华中科技大学

Huazhong University of Science and Technology

课程实验报告

课程名称：大数据存储与管理

专业班级：校交 1901 班

学　　号：U201910681

姓　　名：骆瑞霖

指导教师：华宇、施展

报告日期：2022-04

计算机科学与技术学院

目录

1　Part1: 实验目的

- 使用工具搭建对象存储环境，包括服务端和客户端，并能完成文件数据的上传于下载；
- 在第一点的基础之上，使用评测工具如 COSBench 或者 s3bench 在本机上进行数据传输的性能观测，或者通过编程实现；
- 了解对冲和关联请求，了解尾延迟。

2　Part1: 实验背景

实验的服务端首先考虑了使用 Openstack swift 和 Ceph 项目搭建，但由于所给范例维护时间过早，未能成功启用，外加没有相关项目使用经验，遂放弃，转而使用模仿 Amazon S3 的 mock-S3 工具进行性能评测的服务端搭建，评测过程使用了所给实验文档提供的 S3 bench 进行测试，随后使用 Boto3 库进行 Python 编程, 在 Jupyter 中编写代码完成实验测试。

3　Part1: 实验环境

软硬件环境	
操作系统	windows 10 家庭版
CPU	Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz 2.40 GHz
内存	8.00GB
Python	3.7.6
boto3	1.18.31
boto	2.49.0
Jupyter	1.0.0

4　Part1: 系统搭建与性能观测

4.1　服务端搭建

服务端使用 mock-S3 搭建，该工具需要提供用作服务端的本地端口和数据存放位置，有很多选择，hostname 使用 127.0.0.1，端口使用 5500，执行 main.py 程序就搭建好了服务端，如图所示：

```
D:\BigdataStorage\mock-s3\mock_s3>python main.py --hostname 127.0.0.1 --port 5500 --root ./root
Starting server, use <Ctrl-C> to stop
```

图 1: 服务端搭建

4.2 S3-bench 性能评测

S3-bench 是一个对于指定服务端进行对象存储上传下载服务并监视过程中速率、延迟和吞吐率的功率，使用非常简单，只要指定正确的参数即可完成相应的任务，完成总结报告；

参数解释	
endpoint	上传端口
bucket	对象存储桶名
objectNamePrefix	桶中实例名
numClients	模拟客户端数
numSamples	上传实例数
objectSize	测试文件大小，单位为 B

可以通过编写 run.cmd 进行修改参数循环测试，输出结果进行可视化。测试的样例结果如下:

命令提示符 - run.cmd

```
Results Summary for Write Operation(s)
Total Transferred: 1024.000 MB
Total Throughput:  30.88 MB/s
Total Duration:    33.157 s
Number of Errors:  0
------------------------------------
Write times Max:       1.066 s
Write times 99th %ile: 1.061 s
Write times 90th %ile: 1.048 s
Write times 75th %ile: 1.041 s
Write times 50th %ile: 1.034 s
Write times 25th %ile: 1.029 s
Write times Min:       1.016 s


Results Summary for Read Operation(s)
Total Transferred: 1024.000 MB
Total Throughput:  821.50 MB/s
Total Duration:    1.246 s
Number of Errors:  0
------------------------------------
Read times Max:       0.059 s
Read times 99th %ile: 0.058 s
Read times 90th %ile: 0.049 s
Read times 75th %ile: 0.044 s
Read times 50th %ile: 0.038 s
Read times 25th %ile: 0.033 s
Read times Min:       0.016 s
```

图 2: 部分测试样例

首先选用 8 并发数，256 个实例进行测试，对象大小梯度采样，选用 1KB,2KB,4KB,16KB,64KB,256KB,1MB,4MB,8MB 作为测试对象；得到测试结果如下：

size	max-write	throughput1	duration	max-read	throughput2	duration
1kb	0.02	0.78MB/s	0.321s	0.511	0.49MB/s	0.511s
2kb	0.018	1.6MB/s	0.312s	0.505	0.94MB/s	0.529s
4kb	0.019	1.94MB/s	0.315s	0.509	1.94MB/s	0.515s
16kb	0.018	12.31MB/s	0.325s	0.511	7.52MB/s	0.532s
64kb	0.018	48.05MB/s	0.333s	0.511	27.75MB/s	0.577s
256kb	0.023	170.21MB/s	0.376s	0.518	116.17MB/s	0.551s
1MB	0.526	441.38MB/s	0.580s	0.509	598.13MB/s	0.509s
4MB	1.072	31.06MB/s	32.965s	0.560	909.42MB/s	1.126s
8MB	1.151	60.20MB/s	34.021s	0.573	783.26MB/s	2.615s

