山东大学

Frequency-Hiding Order-Preserving Encryption with Small Client Storage

实验复现

学院名称：网络空间安全学院（研究院）
专业名称：网络空间安全
姓名：谢钟萱
学号：201800150063

目录

1 问题简介

1.1 保序加密问题及其变种

- **保序加密问题（OPE）** 数据库中的条目在加密后以密文的形式存在，而用户希望通过对密文进行有效检索，这就要求密文在被加密的同时需要保持一定的顺序。OPE 的最低安全要求是除了密文顺序，不泄漏任何信息
- **揭序加密问题（ORE）** 在 OPE 的基础思想上，ORE 要求密文在进行比较之前不会显示出明文的顺序。在 ORE 方案中通常通过一种特殊的算法揭示密文的顺序，而不是直接比较。
- **频率隐藏的保序加密问题（FH-OPE）** OPE 和 ORE 存在的问题是可能会泄漏密文的出现次数。FH-OPE 是为了解决该问题而提出的。为了隐藏频率，方案不仅要保留密文的顺序，而且要为相同的明文随机产生不同的（不确定的）密文/顺序。这些随机密文/顺序需要通过一个状态（state）来保存，从而可以将 FH-OPE 分为**客户端状态方案**和**服务器状态方案**两种。有状态的解决方案意味着方案中包含一个动态组织密文顺序的状态，而无状态的解决方案则没有。而客户端状态、服务器状态则是指状态的存储位置。当前已经有多种基于这两种不同状态存储位置的的 FHOPE 方案被提出。

1.2 OPE 安全性定义

- **有序选择明文不可区分** (Indistinguishability under Ordered Chosen Plaintext Attacks，IND-OCPA)。IND-OCPA 规定经过保序加密后，攻击者即使获得 2 个密文也无法根据密文序关系确定明文序关系。在有状态方案中，状态中的顺序也应该看作密文的一部分。
- **统计攻击下有序选择明文不可区分** (Indistinguishability under Frequency-analyzing Ordered Chosen Plaintext Attack, IND-FAOCPA)。 [2] 中提出的 IND-FAOCPA 严格强于 IND-OCPA。当不止一个明文被（确定性）加密的时候，攻击者可以通过频率分析的方式得到密文中间的关系。IND-FAOCPA 定义了在频率分析有序选择的明文处理下的不可区分性。

2 方案综述

本文所提出的方案 [4] 是建立在之前提出的几种具有不同特征的 OPE 方案上的。为了阐明该文献中 FH-OPE 方案的思路，在此处对之前的几种方案进行简要叙述。

2.1 Kerschbaum's FH-OPE

Kerschbaum's scheme[2] 是客户端状态方案。客户端维护一个树状结构的状态，将明文映射到保序密文上。该方案基本思想是随机化密文（即将相同的明文加密成不同的密文），从而将每个明文在树中的位置随机化。当前明文所对应的密文是下一个较小明文和下一个较大明文的密文的平均值。

Kerschbaum's scheme 能够达到 IN-FAOCPA 安全性。该方案的缺点在于客户端的存储为 $O(n)$，其中 n 是明文数量。这意味着客户端和服务器存储量级相同，这对于实际的云端存储是不可行的。

2.2 mOPE 方案

mOPE 方案 [1] 是服务器状态方案。服务器维护一个二叉搜索树的状态，当密文被插入树中时由服务器生成保序编码。核心在于其保序编码生成的过程是交互式的，编码是可变的。每逢插入新数据，客

户端先和服务器端交互，通过不断地比较节点的大小获取插入路径，在客户端加密明文后随编码一并插入。

mOPE 同样能够达到 IN-FAOCPA 安全性。该方案的去缺点在于为了插入和查询密文，需要多轮通信从而确定该密文在服务器树结构中的位置；并且插入密文的时候可能引起大量树的重平衡，从而导致效率大大降低。

2.3 POPE 方案

POPE 方案 [3] 是服务器状态方案。在 mOPE 方案的基础上，该方案通过将树结构改进为 B+ 树，从而得到了效率的提升。POPE 通过服务器端的 B+ 缓冲区树状结构状态来存储密文，实现了部分保序编码。当客户端执行加密操作时，服务器只将密文插入到节点的缓冲区。一旦客户端执行查询操作，缓冲区中的密码文本就会被依次驱逐到树的叶子节点中。这一延时调整操作降低了 POPE 方案中树的重平衡次数。

