



本科生学位论文

题目： 针对高速SSD的KV接口标准

实现与测试

姓 名： 王润辉

学 号： 1200018504

院 系： 信息科学技术学院

专 业： 计算机科学与技术

研究方向： 计算机体系与结构

导 师： 孙广宇

2016年5月

摘要

随着存储技术的迅速发展，当今以KV存储系统为代表的NoSQL数据库出现了百花齐放的状况，在不同的应用领域有着许多针对硬盘或者SSD的开源KV存储项目，然而它们都有着各自定义的KV API接口，上层应用为了实现相同的语义需要针对不同的KV存储分别搭建不同的适配器。为了减少适配器的重复开发，可以提出一个通用的KV框架——KVF，其定义的API的语义概括了通用的KV模型，使上层应用可以使用一个统一的API接口来访问下层的KV存储。因此，KVF简化了基于KV模型的应用的开发过程。

为了验证KV Framework(KVF)底层的实用性，我将Fusion-io盘与KVF框架对接，并开发Benchmark框架下的测试工具并进行端到端功能/性能测试。

YCSB是雅虎云服务测试的简称，是用来测试cloud serving/NoSQL/Key-Value Store的测试工具。由于云服务的流行，传统数据库不能满足可利用性、可扩展性等要求，因此功能简化、一致性简化的NoSQL数据库逐渐流行。然而NoSQL数据库种类繁多，针对不同的数据库各有权衡（读写性能、延迟和持久性、同步和异步等等），用户和开发人员需要针对不同的应用场景选择合适的数据库。YCSB的目标是提供一个公平测试的平台，为这些数据库提供一个统一的测试方案，从而更公平地在不同的方面衡量不同的数据库性能，从而提供有价值的参考。

通过将YCSB与KVF进行连接，我们可以验证KVF的性能，为将来KVF与各种NoSQL数据库的整合打好基础。

关键词： KV存储,统一接口,YCSB

Implementation and benchmark of standard KV interface aimed at high speed SSD

Runhui Wang (Computer science and technology)

Directed by Prof. Guangyu Sun

ABSTRACT

With the rapid development of storage technology, there exist a variety of NoSQL databases which is represented by KV storage system. There are many open—source KV storage projects for specific disks in different fields. However, their KV APIs are different from each other so upper lever applications have to implement different adapters for different KV storages. To avoid developing different adapters in the same application, we can design a general KV framework, whose API concludes general KV model. Therefore, upper level applications can call unified interface to access lower lever KV storage. In this case, KVF simplifies the process of developing applications which are based on KV model.

To testify the utility of KV Framework, I will connect Fusion-disk with KVF and develop a benchmark tool to perform peer-to-peer tests of functionality and performance.

KEYWORDS: KV Store,Framework, YCSB

目录

序言	1
第一章 KV Framework设计	2
1.1 概览	2
1.2 键值存储（Key Value Store）介绍	2
1.3 Key Value Framework (KVF) 数据模型	3
1.4 Key Value Framework (KVF) 体系结构	3
第二章 KV Framework底层管理接口设计	5
2.1 概览	5
2.2 基本数据结构	5
2.2.1 基本数据结构	5
2.2.2 字符串结构定义	5
2.3 KVF注册和注销接口	6
2.4 KV-LIB 接口	6
2.4.1 kvf_init()	6
2.4.2 kvf_shutdown ()	6
2.4.3 kvf_set_prop()	7
2.4.4 kvf_get_prop()	7
2.4.5 kvf_alloc_buf()	7
2.4.6 kvf_free_buf()	7
2.4.7 kvf_get_errstr()	8
2.4.8 kvf_get_stats()	8
2.5 POOL接口	8
2.5.1 pool_create()	9
2.5.2 pool_destroy()	9
2.5.3 pool_open()	9
2.5.4 pool_close()	9
2.5.5 pool_set_prop()	10
2.5.6 pool_get_prop()	10
2.5.7 pool_get_stats()	10

2.6 对象键值接口	10
2.6.1 put	11
2.6.2 get	11
2.6.3 del	12
2.6.4 iter_open	12
2.6.5 iter_next	12
2.6.6 iter_close	13
2.7 上层接口	13
2.7.1 KVF接口	13
2.7.2 Pool接口	14
2.7.3 Pool接口	14
2.8 概览	16
2.9 YCSB应用模型	16
2.9.1 YCSB客户端	16
2.9.2 Workload	17
2.9.3 JNI	18
2.9.4 实现方式	20
2.9.5 实现流程	24
2.9.6 测试计划	24
结论	28
致谢	29
北京大学学位论文原创性声明和使用授权说明	30

序言

在做这个项目的开发过程中遇到了一系列的问题，在探索这些问题的过程中，自己也从对很多知识的一知半解到熟练使用，比如git版本控制、服务器上的各种权限问题、开源项目的编译配置、makefile的编写以及c和java语言的编译链接问题等等。在学以致用的过程中，总是充满了未知的挑战，很多时候会在某个问题卡住很久，但是当解决了一个个棘手的问题之后，内心的喜悦会证明之前的付出都是值得的。做毕业设计是一个不断探索未知，不断挑战自己的过程，在这个过程中不仅我的研究能力得到了锻炼，而且处理问题的能力也得到了很大提升。

本文将由两个部分构成：

- **KVF**: 该部分描述了KVF的基本原理和设计思路主要包括KVF的数据模型、系统架构、应用场景此外也介绍了一些基本的API
- **Benchmark的实现**: 该部分介绍如何实现Benchmark 主要内容包括如何依托开源项目YCSB（Yahoo! Cloud Serving Benchmark）实现Benchmark以及一些基本的测试结果与分析

第一章 KV Framework设计

1.1 概览

这部分将介绍键值对框架（Key Value Framework）的概念、概述以及接口的定义。KVF定义了统一的数据模型、函数和参数。通过这样一个框架，不同的Key value storage可以注册和注销注册，像虚拟文件系统管理ext3、jfs一样。此外该框架提供了统一的API以便上层应用调用，从而避免了同一个应用为了在不同厂商的的KV store实现功能而开发多个适配器的情况。

1.2 键值存储（Key Value Store）介绍

传统的存储系统是基于SCSI架构的，客户端使用逻辑块地址（Logical Block Address, LBA）进行读写数据。在块的基础上我们可以提供更多的服务，比如文件系统、数据库等等。作为替代品，键值存储（也称为目标存储系统）提供了键值对操作。上层应用通过指定键名来存储数据值，键的名字可以是一个字符串变量，数据值可以是一个数据缓冲变量，这样上层应用可以很容易地存储数据，不需要基于线性块空间的复杂的布局设计，只需要聚焦于键名策略的设计和如何存储键名（比如使用分布式哈希表存储）。基于键值存储我们可以很容易地构造文件、swift、hdfs、no-sql甚至是块服务，如图1所示。



