
Introduction to
Kyoto Products

Kyoto Products

- Kyoto Cabinet
  - a lightweight database library
    - a straightforward implementation of DBM
- Kyoto Tycoon
  - a lightweight database server
    - a persistent cache server based on Kyoto Cabinet


Kyoto Cabinet
- lightweight database library -

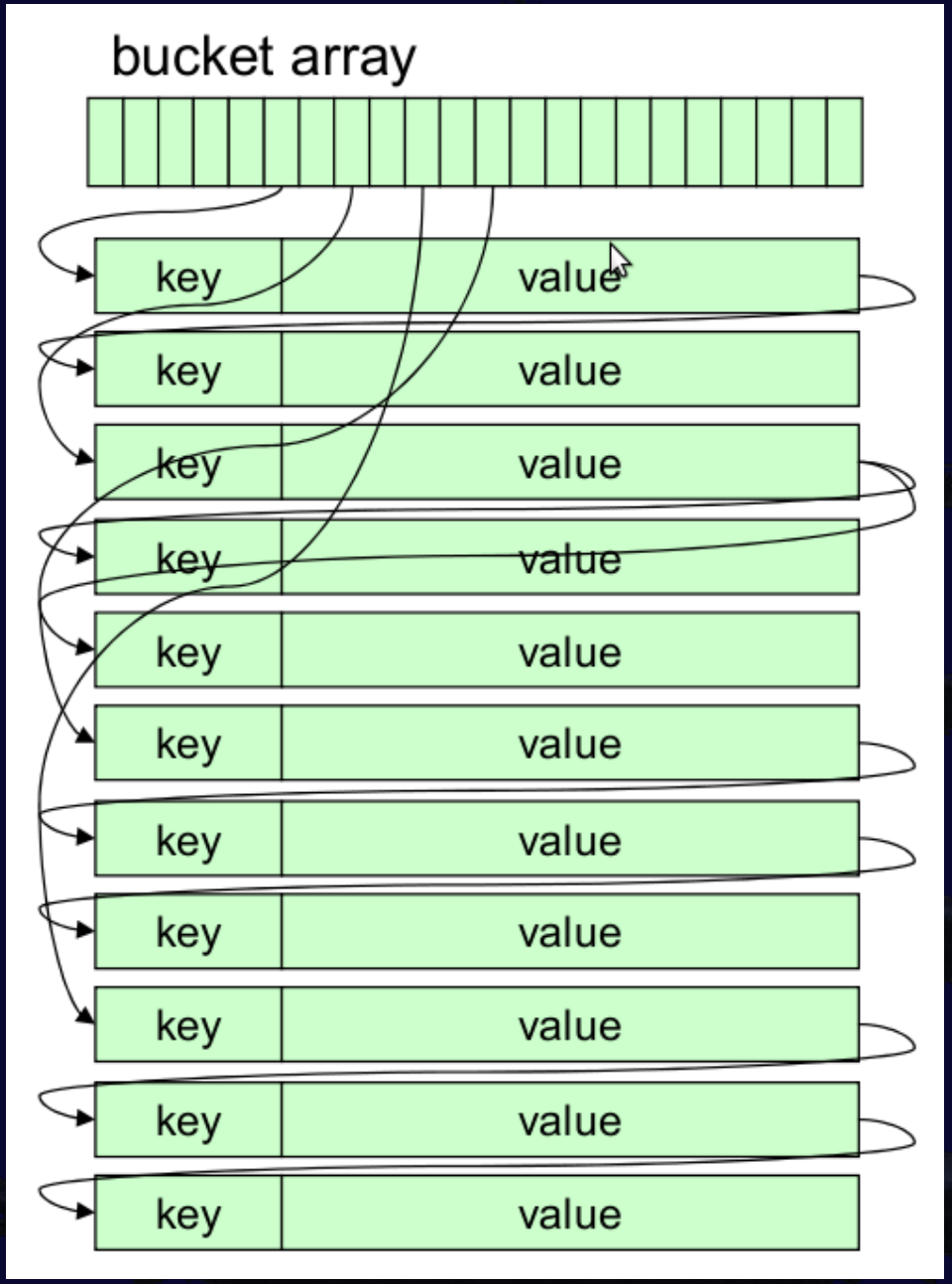
Features

- 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 porable
    - 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
  - utilities: iterator, prefix/regex matching, MapReduce, index database
- other language bindings
  - C, Java, Python, Ruby, Perl, Lua, and so on