图 3: 测试结果，numClients=8,numSamples=256

绘制主要观测量的变化趋势图：观察平均最大读写时间曲线，

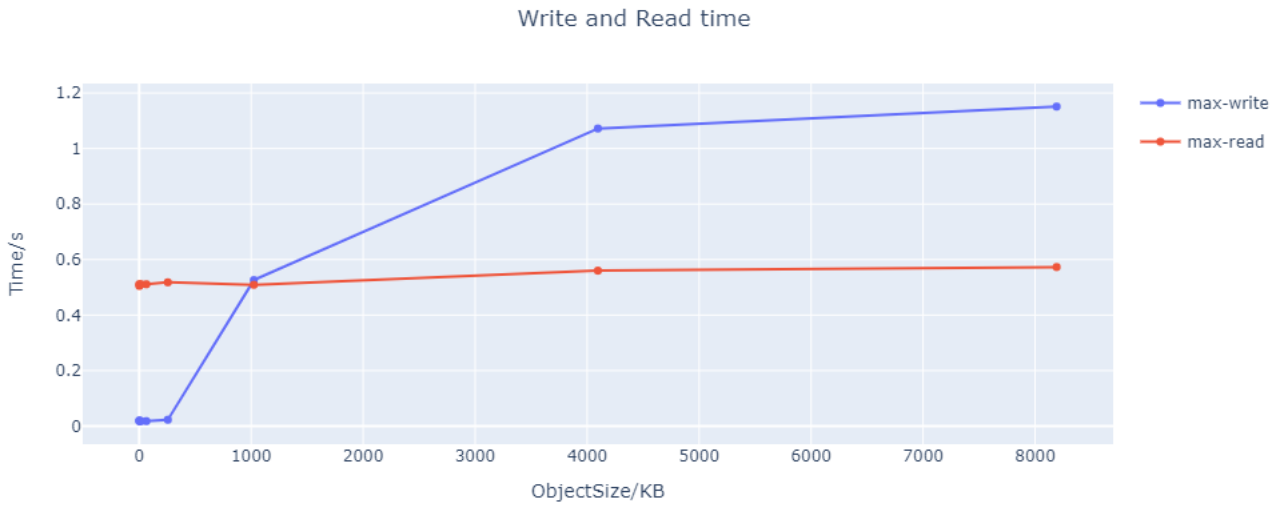

图 4: Max read and write 曲线

可以发现随着对象尺寸的增大，平均读取时间并没有较大的改变，也就是说，仅仅就观察到的数据而言，存储对象尺寸并没有对读取效率产生较为明显的影响；再观察吞吐率的变化曲线：

图 5: 吞吐率变化曲线

可以观察到随着对象尺寸的不断增大，读写两种方法的吞吐率均呈现先曾大后减少的趋势，往后趋于平缓，可以认为起初文件较小时文件传输并无明显压力，多增加对象尺寸便可以多传输对象文件，但是随着文件大小超出一定范围，超过运载负荷后吞吐量将趋于稳定。

最后观察延迟现象：

图 6: 延迟时间变化曲线

可以观察到，随着对象存储大小的增加，延迟稳步增加，在写操作中可以发现对象大小从 1MB 增大到 4MB 这个过程中延迟时间产生了非常大的延长，读操作的增加也较为明显。

针对并发数进行探究，以 numClients 减少为 4 为例，选用相同的实例数和存储对象大小，

得到测试结果如下所示:

size	max-write	throughput1	duration	max-read	throughput2	duration
1kb	0.013	0.80MB/s	0.314s	0.008	0.93MB/s	0.27s
2kb	0.034	1.37MB/s	0.366s	0.007	1.86MB/s	0.269s
4kb	0.012	3.09MB/s	0.324s	0.008	3.67MB/s	0.272s
16kb	0.011	12.30MB/s	0.325s	0.008	14.71MB/s	0.272s
64kb	0.012	48.21MB/s	0.332s	0.009	56.18MB/s	0.285s
256kb	0.019	162.9MB/s	0.393s	0.008	223.43MB/s	0.286s
1MB	0.027	397.64MB/s	0.644s	0.014	574.76MB/s	0.445s
4MB	1.049	15.56MB/s	65.797s	0.032	851.16MB/s	1.203s
8MB	1.102	30.53MB/s	67.077s	0.071	756.96MB/s	2.706s

图 7: 测试结果，numClients=4,numSamples=256

对比两种并发情况下的吞吐率:

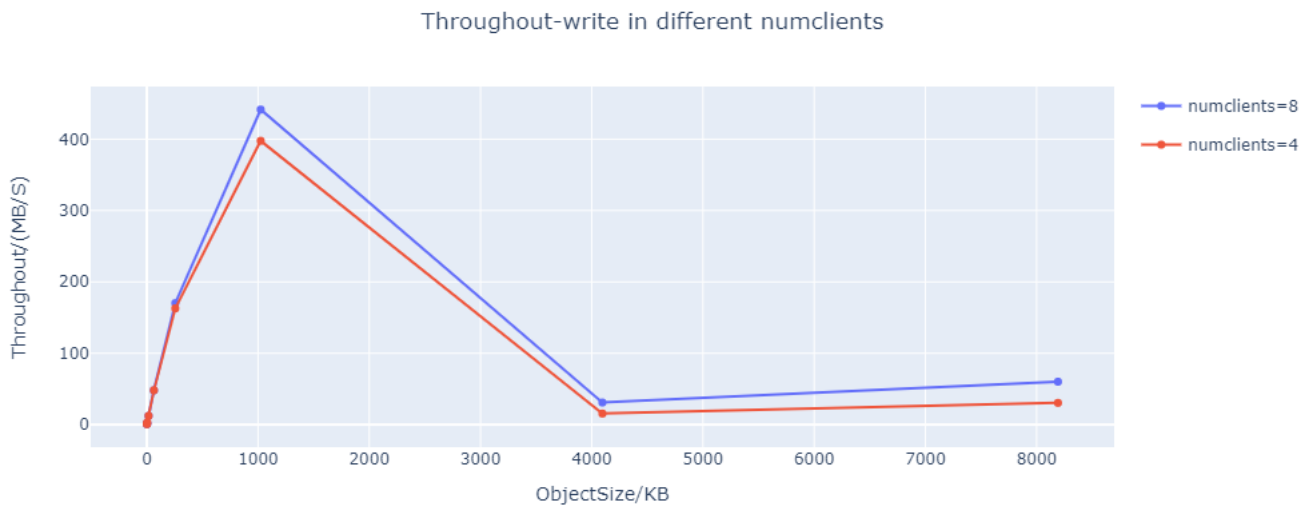

图 8: Throughout-Write 对比

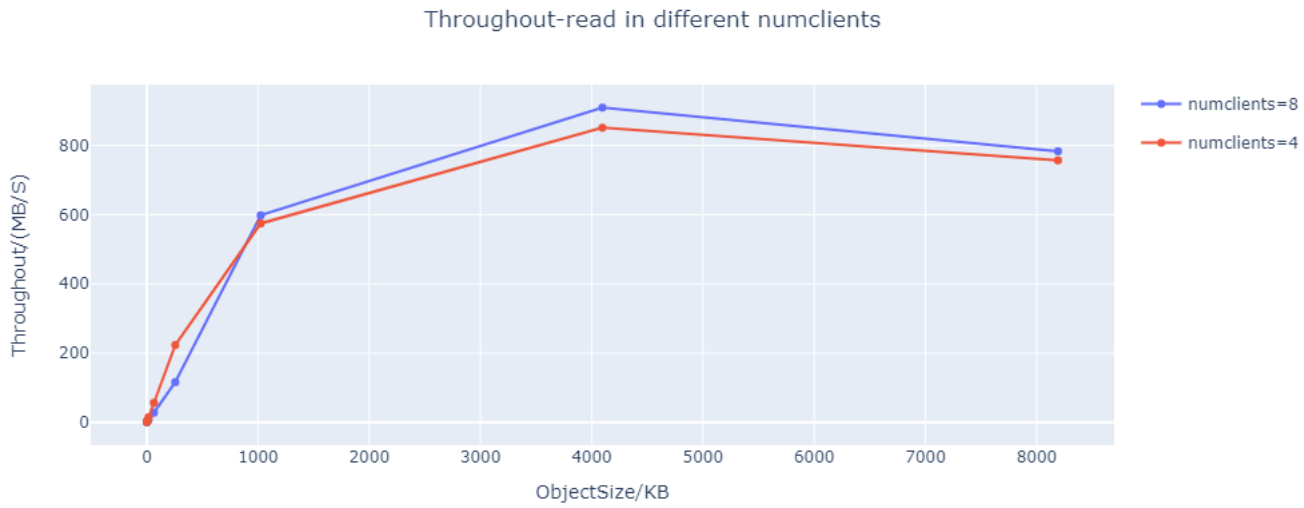

图 9: Throughout-Read 对比

可以观察到不同并发数下，吞吐量的变化曲线几乎相同，平均情况下并发数高的吞吐率表现较好，再来观察我们所关心的并发数带给延迟分布的影响：

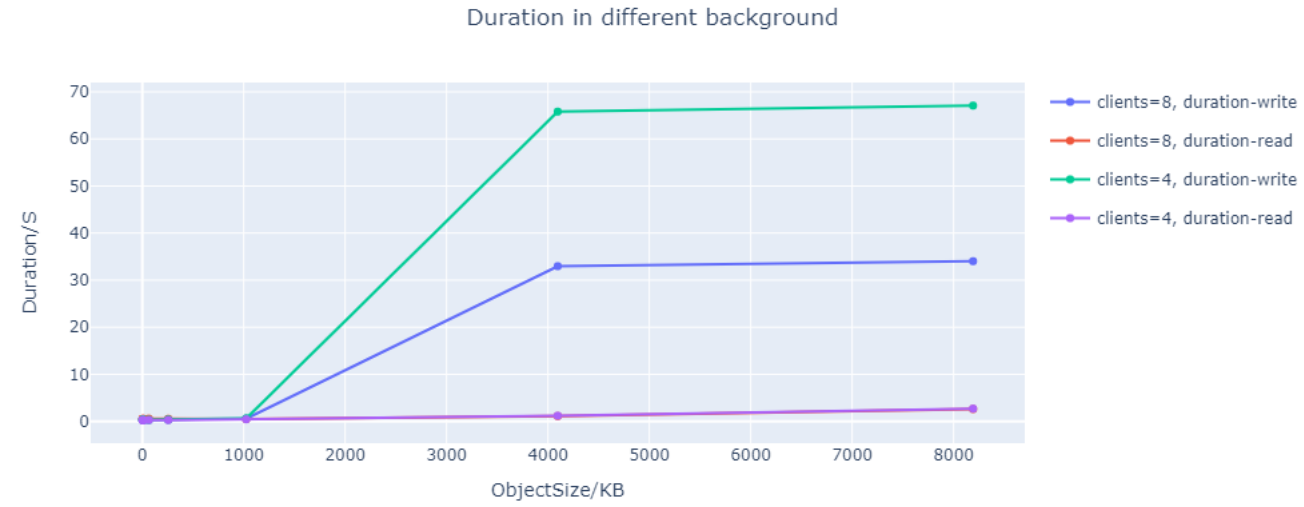

图 10: Durations in different condition

可以看到，read 操作的延迟都比较小，两种并发数下的曲线几乎重合，但是 write 操作下的延迟在对象大小较大时却有明显差距，而且 numclients=4 时的延迟时间几乎恰好为 numclients=8 时的一半，因此根据数据显示可以近似认为并发数对延迟分布的影响是线性的。

4.3 Boto3 性能观测

除了使用 S3-bench 之外，可以使用 Boto3 编程实现传输性能观测，这里以观察延迟作为示例。Boto3 是一个 Python 库，可以利用它提供的 API 完成对象存储的实践操作。实验过程大致如下：

- 初始化本地数据文件

```python
# Python code
# 初始化本地数据文件
local_file = "_test_4K.bin"
test_bytes = [0xFF for i in range(1024 * 1024 * 1)] # 填充至所需大小

with open(local_file, "wb") as lf:
    lf.write(bytearray(test_bytes))
```

该示例给出了 1MB 测试文件的生成；

- 基础设置

```python
# Python code
# 初始化本地数据文件
botocore_config = botocore.config.Config(max_pool_connections=1)
s3client = boto3.client('s3', config=botocore_config)
transfer_config = s3transfer.TransferConfig(
    use_threads=True,
    max_concurrency=1,
)
```

该过程配置了线程池中线程数，客户端类型以及传输器 transfer 的基础配置

- 创造实例

```python