图1

实际上512或者4096字节大小的块设备是一种简单的键值存储，它的键名是LBA（Logical Block Address），数据值是512或者4096个字节大小的数据，因为这种简单性，上层应用需要复杂的布局设计（比如RamCloud），而数据大小可变的键值存储更加灵活，所以上层应用可以很容易在上面存储数据，这是一种性能和可用性之间的权衡。根据冗余的多少，键值存储可以分为三类：

- **独立的KVS:**

独立的KVS是指如下的应用场景：应用客户端直接从一个独立的设备读写数据，比如从一个单独的硬盘、闪存卡等等。相关的产品包括希捷的Kinetic以及闪迪

的Fusionio等等

- **分布式KVS:**

分布式KVS是指如下应用场景：数据储存在多个服务器节点之间存在冗余，通过某种方式相互通信从而将这些服务器节点的存储资源整合到一起，并应用客户端提供读写接口。相关产品包括Ceph、mongoDB等等。

- **多数据中心的KVS:**

多数据中心的KVS应用于如下场景：数据在多重数据中心都有冗余，有复制备份和纠删码的特性，相关的产品包括Amazon S3、SWIFT等等。

这三种KVS是有关联的，但是可以相互独立，例如多数据中心的KVS可以是基于分布式KVS的，分布式KVS页可以是基于独立KVS的。

1.3 Key Value Framework (KVF) 数据模型

通过Key Value Framework (KVF) 不同的厂商可以通过注册自己的Key Value库 (KV-LIB) 来管理它的Key Value Storage (KVS)。这些KV-LIB应该遵循KVF数据模型，KVS提供pool，pool提供对象。pool支持嵌套设计，这意味着一个pool可以在另一个pool的上面，如图2所示。

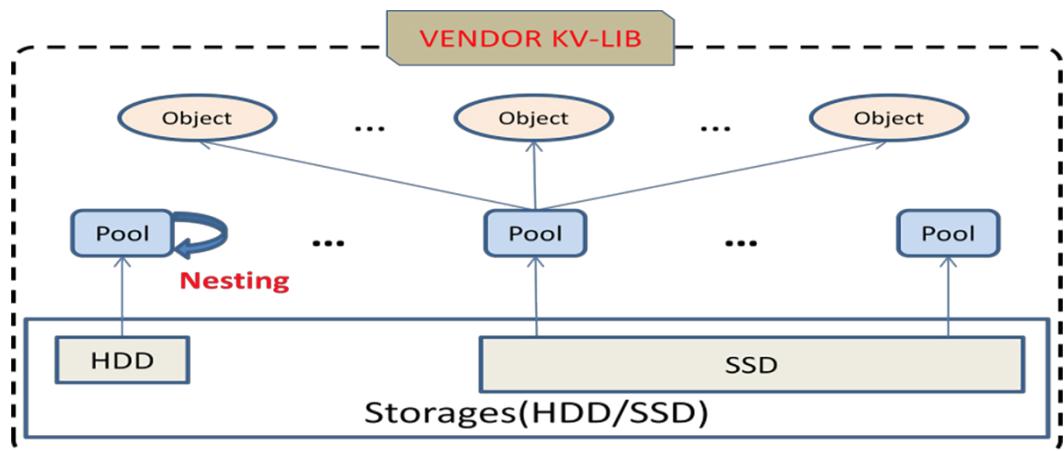


图2

1.4 Key Value Framework (KVF) 体系结构

KVF提供了两层接口：底层管理接口（Lower Layer Manage API）和上层访问接口（Upper Layer Access API）。如图3所示

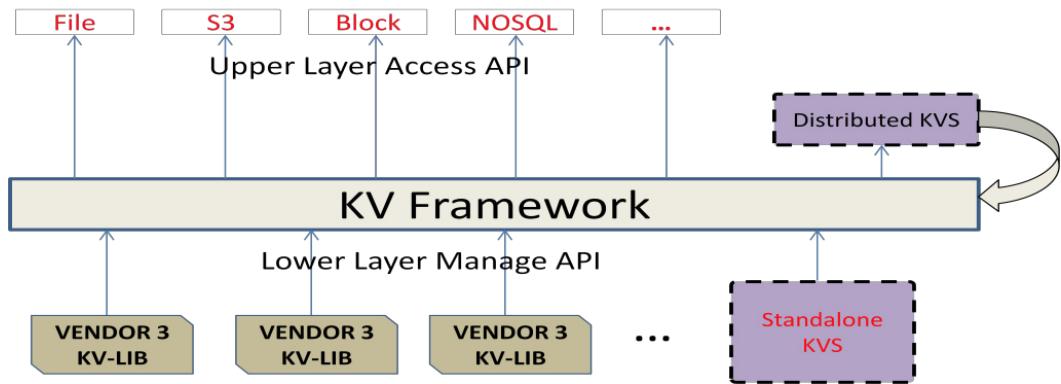


图3

底层管理提供注册和注销函数，共有三种接口：对象接口，Pool接口和KV-LIB接口。上层访问提供统一的键值接口，从而使上层应用程序变得简单并有更强的兼容性。此外一些应用可以通过利用基于其他KVS的上层访问接口提供键值服务，然后再重新注册到KVF中，这样就使得嵌套设计变得容易。

第二章 KV Framework底层管理接口设计

2.1 概览

这章将描述底层管理接口设计，包括KVF接口，Pool接口，对象接口。KVF的相关函数名称都以“KVF_”开始。

2.2 基本数据结构

2.2.1 基本数据结构

自定义数据类型

c语言数据类型	新类型名
signed	char s8
unsigned char u8	char u8
signed short s16	short s16
unsigned short u16	short u16
signed int s32	s32
unsigned int u32	u32
signed long long s64	s64
unsigned long long u64	u64
unsigned int size_t	size_t

2.2.2 字符串结构定义

自定义字符串类型

Type Structure	Definition	Description
String	<pre>typedef struct string { u32 len; s8* data; } string_t;</pre>	<p>len: the length of the string data: a pointer which points to</p>

2.3 KVF注册和注销接口

```
s32 kvf_register(kvf_type_t * kvf)
s32 kvf_unregister(kvf_type_t * kvf)
```

这两个接口使得厂商可以注册和注销自己的键值存储到KVF中，相关信息应该存储在kvf_type_t数据结构中，是开发者必须实现的函数。

kvf_type_t结构定义如下

```
typedef struct kvf_type
{
    s32 magic;      //magic of the kvf
    s32 flags;      //flags
    char* name; //the name should be unique

    struct kvf_operations* kvf_ops; //the kvf operations
    struct pool_operations* pool_ops; //the pool operations of the kvf

    struct list kvf_list; //all the kvf instance are linked together
    struct list pool_list; //all of pool instance of a kvf are linked together
    void* kvf_private;
} kvf_type_t;
```