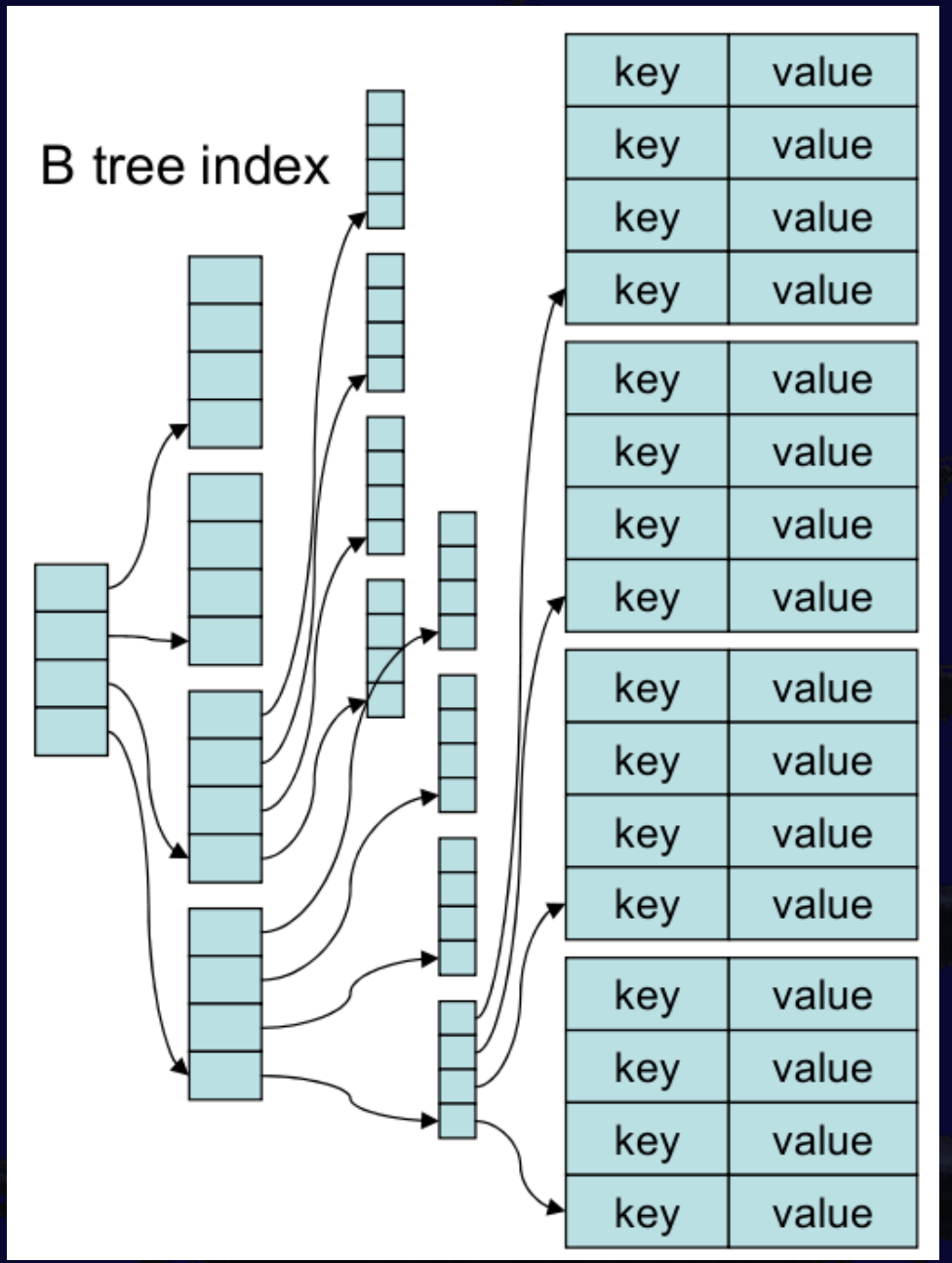
HashDB: File Hash Database

- 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, unable to step back	ordered by application logic jump by range matching, able 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
  - reduces the number of files and alleviates the load
  - the 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>
- StashDB: stash database
  - static hash table with packed serialization
  - much less memory usage than std::map and std::unordered_map
- CacheDB: cache hash database
  - static hash table with doubly-linked list
  - constant memory usage
  - LRU (least recent used) records are removed
- GrassDB: cache tree database
  - B+ tree stands on CacheDB
  - much less memory usage than CacheDB

performance characteristics

class name	persistence	algorithm	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)
StashDB	volatile	hash table	O(1)	undefined	record (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)