s3t = s3transfer.create_transfer_manager(s3client, transfer_config)
```

传入 client 和 transfer 配置生成对象存储工具实例

- 传输与性能观测

```python
total_size = 1024 * 1024
progress = tqdm.tqdm(
    desc='upload',
    total=total_size, unit='B', unit_scale=1,
    position=0,
    bar_format='{desc:<10}{percentage:3.0f}%|{bar:10}{r_bar}')
ans = []
for i in range(100):
    obj_name = "testObj%08d"%(i)
    st = time.time()
    s3t.upload(local_file, bucket_name, dst, subscribers=[
            s3transfer.ProgressCallbackInvoker(progress.update),
        ])
```

```
    ed = time.time()
    ans.append(ed - st)
s3t.shutdown()
progress.close()
```

该过程首先创造一个监视过程的 tqdm 进度条，进行 100 个对象实例的传输，每次传输使用生成的 local_file 作为存储对象。

有了上面的基础之后，只需要动态调整传输文件的大小，就可以对不同配置下的对象存储任务进行延迟的观测；得到测试结果如下：

图 11: Boto3 测试延迟

可以看到延迟分布曲线与 s3-bench 大致一致，但是发现数据大小都要稍大，具体到几十毫秒到几秒不等，推测是由于 Python 自身函数调用稍慢所致。

4.4 思考问答

对象尺寸，并发数的影响在前文测试结果处有分析，不再赘述。

• 对于熟悉的某类应用，根据其数据访问特性，怎样适配对象存储最合适?

个人体验是对于鼓励内容加速分发的场景，比较适合使用对象存储，进而包括海量数据的高并发，基于对象存储可获得更优的读取、分析性能，更强的稳定性和更好的扩展性。

• 尾延迟的影响因?

主机系统其他进程的影响，应用程序里线程调度，CPU 功耗设计，此外根据结果曲线，传输对象大小也一定程度上与尾延迟大小有关。

• 横向扩展效果如何?