2.4 KV-LIB 接口

KV-LIB是厂商指定的键值库，通过这个键值库，厂商可以管理键值存储，包括初始化、关闭、设置属性等待。

2.4.1 kvf_init()

该函数用于初始化KVF

```
s32 kvf_init(const char * config_file)
```

参数说明：

config_file:是指定的配置文件的目录

2.4.2 kvf_shutdown ()

该函数用于关闭KVF

```
s32 kvf_shutdown()
```

参数说明：

- 此函数没有参数

2.4.3 kvf_set_prop()

该函数用于设置KVF的相关属性值

```
s32 kvf_set_prop(const char* name, const char* value)
```

参数说明:

- **name:** 是指相关属性的名称
- **value:** 是相关属性的值

2.4.4 kvf_get_prop()

该函数用于获得KVF的相关属性值

```
s32 kvf_get_prop(const char* name, const char* value)
```

参数说明:

- **name:** 是指相关属性的名称
- **value:** 是相关属性的值

2.4.5 kvf_alloc_buf()

该函数用于在内存中分配一段数据缓冲区

```
void* kvf_alloc_buf (size_t size, s32 flag)
```

参数说明:

- **size:** 是需要的缓冲区空间大小
- **flag:** 是相关属性的值

2.4.6 kvf_free_buf()

该函数用于在内存中释放数据缓冲区

```
void* kvf_free_buf (void** buf)
```

参数说明:

- **buf:** 需要释放内存空间的指针

2.4.7 kvf_get_errstr()

该函数用于根据错误代码得到描述错误问题的字符串

```
const char* kvf_get_errstr (s32 err_code)
```

参数说明:

- **err_code:** 错误代码

2.4.8 kvf_get_stats()

该函数用语获得KVF的统计数据

```
s32 kvf_get_stats (kvf_stats_t* kvfstats)
```

参数说明:

- **kvfstats:** 传入该kvf_stats_t结构的指针, 函数体将数据写入该指针所指向的内存地址

2.5 POOL接口

在KVF-LIB注册并初始化之后, pool才能被管理。pool_t的结构体定义如下:

```
typedef struct pool {
    /* unique pool name */
    char* pool_name;
    /* kvf type */
    kvf_type_t* kvf;
    /* the location info */
    pool_location_t* pool_location;
    /* space info */
    u64 pool_physical_capacity, pool_physical_used, pool_physical_free;
    u64 pool_logical_capacity, pool_logical_used, pool_logical_free;
    /* redundancy policy */
    pool_redundancy_t* pool_availability;
    /* pool's sla */
    u64 pool_latency, pool_throughput;
    u64 pool_obj_cksum_type, pool_obj_cksum_lengh;
    u64 pool_obj_compress_type;
    /* key-value operations */
    struct kv_operations* kv_ops;
    /* pool list link */
    struct list link;
    void* pool_private;
} pool_t;
```

2.5.1 pool_create()

用于创建新的pool

```
s32 pool_create (const char* name, const char* config_path,  
                  pool_t * pool)
```

参数说明:

- **name:** 指定pool的名称，每个KVF里的pool的name须唯一
- **config_path:** 指定相关配置文件的路径
- **pool:** 传入pool_t结构指针，函数体将数据写入该指针所指向的内存地址

2.5.2 pool_destroy()

销毁一个pool

```
s32 pool_destroy (pool_t * pool)
```

参数说明:

- **pool:** 传入pool_t结构指针，函数体将数据写入该指针所指向的内存地址

2.5.3 pool_open()

打开一个pool

```
s32 pool_open (pool_t * pool)
```

参数说明:

- **pool:** 传入pool_t结构指针，函数体将数据写入该指针所指向的内存地址

2.5.4 pool_close()

关闭一个pool

```
s32 pool_close (pool_t * pool)
```

参数说明:

- **pool:** 传入pool_t结构指针，函数体将数据写入该指针所指向的内存地址

2.5.5 pool_set_prop()

设置pool的相关属性

```
s32 pool_set_prop (const pool_t * pool,const char* name,  
                    const char* value )
```

参数说明:

- **pool**: 传入pool_t结构指针, 函数体将数据写入该指针所指向的内存地址, 该参数是常量
- **name**: 需要设置的属性名称
- **value**: 需要设置的属性值

2.5.6 pool_get_prop()

获取指定pool属性的值

```
s32 pool_get_prop (const pool_t * pool,const char* name,  
                    const char* value )
```

参数说明:

- **pool**: 传入pool_t结构指针, 函数体将数据写入该指针所指向的内存地址, 该属性是常量
- **name**: 需要设置的属性名称
- **value**: 需要设置的属性值

2.5.7 pool_get_stats()

获取pool的统计数据

```
s32 pool_set_stats (pool_stats_t * stats)
```

参数说明:

- **stats**: 传入pool_stats_t结构指针, 函数体将数据写入该指针所指向的内存地址

2.6 对象键值接口

在pool成功创建之后, 我们可以操作属于于该pool的对象

2.6.1 put

将数据写入磁盘

```
s32 put(pool_t* pool, const string_t* key, const string_t* value,
       const kv_props_t* props, const put_options_t* putopts)
```

参数说明:

- **pool**: 传入pool_t结构指针, 函数体将数据写入该指针所指向的内存地址, 该参数是常量
- **key**: 需要写入的键
- **value**: 需要写入的属性值
- **props**: 设置键值的属性
- **putopts**: 设置写入模式

相关结构定义:

```
typedef struct kv_props {
    void* kv_private;
} kv_props_t;

typedef struct put_options {
    s8 o_zero_copy;
    s8 o_write_through;
    void* o_private;
} put_options_t;
```

2.6.2 get

从磁盘中根据键获取值

```
s32 get(pool_t* pool, const string_t* key, string_t* value,
        const kv_props_t* props, const get_options_t* getopt)
```

参数说明:

- **pool**: 传入pool_t结构指针, 函数体将数据写入该指针所指向的内存地址, 该参数是常量
- **key**: 需要写入的键
- **value**: 该参数是string_t结构指针, 会将读取的值放入这个指针所指向的地址
- **props**: 设置键值的属性
- **putopts**: 设置写入模式

2.6.3 del

从磁盘中根据键删除值

```
s32 del(pool_t* pool, const string_t* key, const kv_props_t* props,  
       const del_options_t* delopts)
```

参数说明:

- **pool**: 传入pool_t结构指针, 函数体将数据写入该指针所指向的内存地址, 该参数是常量
- **key**: 需要设置的属性名称
- **value**: 需要设置的属性值
- **props**: 设置键值的属性
- **delopts**: 设置写入模式

2.6.4 iter_open

根据正则表达式查询

```
s32 iter_open(const pool_t* pool, const string_t* key_regex,  
              s32 limit, s32 timeout, kv_iter_t* it)
```

参数说明:

- **pool**: 传入pool_t结构指针, 函数体将数据写入该指针所指向的内存地址, 该参数是常量
- **key_regex**: 需要查询的键的正则表达式
- **limit**: 查询次数的上限
- **timeout**: timeout
- **it**:

2.6.5 iter_next

根据正则表达式查询

```
s32 iter_next(pool_t* pool, kv_iter_t* it, kv_array_t* kvarray)
```

参数说明:

- **pool**: 传入pool_t结构指针, 函数体将数据写入该指针所指向的内存地址, 该参数是常量

- **it**: 需要查询的键的正则表达式
- **kvarray**: 查询次数的上限

2.6.6 iter_close

根据正则表达式查询

```
s32 iter_close(pool_t* pool, kv_iter_t* it)
```

参数说明:

- **pool**: 传入pool_t结构指针, 函数体将数据写入该指针所指向的内存地址, 该参数是常量
- **it**: 需要查询的键的正则表达式

2.7 上层接口

当上层应用开发时调用如下的接口即可避免重发的适配工作, KVF将会封装常用的数据库, 这样上层应用就不再考虑不同数据库之间接口的差异, 大大缩短开发时间。

根据不同的抽象层次, 我们将接口分为三种: KV-LIB、Pool、对象。由于大部分功能和底层接口类似, 在这里我将不再详细介绍每一个函数参数的含义。

2.7.1 KVF接口

```
s32 kvf_init(const char * config_file)
```

```
kvf_type_t* get_kvf(const char* name)
```

```
s32 init_kvf(const char * config_file)
```

```
s32 shutdown_kvf()
```

```
s32 set_kvf_prop(const char* name, const char* value)
```

```
s32 get_kvf_prop(const char* name, char** value)
```

```
void* alloc_kvf_buf (size_t size, s32 flag)
```

```
void free_kvf_buf (void** buf)

s32 get_kvf_stats (kvf_stats_t* kvfstats)

s32 start_kvf_trans (kv_trans_id_t ** t_id)

s32 commit_kvf_trans (kvf_trans_id_t* t_id)

s32 abort_kvf_trans (kvf_trans_id_t* t_id)
```

2.7.2 Pool接口

```
s32 create_pool (const char* name, const char* config_path,
pool_t * pool)

s32 destroy_pool (pool_t * pool)

s32 open_pool (pool_t * pool)

s32 close_pool (pool_t* pool)

s32 set_pool_prop(const pool_t* pool, const char* name,
const char* value )

s32 get_pool_prop(const pool_t* pool, const char* name,
char** value)
s32 get_pool_stats (pool_stats_t * stats)
```

2.7.3 Pool接口

```
s32 put(pool_t* pool,const string_t* key, const string_t* value,
const kv_props_t* props, const put_options_t* putopts);
```

```
s32 get(pool_t* pool, const string_t* key, string_t* value,
const kv_props_t* props, const get_options_t* getopt)

s32 del(pool_t* pool, const string_t* key, const kv_props_t* props,
const del_options_t* delopts)

s32 mput(pool_t* pool, kv_array_t* kvarray, const kv_props_t* props,
const put_options_t* putopts)

s32 mget(pool_t* pool, kv_array_t* kvarray, const kv_props_t* props,
const get_options_t* getopt)

s32 mdel(pool_t* pool, array_t* kvarray, const kv_props_t* props,
const del_options_t* delopts)

s32 open_iter (const pool_t* pool, const string_t* key_regex,
s32 limit, s32 timeout, kv_iter_t* it)

s32 next_iter (pool_t* pool, kv_iter_t* it,
kv_array_t* kvarray)

s32 close_iter (pool_t* pool, kv_iter_t* it)

s32 xcopy_obj(const pool_t* src, const pool_t* dest,
const string_t* regex)
```

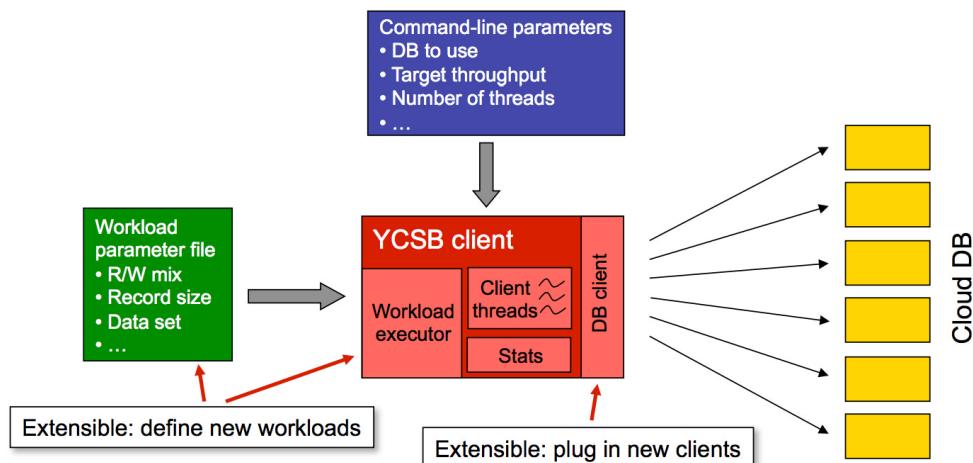
2.8 概览

YCSB(Yahoo! Cloud Serving Benchmark)是用来测试cloud serving/NoSQL/Key-Value Store的测试工具，代码开源，详细介绍见论文Benchmarking Cloud Serving Systems with YCSB。由于云服务的流行，传统数据库不能满足可利用性、可扩展性等要求，因此功能简化、一致性简化的NoSQL数据库逐渐流行。然而NoSQL数据库种类繁多，针对不同目的数据库各有权衡（读写性能、延迟和持久性、同步与异步等等），用户和开发人员需要针对不同的应用场景选择合适的数据库。YCSB的目标是提供一个公平的平台，为这些数据库提供一个统一的测试方案，从而更公平地在不同的方面衡量不同的数据库性能，从而提供有价值的参考。

2.9 YCSB应用模型

YCSB使用java编写，因为很多NoSQL系统拥有Java接口，其他没有Java接口的NoSQL系统可以通过比如HTTP/REST、JNI(Java Native Interface)等方式与java进行连接。由于KVF框架由c语言编写，所以在对KVF进行测试的时候需要用到JNI技术来实现KVF于YCSB的连接。