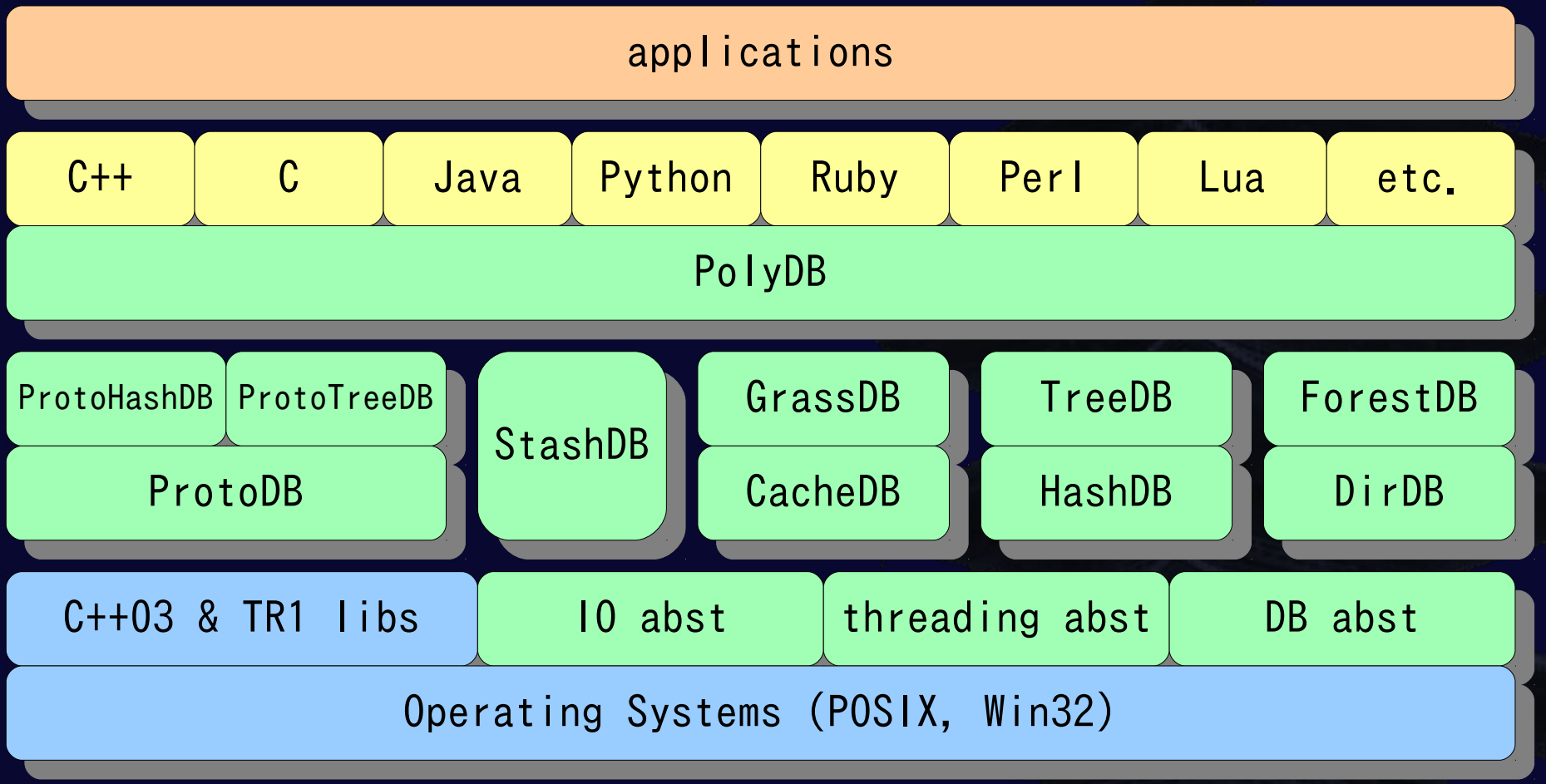
Tuning Parameters

- Assumption
  - stores 100 million records whose average size is 64 bytes
  - pursues time efficiency first, and space efficiency second
  - the machine has 16GB RAM
- HashDB: suited for usual cases
  - opts=TLINEAR, bnum=200M, msiz=12G
- TreeDB: suited for ordered access
  - opts=TLINEAR, bnum=10M, msiz=10G, pccap=2G
- StashDB: suited for fast on-memory associative array
  - bnum=100M
- CacheDB: suited for cache with LRU deletion
  - bnum=100M, capcnt=100M
- GrassDB: suited for memory saving on-memory associative array
  - bnum=10M, psiz=2048, pccap=2G
- DirDB and ForestDB: not suited for small records

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"

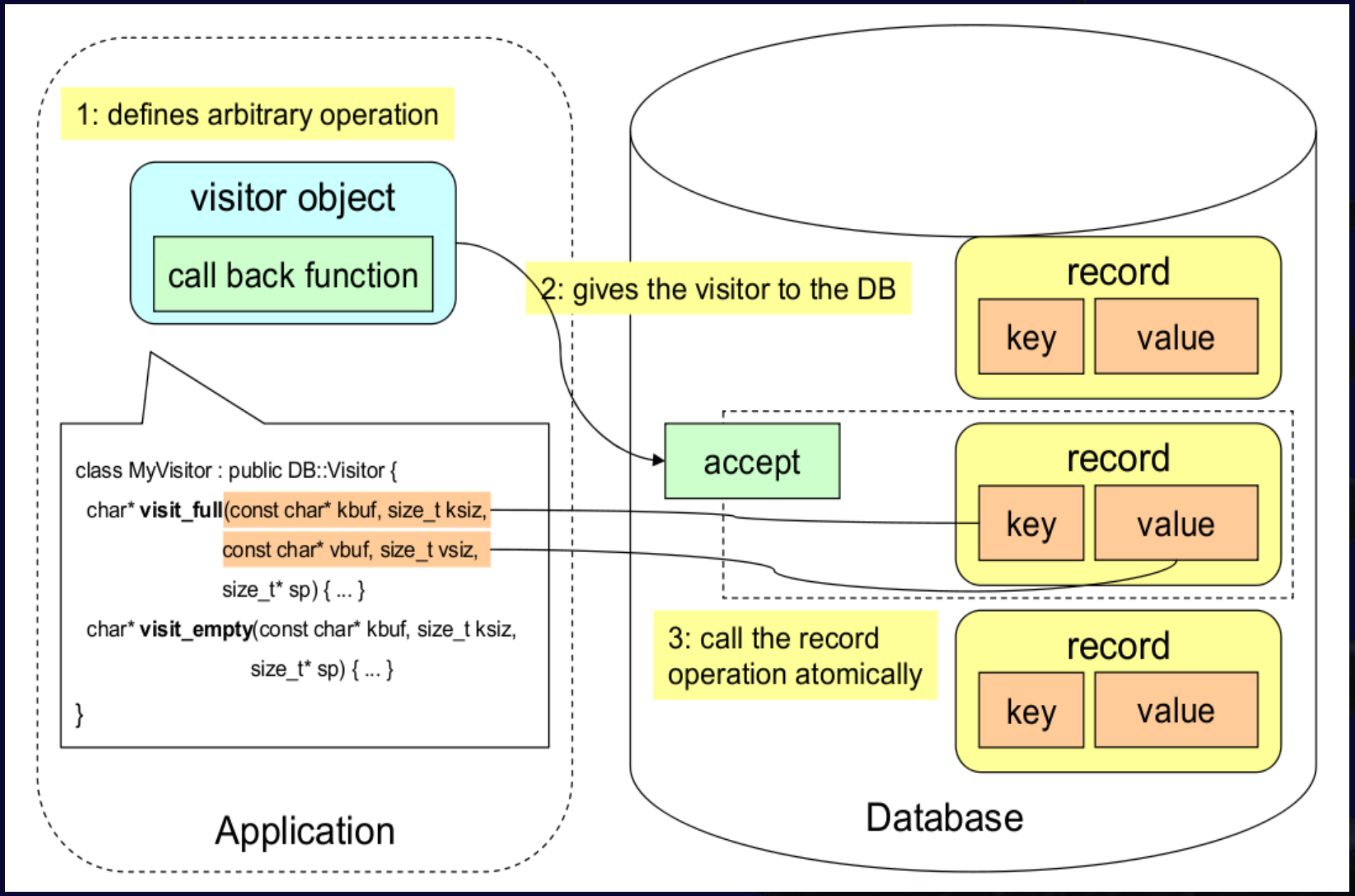
Components


applications
C++
C
Java
Python
Ruby
Perl
Lua
etc.
PolyDB
ProtoHashDB
ProtoTreeDB
ProtoDB
StashDB
GrassDB
CacheDB
TreeDB
HashDB
ForestDB
DirDB
C++03 & TR1 libs
IO abst
threading abst
DB abst
Operating Systems (POSIX, Win32)

