }
    // define the visitor
    class VisitorImpl : public DB::Visitor {
        // call back function for an existing record
        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;
        }
        // call back function for an empty record space
        const char* visit_empty(const char* kbuf, size_t ksiz, size_t *sp) {
            cerr << string(kbuf, ksiz) << " is missing" << endl;
            return NOP;
        }
    } visitor;
    // retrieve a record with visitor
    if (!db.accept("foo", 3, &visitor, false) ||
        !db.accept("dummy", 5, &visitor, false)) {
        cerr << "accept error: " << db.error().name() << endl;
    }
    // traverse records with visitor
    if (!db.iterate(&visitor, false)) {
        cerr << "iterate error: " << db.error().name() << endl;
    }
    // close the database
    if (!db.close()) {
        cerr << "close error: " << db.error().name() << endl;
    }
    return 0;
}
```

Other Kyoto Products?

- now, planning...
- **Kyoto Tycoon?**
 - database server using Kyoto Cabinet
 - supports expiration and multiple databases
- **Kyoto Diaspora?**
 - inverted index using Kyoto Cabinet

maintainability is our paramount concern...

FAL Labs

<http://fallabs.com/>
<mailto:info@fallabs.com>

京都



キャビネット

8 EiB