横向扩展可以添加若干并行工作的节点并作为一个节点进行管理，从而实现吞吐量和容量

的独立扩展。Scale-out 虽能突破单机限制，但也会引入一些问题，比如如何在部分节点故障时仍能保持高可用性 (HA)，以及当多个节点有状态需要同步时如何保证状态信息在不同节点的一致性。

5 Part1: 尾延迟挑战

5.1 基础知识

在网络实际情景中，单个服务组件的长尾高延迟可能由于许多原因产生;

• 机器共享资源，机器可能被争夺共享资源的不同应用所共享，而在同一应用中，不同的请求可能争夺资源。

• 守护程序，后台守护程序可能平均只使用有限的资源，但在运行时可能会产生几毫秒的峰值抖动。

• 全局资源共享，在不同机器上运行的应用程序可能会争抢全局资源。

• 排队，中间服务器和网络交换机中的多层队列放大了这种可能性。

而对于高尾延迟，可以采取一些自适应手段;

• 对冲请求，抑制延迟变化的一个简单方法是向多个副本发出相同的请求，并使用首先响应的结果。一旦收到第一个结果，客户端就会取消剩余的未处理请求。不过直接这么实现会造成额外的多倍负载。所以需要考虑优化。一个方法是推迟发送第二个请求，直到第一个请求到达相当高的分位数还没有返回。这种方法将额外的负载限制压的很小，同时大大缩短了长尾时间。

• 捆绑式请求，不像对冲一样等待一段时间发送，而是同时发给多个副本，但告诉副本还有其它的服务也在执行这个请求，副本任务处理完之后，会主动请求其它副本取消其正在处理的同一个请求。需要额外的网络同步。

5.2 对冲请求

当请求在指定的时间间隔后没有返回时，会发起对冲请求，继续等待，如果依然没有返回，则重复发送直到接收到返回结果或者超时取消，因此，使用 boto3 实现，设计一个 s3transfer 数组作为等待发送的上传器，设置等待时间逐步增大，这样，当第一个请求发送后，若未按时完成请求任务从而执行循环 shutdown 语句的话，就会从数组中取出下一个 transfer 发送请求。主要代码实现如下:

```
1 import threading
2 transfers = []
3 for i in range(10):
4     transfers.append(s3transfer.create_transfer_manager(s3client,
          transfer_config))
5 ans = []
6 for i in range(100):
7     obj_name = "testObj%08d"%(i)
8     st = time.time()
9     st_time = 0.9
```

```
    for j in range(10):
        threading.Timer(st_time * 1.1, transfers[j].upload(local_file,
            bucket_name, dst, subscribers=[
            s3transfer.ProgressCallbackInvoker(progress.update),
        ]))
    s3t.upload(local_file, bucket_name, dst, subscribers=[
            s3transfer.ProgressCallbackInvoker(progress.update),
        ])
    for j in range(10):
        transfers[j].shutdown()
    ed = time.time()
    ans.append(ed - st)
s3t.shutdown()
progress.close()
```

对相同大小的对象尺寸进行重复实验，与任务二的结果相对比：

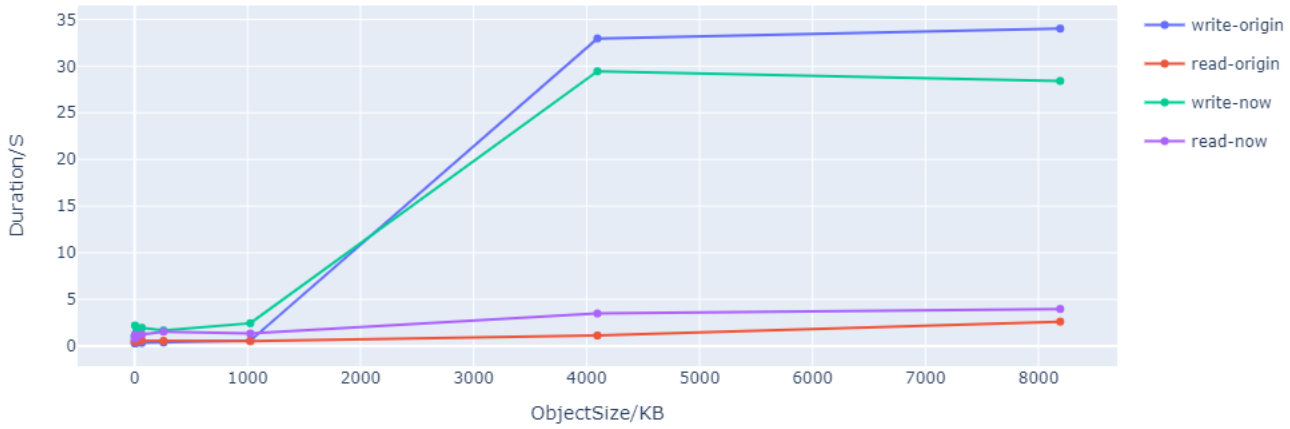

图 12: 对冲前后时延分布

最直观的感受是对象尺寸较小时对冲请求后延迟反而比原来的要长，可能是程序设计还能进一步优化，编程实现的调用时间以及建立连接占了主导，但是可以发现，整体发展趋势与原曲线几乎相同但是在高尺寸下有较为明显的优化效果，大概下降了百分之十几。