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: StashDB, 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

```cpp
#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;
}
```

```cpp
#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;
}
```


Kyoto Tycoon
- lightweight database server -

Features

- persistent cache server based on Kyoto Cabinet
  - persistent associative array = key-value storage
  - client/server model = multi processes can access one database
- auto expiration
  - expiration time can be given to each record
    - records are removed implicitly after the expiration time
  - "GC cursor" eliminates expired regions gradually
- high concurrency and performance
  - resolves "c10k" problem with epoll/kqueue
  - 1M queries / 25 sec = 40,000 QPS or more
- high availability
  - hot backup, update logging, asynchronous replication
  - background snapshot for persistence of on-memory databases

- protocol based on HTTP
  - available from most popular languages
  - provides access library to encapsulate the protocol
  - supports bulk operations also in an efficient binary protocol
  - supports "pluggable server" and includes a memcached module
- various database schema
  - using the polymorphic database API of Kyoto Cabinet
  - almost all operations of Kyoto Cabinet are available
  - supports "pluggable database"
- scripting extension
  - embeds Lua processors to define arbitrary operations
  - supports visitor, cursor, and transaction
- library of network utilities
  - abstraction of socket, event notifier, and thread pool
  - frameworks of TCP server, HTTP server, and RPC server

Networking

- abstraction of system-dependent mechanisms
  - encapsulates threading mechanisms
    - thread, mutex, rwlock, spinlock, condition variable, TLS, and atomic integer
    - task queue with thread pool
  - encapsulates Berkeley socket and Win32 socket
    - client socket, server socket, and name resolver
  - encapsulates event notifiers
    - "epoll" on Linux, "kqueue" on BSD, "select" on other POSIX and Win32
    - substitutes for "libevent" or "libev"
- TSV-RPC
  - lightweight substitute for XML-RPC
  - input data and output data are associative arrays of strings
    - serialized as tab separated values
- RESTful interface
  - GET for db::get, PUT for db::set, DELETE for db::remove

Example Messages

db::set over TSV-RPC

```
POST /rpc/set HTTP/1.1
Content-Length: 22
Content-Type: text/tab-separated-values

key     japan
value   tokyo

HTTP/1.1 200 OK
Content-Length: 0
```

db::get over TSV-RPC

```
POST /rpc/get HTTP/1.1
Content-Length: 10
Content-Type: text/tab-separated-values

key     japan

HTTP/1.1 200 OK
Content-Length: 12
Content-Type: text/tab-separated-values

value   tokyo
```

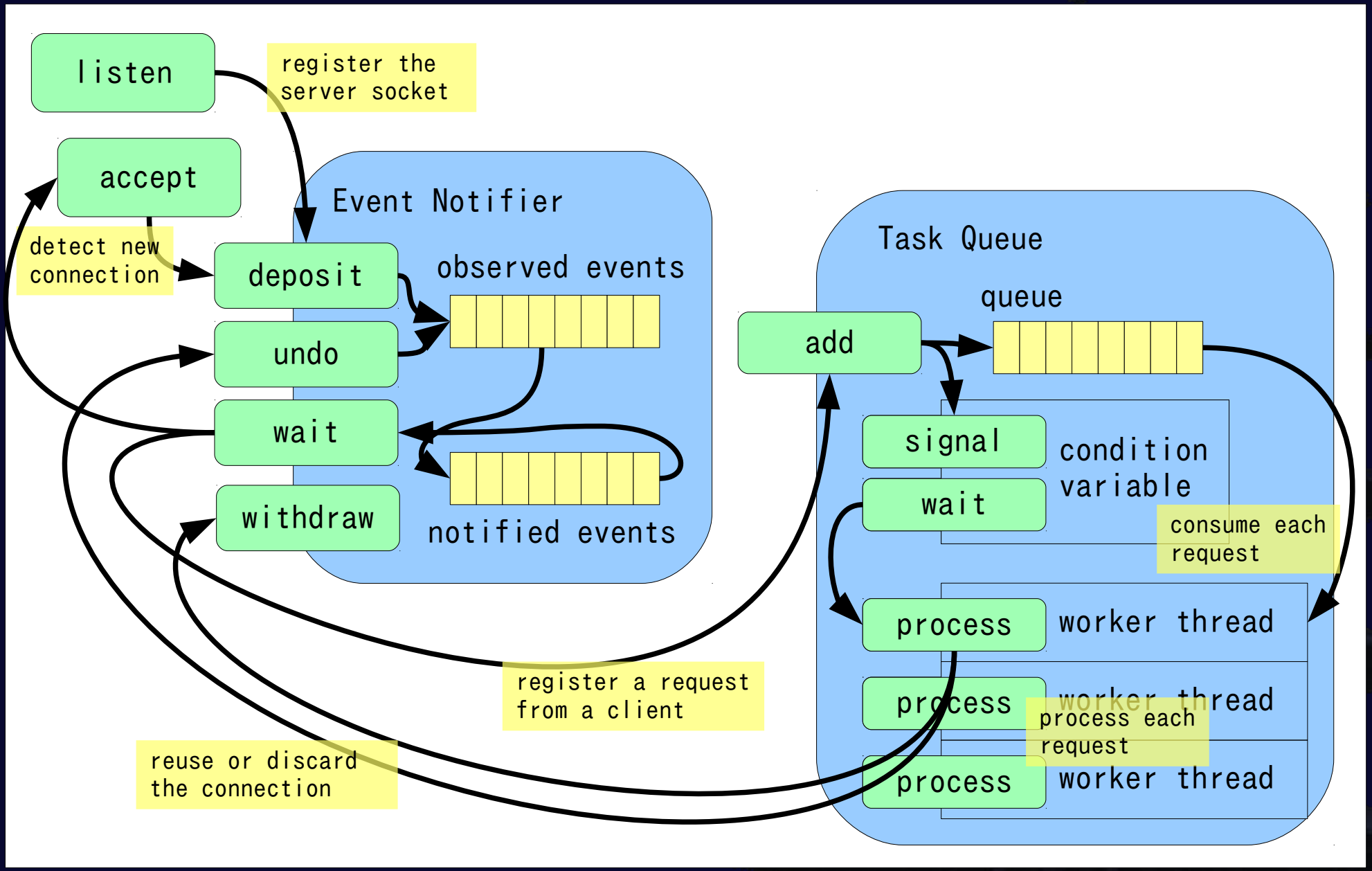
db::set over RESTful

```
PUT /japan HTTP/1.1
Content-Length: 5
Content-Type: application/octet-stream

tokyo

HTTP/1.1 201 Created
Content-Length: 0
```