YCSB模型图如下所示：



可以看出YCSB系统可以分为四个部分，分别是YCSB客户端，云数据库，任务集和命令行参数。

2.9.1 YCSB客户端

YCSB客户端是整个系统的核心，它包括任务集执行器，客户端线程，统计数据，和数据库客户端。

YCSB客户端是一个用来产生数据和操作的Java程序，数据用来被加载到数据库中，操作包括读写更新删除等等，这些操作构成了任务集。基本的运行流程是任务集执行器驱动多个客户端线程，每一个线程执行一系列的操作，这些操作被标记为数据库接口层的调用，包括加载数据库（加载阶段）和执行任务集（交易阶段）。这些线程可以自行调节它们所产生的不同请求的比例，因此我们通过直接编写workload的方式来控制它。这些线程还会测量延迟和这些操作实现的吞吐量，并将这个测量数据报告到统计数据模块中，在每一次实验的结束，统计数据模块负责整合测量数据并报告前95%的延迟和99%的延迟。

客户端将用一系列的属性（属性名/值对）来定义它的操作。通常情况下，我们将这些属性分为两组：

- **任务集属性：**用来定义任务集的一系列属性，独立于一个给定的数据库或者实验运行。比如指定对数据库读写操作的混合比例，指定分布方式，和记录的数量、大小和域的数目等等。
- **运行时属性：**是针对于一个给定的实验的属性集合，比如指定哪一种数据库（比如Cassandra、HBase等），指定初始化接口层的属性（比如数据库服务的主机名），指定客户端线程的数量等等。

因此任务集属性文件可以是静态的，并被应用于不同的不同数据库的测试实验中，运行时属性也一样可以存储在属性文件中，但是针对不通的实验、数据库和目标吞吐量这些属性的值可能会大有不同。

2.9.2 Workload

workload指的是测试数据库系统时需要完成的任务集，在YCSB系统中自带六个workload，分别针对不用的应用场景

Workload A: 更新操作权重很高的任务集

这个任务集是由50%的读操作和50%的写操作混合而成，一个典型的应用例子是会话存储记录最近的行为。

Workload B: 几乎全是读操作的任务集

这个任务集由95%的读操作和5%的写操作混合而成，应用场景为照片打标签，添加标签是更新操作，但是绝大部分的操作是读取tag

Workload C: 全部是读写的任务集

这个任务集100%是读，应用例子是用户简介的缓存，用户的简介实际上是在其他地方创建的。

Workload D: 读最近的任务集

在这个任务集中，新的记录被插入，最新插入的记录是最经常被读取的。一个典型的应用例子：用户更新，其他用户想要看到的是最新修改的内容。

Workload E: 短的排列

在这个任务集中，一小排记录被查询，而不是一个单独的记录，应用例子：一段对话，每一次扫描

Workload F: 读—修改—写

在这个任务集中，客户端将读一个记录，修改它，然后写回。应用例子：用户数据库，用户读取自己的记录然后

2.9.3 JNI

The Java Native Interface (JNI) 是一个Java软件开发工具的一个原声编程接口，通过它可以使得Java代码使用其他语言编写的代码和代码库，比如C和C++。Invocation 接口是JNI的一部分，可以用来讲Java虚拟机（JVM）嵌入到原生应用中，从而允许其他语言的原生应用可以调用Java代码。

在benchmarkd实现的过程中，需要以下工具：

Java编译器：javac执行命令需要SDK

Java虚拟机：java执行命令需要SDK

原生方法C文件产生器：javah命令需要SDK

定义了JNI的库文件和原生头文件：C头文件jni.h、jvm.lib、jvm.dll或者jvm.so都需要SDK

可以产生共享库的C和C++编译器：在linux系统下cc编译器是最常见的

在Java 2 SDK中，Java虚拟机和Java运行时支持位于共享库中的libjvm.so

连接YCSB和KVF需要如下六步：

编写Java代码。我们需要首先编写Java类来完成三个任务：声明将要调用的原生方法，加载包含原生代码的共享库，调用原生方法。

编译Java代码。在使用之前我们必须成功将Java类编译成为二进制代码。

创建C头文件。头文件中声明我们想调用的原生函数，这个头文件将用来和C函数实现一起创建共享库。

写C代码，这一步需要实现C源代码，该文件必须包含在第三步创建的头文件。

创建共享库文件。我们将从第4步获得的C源代码文件创建共享库文件。

运行Java程序。

为了对比KVF底层接于nvmkv后的功能和性能，我分别编写了Makefile来对KVF和nvmkv进行基本的测试，Makefile的实际上是指定了编译规则以及依赖关系，在每一个编译指

令的最开始会指定生成对象的名称，然后指定生成该文件所依赖的文件名，在第二行指定编译工具以及编译细节，在编写Makefile时需要将最后编译的文件在最前面指出，因此整个Makefile的罗列顺序和实际的编译顺序是相反的。如下代码是列出了测试nvmkv时Makefile中关键的步骤，我将按照实际的编译顺序进行介绍。

```
com/yahoo/ycsb/db/JNvmkv.class : com/yahoo/ycsb/db/JNvmkv.java
javac com/yahoo/ycsb/db/JNvmkv.java
```

首先编译JNvmkv对象，这个类对nvmkv的底层函数进行了封装，供YCSB client调用。

```
com/yahoo/ycsb/db/com_yahoo_ycsb_db_JNvmkv.h \
com/yahoo/ycsb/db/JNvmkv.class
javah -classpath . -d ./com/yahoo/ycsb/db/ \
com.yahoo.ycsb.db.JNvmkv
```

这一步生成负荷JNI格式的头文件，其中每一个函数名都以com_yahoo_ycsb_db_开头，在编写Jnvmkv.c文件时，函数名要和这里的保持一致。

```
com.yahoo.ycsb.db.com_yahoo_ycsb_db_JNvmkv.o : \
    com/yahoo/ycsb/db/JNvmkv.c \
    com/yahoo/ycsb/db/nvmkv-test.h
g++ -c $< -o com/yahoo/ycsb/db/com_yahoo_ycsb_db_JNvmkv.o \
$(LDFLAGS) -I/usr/lib/jvm/java/include/ -I/usr/lib/jvm/ \
java/include/linux/ -fPIC
```