6 Part2: Multi-Probe LSH

6.1 方法

LSH 常用于海量高维数据的近似最近邻的快速查找，LSH 的基本思想是相似度高的数据经过 HashTable 的映射后的数据也具有比较高的相似度，这样在对于给定的查询，只需要先找到待查询数据所映射入的多个桶中的其他元素，然后再进行距离的计算与排名，这样就把搜索范围从大的全局空间放缩到了一个很小的数据范围内，以牺牲一定的准确度来换取搜索的高效率。但是 cm 的 LSH 方法需要构造非常多的 HashTable 来提高算法的准确率。

论文《Multi- Probe LSH: Efficient Indexing for High-Dimensional Similarity Search》提出的方法减少了 HashTable 的数量，在更为局限的映射空间中通过搜索“接近”的桶，来提高算法的召回率，同时减少了算法的空间开销。

6.2 实现

•Hash perturbation vector

在 Multi-Probe LSH 中，对于查询点 q，定义扰动序列向量 $\Delta = \delta_1, \delta_2, \cdots, \delta_m$，每一次探寻，不再是探寻桶 $g(q_i)$，而是 $g(q_i) + \delta_i$，$\delta_i \in \{-1, 0, 1\}$，对于多个扰动序列可以生成多个桶集合，这些在各个 HashTable 中“相邻”的桶中很可能含有与 q 邻近的元素。

•Step-Wise Probing

定义 n 步扰动向量在 n 个位置上有非零值，根据局部敏感哈希函数的性质，1 步距离的哈希表中的桶要比更远距离的桶更有可能包含 p 的邻近点的，因此，先检查 1 步距离的桶，然后再找二步距离，等等，直到我们找到足够的哈希桶。论文中实践证明，一般情况下 2 步内的步进式探寻足够满足大概率的成功可能。论文中通过图示说明，绝大多数 k 近邻点 (k=20,40,60,80,100)，都被映射到了查询点两步距离内的桶里。

•Success Probability Estimation

在步进式查询中，每一个哈希值位置被扰动的概率相同，扰动的值决定的概率也相同。每一个哈希函数都会将查询值映射到一个直线上，在映射区域内，邻近点总是会落入与待查询点相同或者相邻的 slot 中，实验描述了待查询点的近邻点落在其相邻区域内的概率，经过数学推导得到一个扰动向量能够产生查询点 q 附近的扰动点的可能性，并进行评分。

$$score(\Delta) = \sum_{i=1}^{m} x_i(\delta_i)^2$$

score 越小的 perturbation vector 有更大的概率得到邻近 q 的点集。

•Query-Directed Probing Sequence

由于我们得到了扰动序列分数的计算方法，一种方式是获取所有的扰动向量并按得分从小到大来选取最终可用的向量，但是这种方法的问题是向量数过多，如果能够准确地生成 score 较小的那部分向量就更好了。因此论文中提出一种按得分升序生成扰动序列集合的方法。方法

涉及两个关键操作：

1. shift(A): 这个操作用元素 1+max(A) 来替代集合 A 中的元素 max(A), e.g. shift({1,3,4})={1,3,5}
2. expand(A): 这个操作扩充集合，将 1+max(A) 添加到集合中，e.g.shift({1,3,4})={1,3,4,5}.

•Optimized Probing Sequence Construction

上一节所阐述的扰动序列构造方法通过维护一个堆以及在查询时查询堆来生成扰动向量。除此之外，作者还在文中描述了一种方法来在查询时维护和查询这个堆的开销。该方法预先计算一组序列来替代在查询时对堆的查询以及插入，从而减少扰动序列的生成时间。注意到扰动序列的生成可以被分为两个部分：(1) 按顺序生成扰动集合，(2) 将每个扰动集合映射为一个扰动向量。第一个部分需要值 z_j, 而第二个部分需要一个映射 π 将集合 1,⋯,2K 映射到 (i,δ) 对. 事实上我们可以预先精确的知道值 z_j 的分布并对于每个 j 计算 $E[z_j^2]$, 所以，我们可以做如下优化：我们通过这个期望来近似值 z_j^2。

使用这个近似，可以事先按顺序计算扰动集合 (因为集合的分数是值 z_j^2 的函数)。这个生成的过程可以于上一节描述的过程一样，但是使用值 $E(z_j^2)$ 来替代它们真实的值，这个可以独立于查询点 q 完成。在查询时，根据查询点 q 来计算映射 π(每个哈希表是不同的)。