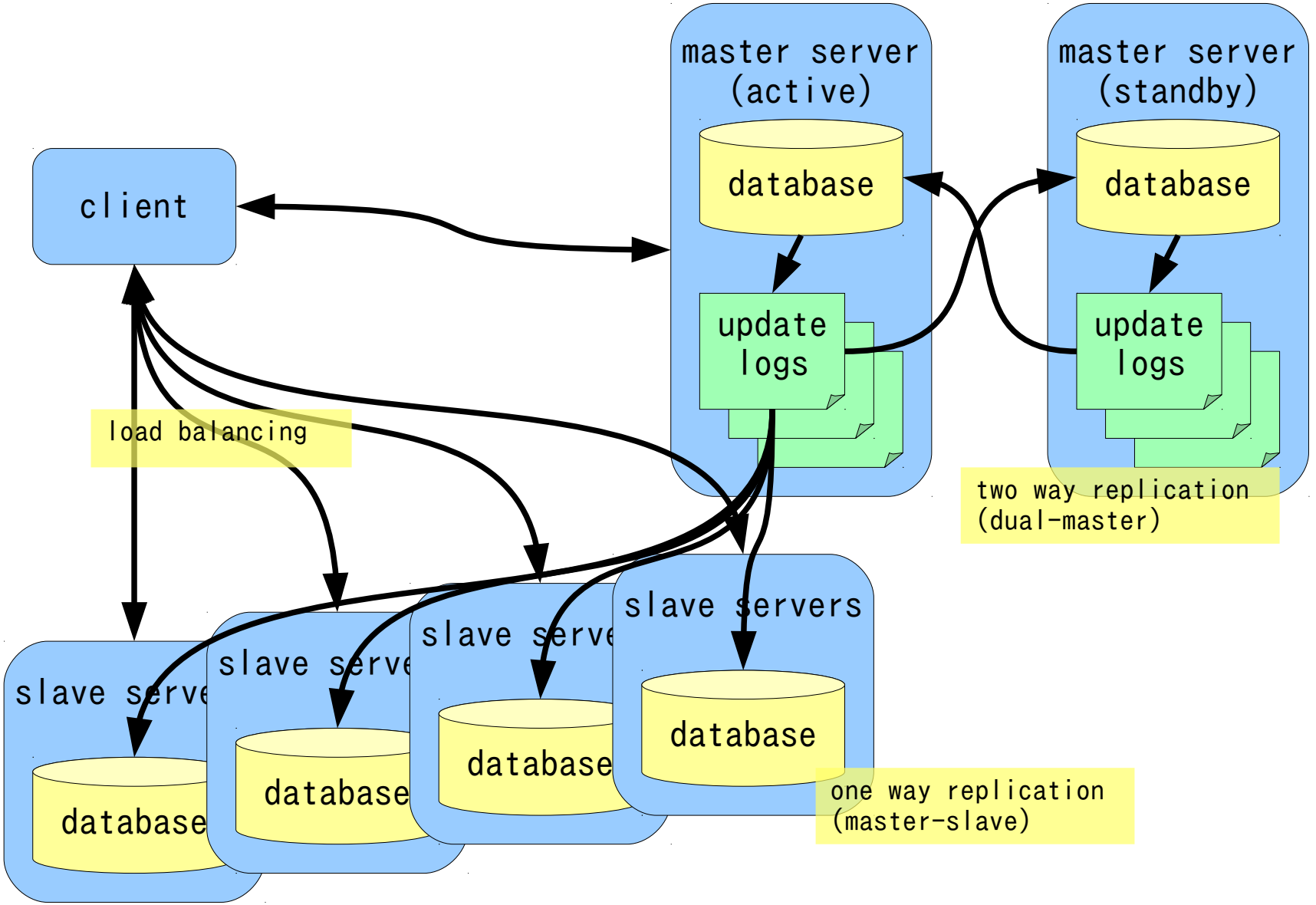
db::get over RESTful

```
GET /japan HTTP/1.1

HTTP/1.1 200 OK
Content-Length: 5
Content-Type: application/octet-stream

tokyo
```