这一步生成目标文件，用于最终生成共享函数库文件。

```
com.yahoo.ycsb.db.libcom_yahoo_ycsb_db_JNvmkv.so : \
    com/yahoo/ycsb/db/com_yahoo_ycsb_db_JNvmkv.h : \
    com.yahoo.ycsb.db.com_yahoo_ycsb_db_JNvmkv.o
gcc -I/usr/lib/jvm/java/include/ \
-I/usr/lib/jvm/java/include/linux/ \
```

```

-L$(PATH_TO_ANANAS_LIB) $(LDLIBS) $(LDFLAGS) \
-fPIC -o com/yahoo/ycsb/db/libcom_yahoo_ycsb_db_JNvmkv.so \
-shared -Wl,-soname,com/yahoo/ycsb/db/\
com_yahoo_ycsb_db_JNvmkv.so com/yahoo/ycsb/db/\
com_yahoo_ycsb_db_JNvmkv.o

javah -classpath . -d ./com/yahoo/ycsb/db/ \
com.yahoo.ycsb.db.JNvmkv

```

这一步最终生成YCSB Client在运行时会调用的共享函数库文件。

```

run :

java -Djava.library.path=$(PATH_TO_YCSB)YCSB/nvmkv/src/\
main/java/com/yahoo/ycsb/db/\
com.yahoo.ycsb.db.JNvmkv

```

在最终使用YCSB测试之前，我做了一个单独的基本功能测试，在运行时需要指定库文件路径，一遍调用上一步编译出来的共享函数库文件。

2.9.4 实现方式

YCSB是一套完整的测试系统，我们需要将KVF当作YCSB的一个数据库模块添加到YCSB中。YCSB的数据库接口层隐藏了所在做测试的NoSQL数据库的细节，这就允许客户端产生像“读记录”“更新记录”这样简单的操作，而不用理解你所指定数据库的特定接口。因此，YCSB很容易来测评一个新加入的数据库系统。一旦你创建了数据库接口层，YCSB框架的其他地方不需要做任何改变就可以顺利运行。

YCSB的数据库接口层是一个简单的抽象类，为你的数据库提供了读、插入、更新、删除和扫描操作。为KVF实现一个数据库接口层只需要实现这具体的函数体，一旦这个接口层可以顺利编译，我们就可以在命令行（或者属性文件中）指定为KVF实现的类名。因此我们并不需要因为添加或者更改了数据库接口而重新编译YCSB客户端。

根据下列步骤即可完成KVF模块的创建

第一步：继承 `com.yahoo.ycsb.DB`

该类是所有数据库接口层实现的基类，它是一个抽象类，所以我们需要实现一个继承于com.yahoo.ycsb.DB的类，在这类我们把这个类声明为AnanasClient，这个类需要拥有一个无参数的构造函数，因为在AnanasClient的实例将会在YCSB内部通过无参数的构造函数生成。

YCSB客户端框架会为AnanasClient的每一个线程都创建一个实例，但是我们需要测试多线程的功能，所以会有多个工作者线程都产生任务集，每一个线程都会创建一个AnanasClient实例。

第二步：实现init()如果有必要的话

通过下面的方法我们可以初始化我们的DB对象：

```
public void init() throws DBException
```

该方法在每个AnanasClient实例中都会被调用一次，所以如果在测试的时候使用多线程的方式，每个AnanasClient实例都会被各调用一次。

init()方法应该被用来连接数据库和其他相关初始化工作。需要指出的是，YCSB提供了一种可以在运行时使用属性文件配置数据库的方式。事实上，YCSB客户端会在启动的时候将在属性文件指定的所有参数文件都传递到数据库接口层。因此，我们可以为数据库接口层创建新的属性文件来，在参数文件中或者命令行中进行设置，然后在我们自己实现的AnanasClient中获得这个参数文件并设置相应的属性。

这些属性将会在构造函数被调用后传递到实例中，因此我们需要在init()方法中获得这些属性而不是在构造函数中，我们可以调用下列方法，该方法已经被实现并从DB基类继承而来。

第三步：实现数据库查询更新方法

下列方法是AnanasClient需要实现的：

```
//Read a single record
public int read(String table, String key, Set<String> fields,
HashMap<String, String> result);

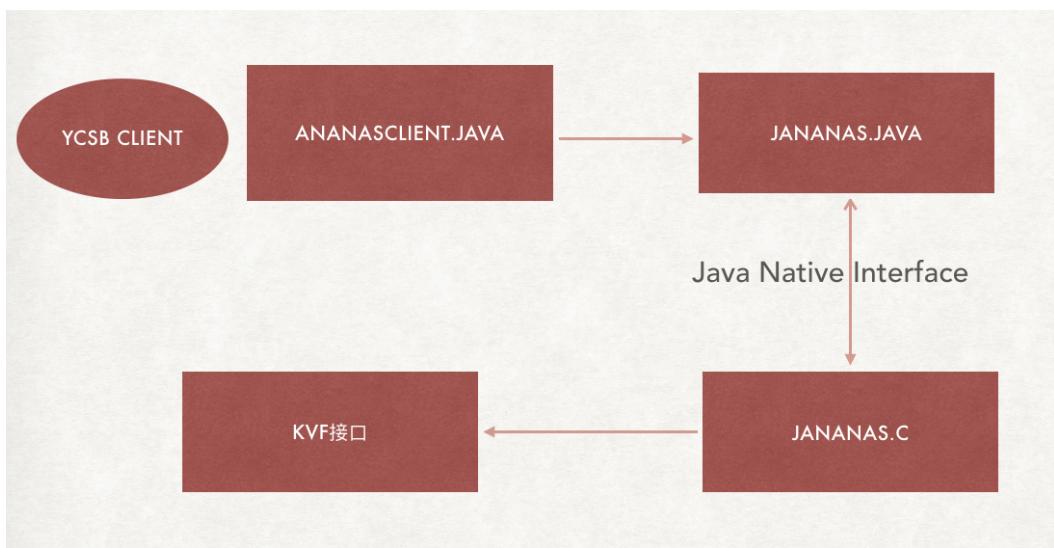
//Perform a range scan
public int scan(String table, String startkey, int recordcount,
Set<String> fields, Vector<HashMap<String, String>> result);
```

```
//Update a single record  
public int update(String table, String key, HashMap<String,  
String> values);  
  
//Insert a single record  
public int insert(String table, String key, HashMap<String,  
String> values);  
  
//Delete a single record  
public int delete(String table, String key);
```

在每一次调用的时候，这些方法都会使用一个表名称和记录的键。对于读和扫描方法，需要额外的参数来提供一个数据结构存储返回的数据。对于插入和更新方法来说，需要传入映射了域名和值的哈西映射（HashMap）。

数据库应该在运行测评之前创建合适的表。我们在实现上述方法的时候认为适当的表已经存在，只需要从指定的表中读和写。

在这里我将介绍KVF对应的YCSB Client的实现细节。



如上图所示，JAnanasClient是一个YCSB Client的实例，它继承了com.yahoo.ycsb.DB类，并实现了需要实现的函数，在实现的过程中使用JAnanas类进行具体的对数据库的操作，JAnanas类将KVF封装成为一个数据库，并通过JNI与JAnanas.c进行连接，从而调用KVF中实现的接口，达到访问数据库的目的。