7　Part2: 数据结构设计与测试

下面给出程序设计的版块的数据结构，成员变量和函数的解释见注释；

7.1　Indexing 部分

• HashBucket

HashBucket 部分需要记录归属在本 HashTable 中各个独立的 bucket 里的数据，同时完成计算该桶的 hash 值以及在应用 perturbation 时返回邻近桶的的任务；

```
1  public class HashBucket implements Serializable {
2
3          private static final long serialVersionUID = 4L;
4          private List<Integer> objectHashBucket; // 存储桶中元素列表
5          private int hashCode; // 桶对应哈希值
6          public HashBucket(List<Integer> objectHashBucket) {
7                  this.objectHashBucket = objectHashBucket;
8                  this.hashCode = computeHashCode();
9          }
10         +
11         private int computeHashCode() {} // 计算哈希值函数
12         public boolean equals(Object other) {} // 重写equals，以便可以唯一标识
               HashBucket对象
```

```
13          public HashBucket getNeighboringBucket(List<Integer> perturbation) {}
                // 在应用扰动后返回一个邻近的HashBucket
14  }
```

•HashFunction

哈希函数类需要包含函数中的常数参数和数据维度数据成员，以及生成函数和根据函数计算返回值的函数成员。

```
1   public class HashFunction implements Serializable {
2
3           private static final long serialVersionUID = 3L;
4           private int numberOfDimensions; // 维度大小
5           private List<Double> hashFunctionCoefficients; // 哈希函数系数a
6           private double offset, slotWidthW; // 偏移b, slot宽度W
7
8           public HashFunction(int numberOfDimensions, double slotWidthW, Random
                randomNumberGenerator) {} // 生成哈希函数的方法
9           public int getSlotNumber(List<Double> objectFeatures) {} // 返回位置敏
                感哈希的插槽号
10          public double getF(List<Double> objectFeatures) {} // 传入object的特征
                值，套入哈希函数参数计算值
11
12  }
```

•HashTable

HashTable 类需要包含哈希函数数量、hashTable 中的多个具体的哈希函数 HashFunction 类列表等数据成员，并能完成根据 object 定位到对应 Bucket 的映射，并完成 object 加入 HashTable 中的桶、根据 bucket 返回其中包含的 object 等成员函数。

```
1   public class HashTable implements Serializable {
2
3           /**
4            *
5            */
6           private static final long serialVersionUID = 2L;
7           private List<HashFunction> hashFunctionTable; // 一个HashTable中含有多
                个哈希函数
8           private int numberOfHashFunctions; // 哈希函数数量
9           private Map<HashBucket, List<SearchableObject>> objectIndex; // 桶与包
                含可搜寻到的object列表的映射
10
11          public HashTable(int numberOfHashFunctions, int numberOfDimensions,
                double slotWidthW, Random randomNumberGenerator) {} // 构造函数
12          public HashBucket getHashBucket(SearchableObject searchableObject) {}
                // 输入object的特征为参数，返回object对应的桶
13          public void add(SearchableObject searchableObject) {} // 将object加入桶
```

```
14          public List<SearchableObject> getObjects(HashBucket hashBucket) {} //
                传入bucket返回桶中的object
15          public List<SearchableObject> getAllObjects(){} // 对于object所映射入的
                所有桶中的object进行返回
16
17          public List<HashFunction> getHashFunctions() { return this.
                hashFunctionTable; }
18
19  }
```

•SearchableObject

可搜寻的 object 实体，应该包括自身的特征列表，资源地址以及对应的 get 方法，还有计算与其他 object 的距离方法，以及比较是否与待比较 object 相等的比较函数，最后还有 object 的哈希编码。

```
1   public class SearchableObject implements Serializable{
2
3           private static final long serialVersionUID = 5L;
4           private List<Double> objectFeatures; // 特征列表
5           private URL objectUrl; // 资源地址
6
7           public SearchableObject(SerializableHistogram h) {}
8
9           public SearchableObject(List<Double> v) {}
10
11
12          public List<Double> getObjectFeatures() {} // 得到object的feature
13          public URL getObjectUrl() {} // 得到url
14
15          /**
16           * @param other
17           * @return square of euclidian distance to another object.
18           * Used for brute force similar search.
19           */
20          public double distanceTo(SearchableObject other) {} // 计算与其他object
                的距离，根据本数据集特征，采用向量的欧几里得距离平方作为距离
21
22          @Override
23          public boolean equals(java.lang.Object other) {} // 比较与传入object是
                否相同
24
25          @Override
26          public int hashCode() {}
27  }
```