Threaded TCP Server

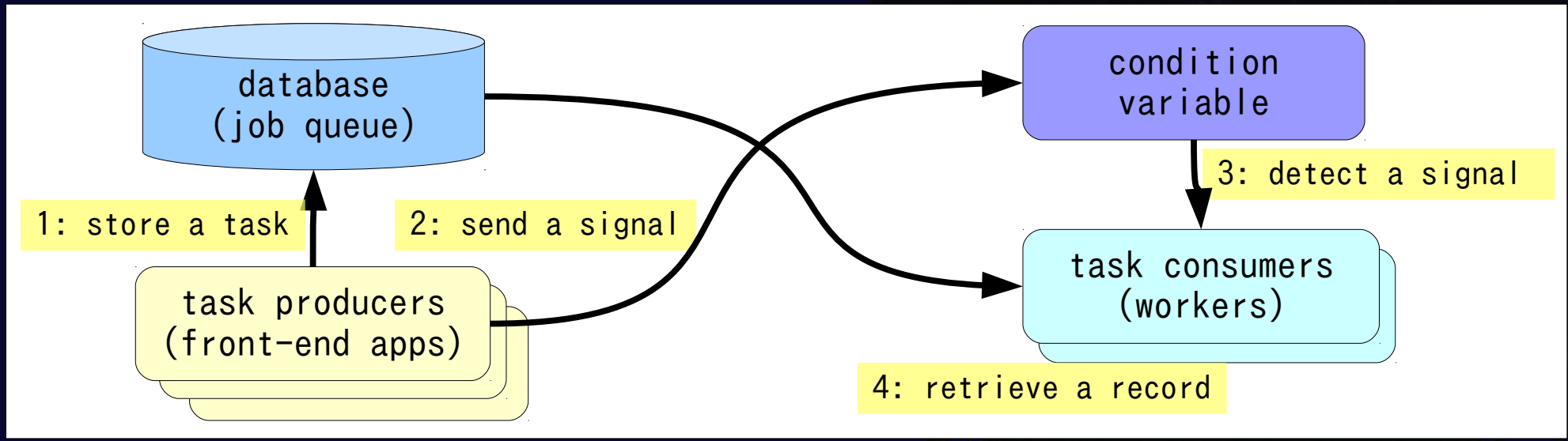


Asynchronous Replication



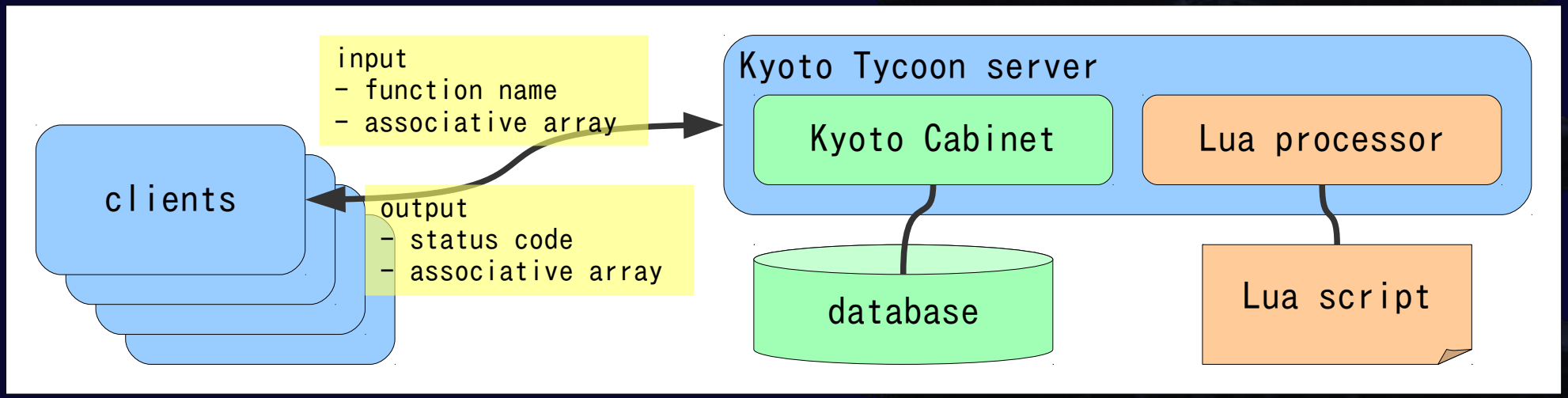
Update Monitoring by Signals

- named condition variables
  - waits for a signal of a named condition before each operation
  - sends a signal to a named condition after each operation
- task queue
  - uses B+ tree and stores records whose keys are time stamps
    - every records are sorted in the ascending order of time stamps
  - writers store tasks and send signals, readers wait for signals and do