第四步：编译数据库接口层

我们的代码可以独立于YCSB客户端和YCSB框架单独编译，因此当我们对自己的数据库接口层AnanasClient作修改之后只需编译自己的模块而不用重新编译YCSB客户端。

第五步 测试数据库

YCSB提供了简单的命令行客户端来测试接口层，它创建一个数据库实例，允许你操作数据库而不用启动一个任务集，例子如下：

```
% java com.yahoo.ycsb.CommandLine -db com.yahoo.ycsb.db.MongoDbClient  
-p mongodb.url=mongodb://localhost:27017 -p mongodb.database=ycsb  
  
YCSB Command Line client  
Type "help" for command line help  
Start with "-help" for usage info  
Connected.  
> insert brianfrankcooper first=brian last=cooper  
Return code: 1  
191 ms  
> read brianfrankcooper  
Return code: 0  
last=cooper  
_id=brianfrankcooper  
first=brian  
2 ms  
> quit
```

首先使用命令行工具以及相关的参数启动一个数据库，然后进行单独的插入、读、更新、删除等操作，这样即可自行验证功能的正确性。比如在上述实例中，首先插入一条记录，`id`为`brianfrankcooper`，第一个值为`brian`，第二个值为`cooper`，然后读取记录，读出的信息和插入的一样，验证成功。

第六步：使用YCSB客户端测试

使用YCSB集成测试需要在YCSB项目的根目录下运行ycsb的python脚本，在命令行指定数据库和任务集以及线程等相关信息。

2.9.5 实现流程

1. 在YCSB中新建一个文件kvf/, 将KVF作为一个新的Database添加到YCSB框架中
2. 在YCSB/kvf/目录下新建pom.xml文件用于编译，文件内容和YCSB中其他database的相应文件一样，只需做与kvf相关的修改即可
3. 在YCSB/kvf/目录下仿照其他database的目录结构新建src/main/java/, 在这个目录下创建Makefile用于简化编译过程
4. 在YCSB/kvf/src/main/java目录下新建com/yahoo/ycsb/db/目录结构，实现java的包结构方便进行编译
5. 在这个目录下新建文件AnanasClient.java，该类继承了com.yahoo.ycsb.DB类，在该类中重写了一些关键的方法来供YCSB的上层测试函数的调用
6. 创建JAnanas.java用于对kvf的包装
因为kvf用c语言编写，YCSB为java编写，所以需要使用Java Native Interface来将其连接
7. 通过JAnanas.java编译出头文件com_yahoo_ycsb_db_JAnanas.h
根据编译出的头文件编写com_yahoo_ycsb_db_JAnanas.c实现JAnanas.java中的native函数
8. 在YCSB/bin中的ycsb脚本文件第75行添加kvf
9. 在YCSB/pom.xml中添加kvf
10. 在distribution/pom.xml中添加kvf

2.9.6 测试计划

在图形界面客户端中进行测试对多线程进行测试：将线程数分别设置为1, 2, 4, 8, 16, 21, 64, 128分别测试。

通过参数设置配置不同读、插入、更新、删除操作比例的测试文件进行测试：例如：不同读写比率（100%插入、90%更新+10%读、65%读+25%插入+10%更新、90%读+10%插入、100%读等）

如下我可以自定义任务集，指定操作数、插入、更新、读取、删除的比例，以及分布方式：

```
workload=com.yahoo.ycsb.workloads.CoreWorkload
readallfields=false
```

```

readproportion=0.65
updateproportion=0.25
scanproportion=0
insertproportion=0.1
requestdistribution=uniform

```

在这个任务集中首先会将1000个记录加载到数据库中，在实际测试的时候将有2000次操作，读取数据的比例是65%，更新的比例为25%，插入的比例为10%，分布方式为全部随机分布，在命令行中指定线程数为16.

通过YCSB的stat结构统计数据，我们可以得到如下的测试结果：

```

[OVERALL], RunTime(ms), 516.0
[OVERALL], Throughput(ops/sec), 3875.968992248062
[CLEANUP], Operations, 1.0
[CLEANUP], AverageLatency(us), 5.0
[CLEANUP], MinLatency(us), 5.0
[CLEANUP], MaxLatency(us), 5.0
[CLEANUP], 95thPercentileLatency(us), 5.0
[CLEANUP], 99thPercentileLatency(us), 5.0
[INSERT], Operations, 216.0
[INSERT], AverageLatency(us), 554.5185185185185
[INSERT], MinLatency(us), 283.0
[INSERT], MaxLatency(us), 1852.0
[INSERT], 95thPercentileLatency(us), 961.0
[INSERT], 99thPercentileLatency(us), 1554.0
[INSERT], Return=0, 216
[READ], Operations, 1302.0
[READ], AverageLatency(us), 89.02150537634408
[READ], MinLatency(us), 34.0
[READ], MaxLatency(us), 701.0
[READ], 95thPercentileLatency(us), 206.0
[READ], 99thPercentileLatency(us), 288.0
[READ], Return=0, 1302
[UPDATE], Operations, 482.0

```

```
[UPDATE], AverageLatency(us), 143.00829875518673
[UPDATE], MinLatency(us), 69.0
[UPDATE], MaxLatency(us), 841.0
[UPDATE], 95thPercentileLatency(us), 236.0
[UPDATE], 99thPercentileLatency(us), 298.0
[UPDATE], Return=0, 482
```

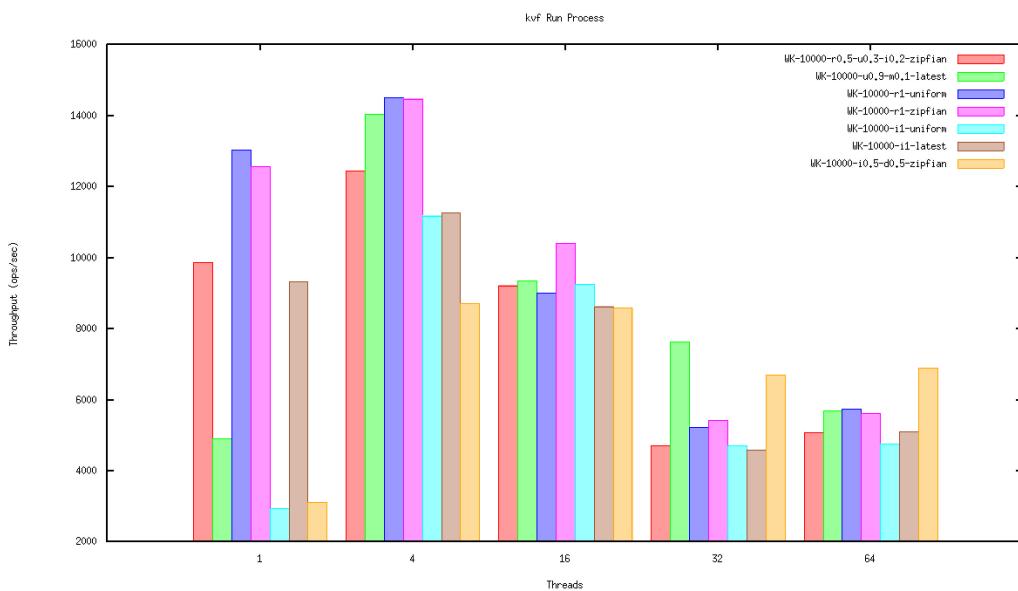
从结果中我们可以看到，吞吐量为3875.96ops，由于有16个线程所以会清除16次。