7.2 perturbation 与 query 部分

•MultiProbe

MultiProbe 多探针方法, 最重要的是要实现随机生成的扰动序列，为扰动向量计分，还有生成最终扰动序列集过程中的 shift、expand 操作。

```
public class MultiProbe {

        private int numberOfHashFunctions; // hashFunction数量
        private double slotWidth; // slot宽度W
        Random randomNumberGenerator; // 随机数，用于在原index上作为随机浮动
        private List<distanceToNextSlot> randomDistancesToNextSlot,
            sortedRandomDistancesToNextSlot; // 到原slot的随机生成距离序列， 以
            及排序后结果
        private List<Perturbation> randomPerturbations; // 随机扰动

        public MultiProbe(int numberOfHashFunctions, double slotWidth, int
            numberOfPerturbations, Random randomNumberGenerator) {} // 初始化成
            员
        private void generateRandomDistancesToNextSlot() {} // 生成z_j 的联合分
            布
        private int getIndexForDistanceToPreviousSlot(int
            indexForDistanceToNextSlot) {}
        private double getPerturbationScore(List<Integer> perturbedVector) {}
            // 传入参数perturbation向量，计算分数score
        private boolean isValidPerturbation(Perturbation candidatePerturbation)
             {} // 判断是否为有效扰动
        private int getMaximumPerturbationComponent(Perturbation
            inputPerturbation) {}
        private Perturbation shiftPerturbation(Perturbation inputPerturbation)
            {} // 生成过程中shift操作
        private Perturbation expandPerturbation(Perturbation inputPerturbation)
             {} // 生成过程中expand操作
        private void generateRandomPerturbations(int numberOfPerturbations) {}
            // 随机生成扰动序列
        private class distanceToNextSlot implements Comparable<
            distanceToNextSlot> {} // distanceToNextSlot类
}
```

•Perturbation

Perturbation 的最重要的数据成员就是扰动向量，是 integer 的列表形式，然后还有该扰动向量的 score, 并提供分数的比较功能。

```
public class Perturbation implements Comparable<Perturbation> {

        private List<Integer> perturbedVector; // 扰动向量
        private double perturbationScore; // 扰动向量的Score
```

```
5
6           public Perturbation(List<Integer> perturbedVector, double
                perturbationScore) {} // 构造函数
7
8           @Override
9           public int compareTo(Perturbation other) {} // 与接收的Perturbation进行
                Score的比较
10          public List<Integer> getPerturbedVector() {}
11          public double getPerturbationScore() {}
12  }
```

•PerturbationSequences

扰动队列类中需要包含初始随机扰动序列的生成，生成扰动序列的 sequence 过程涉及的函数，如 shift、expand，添加扰动分数，判断扰动序列是否有效等等私有方法。

```
1   public class PerturbationSequences {
2       private int numberOfHashFunctions; // 哈希函数数量
3       private double slotWidth; // W
4       private int numberOfPermutations; // 扰动向量数量
5       private List<Perturbation> randomPerturbations; // 率先生成的随机扰动序列
6
7       public PerturbationSequences(int numberOfHashFunctions, double slotWidth,
            int numberOfPerturbations) {} // 需要调用generateRandomPerturbations
8       private double addPerturbationScore(double perturbedComponent) {} // 添加扰
            动分数，shift和expand步骤使用
9       private int getIndexForDistanceToPreviousSlot(int
            indexForDistanceToNextSlot) {} // index --> slot
10      private boolean isValidPerturbation(Perturbation candidatePerturbation) {}
            // 是否是有效扰动
11      private int getMaximumPerturbationComponent(Perturbation inputPerturbation)
             {} // 取得最大的perturbationComponentIndex
12      private Perturbation shiftPerturbation(Perturbation inputPerturbation) {}
            // 生成扰动序列的队列过程中的shift操作
13      private Perturbation expandPerturbation(Perturbation inputPerturbation) {}
            // 生成扰动序列的队列过程中的expand操作
14      private void generateRandomPerturbations(int numberOfPerturbations) {} //
            生成随机扰动序列
15      public List<Perturbation> getPerturbations() {}
16  }
```

7.3 实验测试

设计论文中的几个用于评价的 metrics;

• 召回率 recall, 对于查询 q, 让 $I(q)$ 为理想的邻近点向量集合，$A(q)$ 为算法所找到的最近集合（认为是 k 近邻），那么召回率计算为:

$$recall = \frac{|A(q)(q)|}{|I(q)|}$$

• 错误率 error ratio，用来衡量近似近邻搜索的质量:

$$error\ ratio = \frac{1}{|Q|K} \sum_{q \in Q} \sum_{k=1}^{K} \frac{d_{LSH_k}}{d_k^*}$$

d_{LSH_k} 代表 LSH 方法找到的第 k 近邻，d_k^* 而是样本的真实的第 k 近邻，这个指标实际衡量了 LSH 方法得到的 k 近邻的向量表达与真实的 k 近邻的向量表达之间的 distance. 采用 10 次实验，每次实验随机取 100 个 query 点进行实验，使用 10 个 HashTable，K=10 进行实验（论文中是 20）得到结果如下:

表 1: 测试结果

Category	**recall**	**error ratio**	**query times(s)**
1-Step-Wise Probing	0.917	1.033	0.074
1,2-Step-Wise Probing	0.949	1.020	0.112
1,2,3-Step-Wise Probing	0.956	1.018	0.207

可以观察到 1-Step-Wise Probing 方法就已经能够达到九成以上的召回率，1-Step 到 1,2-Step，多扩展一个距离位移召回率有较为明显增大，但是从 1,2-Step 到 1,2,3-Step 就不那么明显了，而且时间成本几乎要翻倍了，基本达到了论文中的预期效果。