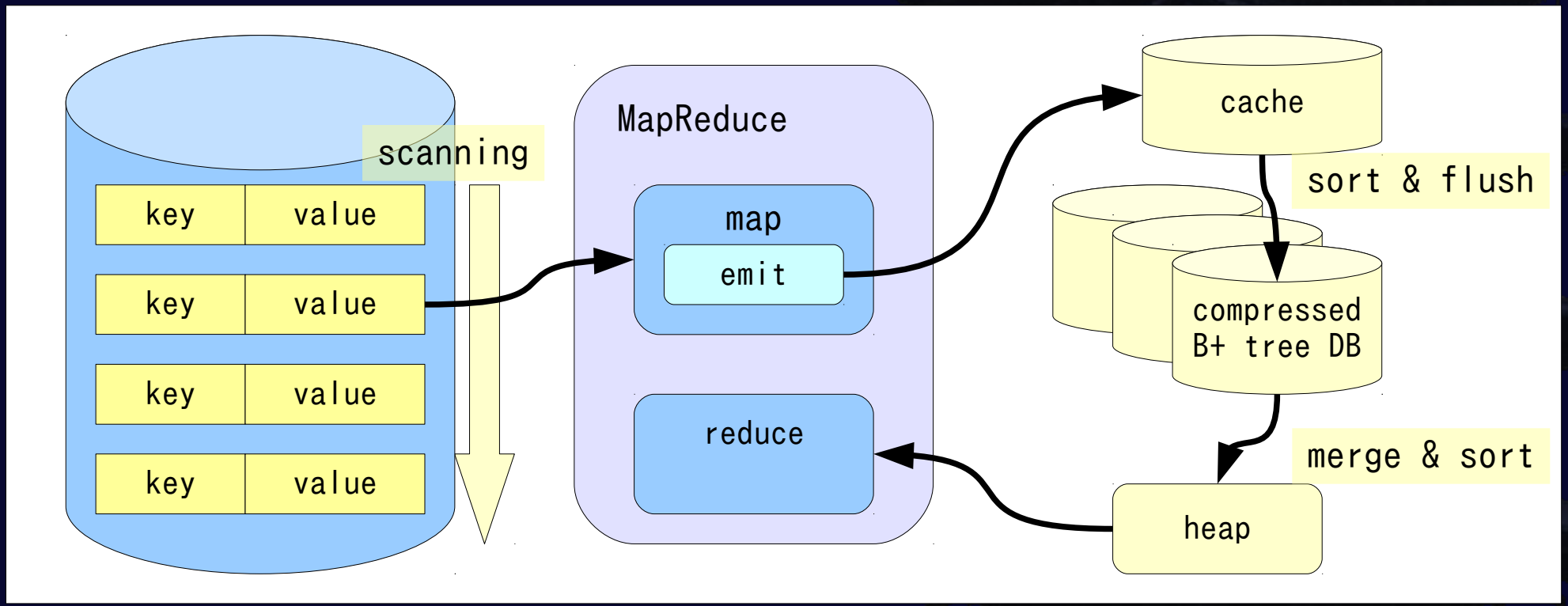
Scripting Extension with Lua

- prominent time efficiency and space efficiency
  - suitable for embedding use
- extreme concurrency
  - each native thread has respective Lua instance
- poor functionality, compensated for by C functions
  - full set of the Lua binding of Kyoto Cabinet
  - pack, unpack, split, codec, and bit operation utilities