插入操作有216次，平均延迟为554.51us，最大一次延迟为1852.0us，最小的一次延迟为283.0us，99%的插入延迟在1554.0us以内，95%的插入延迟在961.0us%以内。

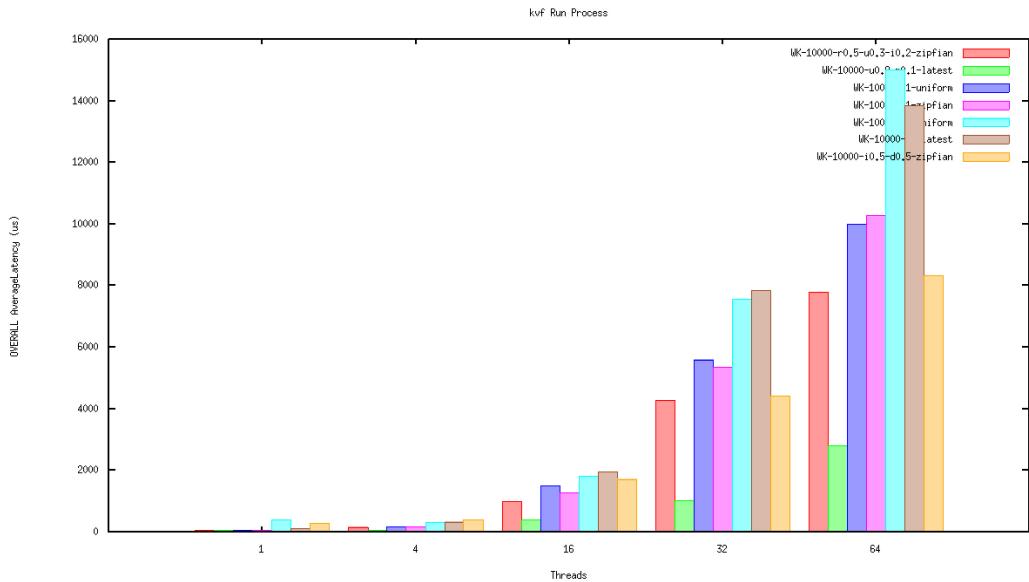
读操作有1302次，平均延迟为89.02us，最大一次延迟为701.0us，最小的一次延迟为34.0us，99%的读延迟在288.0us以内，95%的读延迟在206.0us%以内。

更新操作有482次，平均延迟为143.00us，最大一次延迟为841.0us，最小的一次延迟为69.0us，99%的读延迟在298.0us以内，95%的读延迟在236.0us%以内。

为了更直观地分析和对比数据，我将YCSB的测试结构进行可视化处理。



图中测试了六种不同的任务集分别在1、4、16、32、64线程中总体的吞吐量统计图。除了总体的数据，还可以短读显示插入、读、删除等每一种操作单独的统计数据。



该图是同样测试的延迟统计。可以看出当线程数增加时延迟会越来越大。

结论

根据实际的使用情况，KV接口标准定义的API的语义概括了通用的KV模型，使上层应用可以使用一个统一的API接口来访问下层的KV存储。因此，KVF在实际的应用过程中有希望极大简化基于KV模型的应用的开发过程。另外KVF具有很好的扩展性，在上层应用开发的过程中可以灵活地根据自己的实际需要增加参数配置，定义自己的所需要的特定的数据结构，从而满足不同层次的需求。

YCSB作为一个提供键值存储测试的平台，可以供很好的提测试方案，并且提供了易于使用的任务集配置方案，将其统计结果可视化处理之后我们可以很直观地对比不同任务集的各项数据，从而更好地进行分析。将YCSB和KVF进行连接是可行的，并且还可以直接与磁盘访问的底层函数相连接，从而可以分析和对比不同存储系统的性能损耗。

致谢

随着论文的结束，我的大学生涯也接近尾声，在这里我要感谢大学四年里帮助过我的各位老师、同学、师兄师姐们，正是你们的支持和帮助才使得我顺利度过了人生中国年最为重要的四年。

首先要非常感谢我的班主任谢冰老师，在我困惑迷茫的时候给我提出很好的建议，并在很多地方给我很大的帮助。

感谢孙广宇老师帮助我选择这样一个具有实际意义并且具有一定挑战性的毕业设计题目，如果不是这次毕业设计，我可能不会对键值存储数据库有这么深的了解，也不会对Linux系统有如此深入的探索。同样非常感谢在完成毕设过程中耐心指导我的王鹏师兄和王晓阳师兄，没有你们的帮助我也不会如此顺利地完成自己的题目。

感谢孙广宇和李文新老师在申请季给予的帮助和支持。

感谢大学各位朋友们的鼓励和帮助，特别是在羽协和山鹰社认识的各位好友，因为你们我的大学生活才如此精彩。

最后要感谢家人们默默的付出和关怀。

北京大学学位论文原创性声明和使用授权说明

原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不含任何其他个人或集体已经发表或撰写过的作品或成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本声明的法律结果由本人承担。

论文作者签名： 日期： 年 月 日

学位论文使用授权说明

(必须装订在提交学校图书馆的印刷本)

本人完全了解北京大学关于收集、保存、使用学位论文的规定，即：

- 按照学校要求提交学位论文的印刷本和电子版本；
- 学校有权保存学位论文的印刷本和电子版，并提供目录检索与阅览服务，在校园网上提供服务；
- 学校可以采用影印、缩印、数字化或其它复制手段保存论文；
- 因某种特殊原因需要延迟发布学位论文电子版，授权学校在一年/两年/三年以后在校园网上全文发布。

(保密论文在解密后遵守此规定)

论文作者签名： 导师签名： 日期： 年 月 日