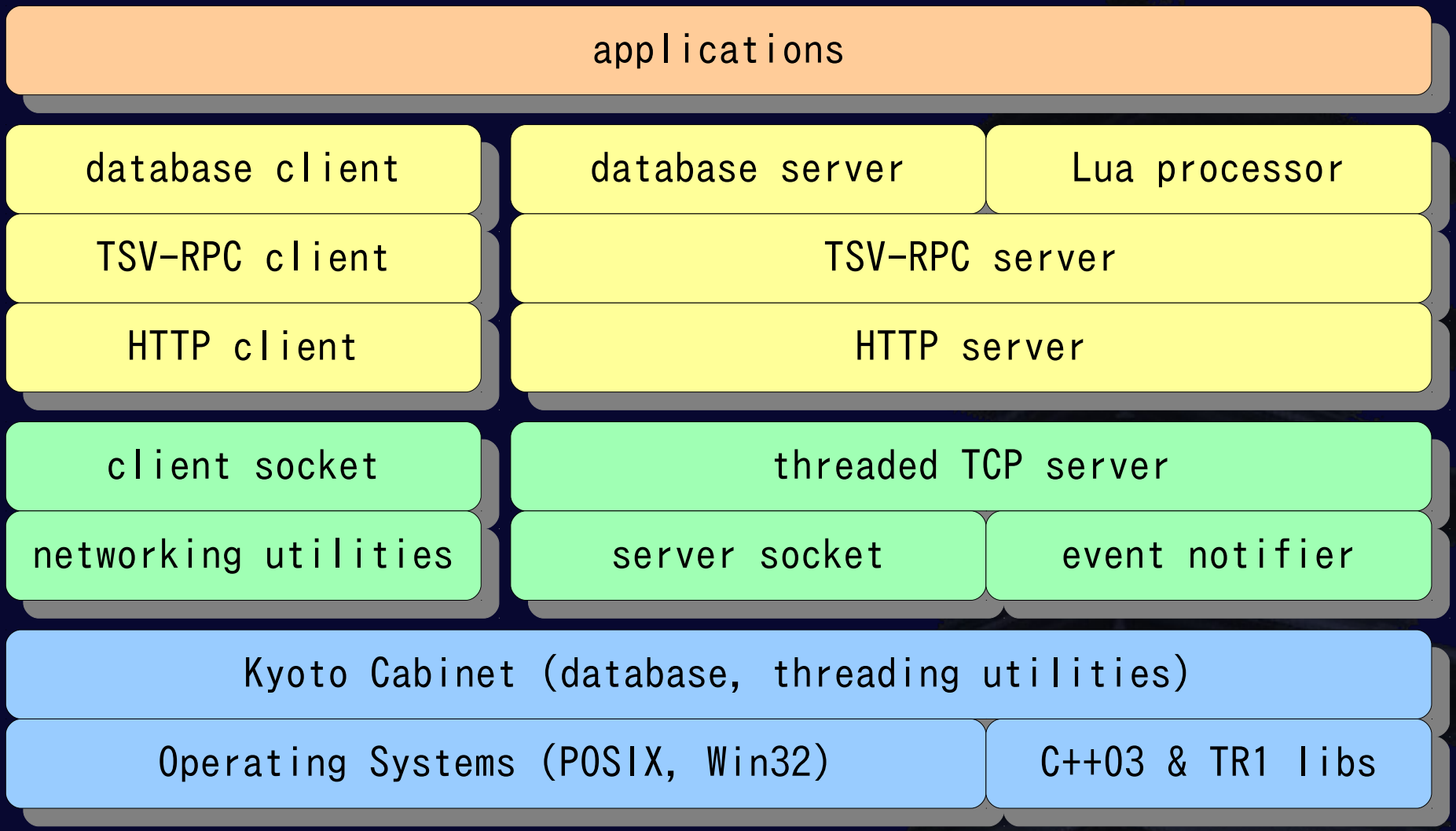


Local MapReduce in Lua

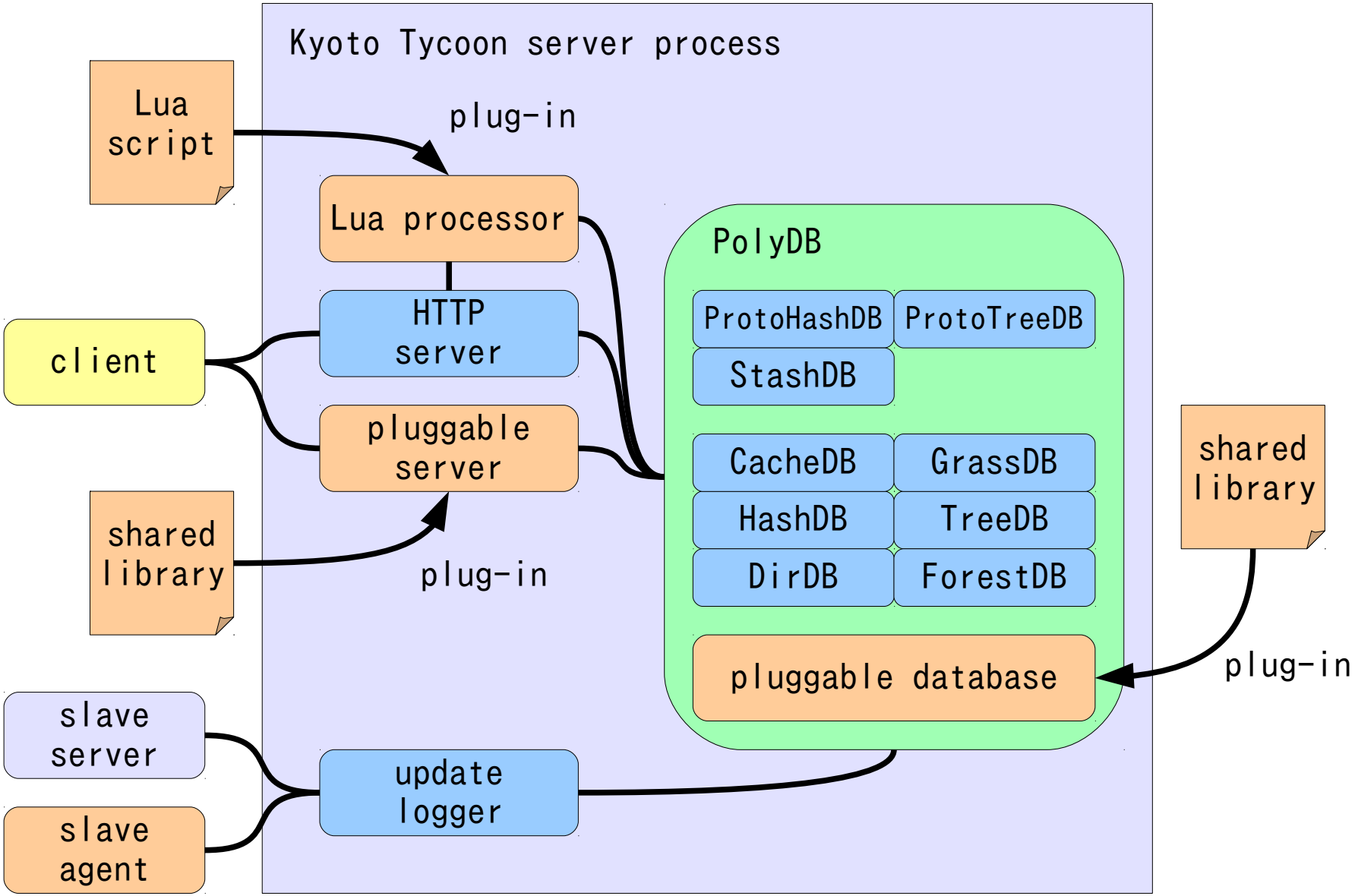
- for on-demand aggregate calculation
  - the "map" and the "reduce" functions defined by users in Lua
- local = not distributed
  - fast cache and sort implementation in C++, no overhead for networking
  - space saving merge sort by compressed B+ trees



Components


applications
database client
database server
Lua processor
TSV-RPC client
TSV-RPC server
HTTP client
HTTP server
client socket
threaded TCP server
networking utilities
server socket
event notifier
Kyoto Cabinet (database, threading utilities)
Operating Systems (POSIX, Win32)
C++03 & TR1 libs

Pluggable Modules



Comparison with Tokyo Tyrant

- pros
  - using Kyoto Cabinet
    - inherits all pros of Kyoto Cabinet against Tokyo Cabinet
  - auto expiration
    - easy to use as a cache server
  - multiple databases
    - one server can manage multiple database instances
  - thorough layering
    - database server on TSV-RPC server on HTTP server on TCP server
    - easy to implement client libraries
  - portability: non-POSIX platform support
    - supports Win32 (work-in-progress)
- cons
  - discarded features: fixed-length database, table database

Example Codes

```cpp
#include <ktremotedb.h>

using namespace std;
using namespace kyototycoon;

// main routine
int main(int argc, char** argv) {
  // create the database object
  RemoteDB db;
  // open the database
  if (!db.open()) {
    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
  RemoteDB::Cursor* cur = db.cursor();
  cur->jump();
  pair<string, string>* rec;
  while ((rec = cur->get_pair(NULL, 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;
}
```

```lua
kt = __kyototycoon__
db = kt.db

-- retrieve the value of a record
function get(inmap, outmap)
   local key = inmap.key
   if not key then
      return kt.RVEINVALID
   end
   local value, xt = db:get(key)
   if value then
      outmap.value = value
      outmap.xt = xt
   else
      local err = db:error()
      if err:code() == kt.Error.NOREC then
         return kt.RVELOGIC
      end
      return kt.RVEINTERNAL
   end
   return kt.RVSUCCESS
end

-- list all records
function list(inmap, outmap)
   local cur = db:cursor()
   cur:jump()
   while true do
      local key, value, xt = cur:get(true)
      if not key then break end
      outmap.key = value
   end
   return kt.RVSUCCESS
end
```

Other Kyoto Products?

- now, planning...

- Kyoto Diaspora?
  - inverted index using Kyoto Cabinet
- Kyoto Prominence?
  - content management system 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