POPE 依旧存在和 mOPE 相同的大量交互的缺点，同时由于 B+ 树的结构 POPE 并未实现完全有序，只实现了部分有序。

2.4 LCOPE

文献 [4] 提出的方案 LCOPE 是一种客户端-服务器状态方案，即在客户端和服务器上都存在记录顺序的存储结构。该方案提出的动机是基于以下现实情况的考虑：数据库中的某个字段数据量 N 远大于数据可能的范围 N（换言之存在大量重复数据，例如用户年龄字段），这些字段是易受频率攻击的，因此需要提供 IND-FAOCPA 安全性。然而前文所述的几种方案都存在某方面劣势从而无法应用到实际中，因此该文章提出的 LCOPE 对各方面效率进行综合考虑，综合了以上几种方案的优点提出了全新的存储树结构，具有着单轮交互、客户端存储量小、树重平衡次数少等多个优势。

以下对 LCOPE 的思想进行简述：

客户端状态

建立一个小型本地表，将排序过的明文映射到该明文的计数上。客户端的存储为 O(N)，其中 N 是不同明文的数量。和 Kerschbaum's FH-OPE（存储量为明文总数）进行对比，存储量明显下降。在客户端进行密文的插入时无需和服务器进行交互，只需要在本地表上就可以找到该密文对应的位置。

服务器状态

保持一棵类似于 B+ 树的结构，每个叶子节点都在一个特定的区间 $(lower, upper]$ 内存储一些密文。当插入新的密文的时候，客户端的输入是密文 c 和从本地表所得到的位置 pos。本文采取的编码策略类似 2.1Kerschbuam 的方案，采用左右平均值作为新节点编码。对客户端的编码策略的叙述如下：

- 客户端从根节点开始找到该位置所在节点，使用左右邻居的编码 cd_1, cd_2 的平均数 $\lceil \frac{cd_1+cd_2}{2} \rceil$ 作为该密文的编码。
- 如果左右邻居中有空则使用 $lower/upper$ 代替。
- 如果插入时已无可用编码，则对区间中所有节点进行均匀分布。该过程称为**重编码**（recoding）。

[1]作者在文中并未明确给出该方案的名称，为了实验报告行文流畅，下文中简称其为 LCOPE 方案。

- 如果一个叶节点满了，触发树的**重平衡**（rebalance）。

按以下1给出区间为 $(0, 8]$ 的叶子节点插入示范。

插入节点位置	密文	编码	情况说明
空节点	ct_1	4	两侧为空，使用两次区间范围平均值为编码
ct_1 前	ct_2,ct_1	2,4	左侧为空，使用 $lower$ 和 ct_1 编码平均值
ct_1 后	ct_2,ct_1,ct_3	2,4,6	右侧为空，使用 $upper$ 和 ct_1 编码平均值
ct_1 前	ct_2,ct_4,ct_1,ct_3	2,3,4,6	使用 ct_2 和 ct_1 编码平均值
ct_4 前	ct_2,ct_5,ct_4,ct_1,ct_3	1,3,5,7,8	使用 ct_2 和 ct_4 编码值中无可用，进行重编码

表 1: 客户端状态树中插入节点实例

2.5 方案效率对比概览

在文献中给出了现有的几种方案的效率对比，可以看到 LCOPE 方案在提供高安全性的同时在各方面的表现都较为优秀。在本实验报告的后续章节将对下表中的效率进行实测。

OPE scheme	Security		Interaction		Client Storage		Incomparable	Server
	IND-OCPE	IND-FAOCPA	Insert	Query	Working	Persistent	Elements	Storage
K's scheme	Yes	Yes	1	1	$O(n)$	$O(n)$	0	$O(n)$
mOPE	Yes	No	$O(\log n)$	$O(\log n)$	$O(1)$	$O(1)$	0	$O(n)$
POPE	Yes	Yes	1	$O(\log n)$	$O(1)$	$O(n^{\epsilon})$	$\Omega(\frac{n^{2-\epsilon}}{m} - n)$	$O(n)$
LCOPE	Yes	Yes	1	1	$O(N)$	$O(N)$	0	$O(n)$

表 2: Comparison with Previous OPE Schemes.

3 LCOPE 服务器端编码树代码实现

为了解决 mOPE 中的重平衡问题，LCOPE 方案中提出了全新的编码树结构。该编码树代码使用 C 语言实现，在每小节中详细解释了代码位置及功能。

3.1 编码树结构

树中的节点分为内部节点和叶节点。在 m-次编码树中，每个内部节点包含 m 个孩子节点，且每个叶节点包含 m 个密文。

内部节点

该结构体的实现在 Node.h 中的 class InternalNode。内部节点的作用为存储关键词 kwd，从而对客户端输入的 pos 进行检索，其中的成员包含两个数组：vector<Node*> child 为孩子节点数组，存储每个孩子节点的指针。vector<int> child_num 为关键词数组，存储每个孩子节点中的数量。由于在树中每层节点是按照大小顺序组织的，当收到客户端的 pos 时，服务器只需要逐个对孩子节点中的数量进行对比，就能够找到应该插入的孩子节点位置，并向下递归到叶节点。

vector<int> child_num	kwd_1	kwd_2	...	kwd_m
vector<Node*> child	$child_1$	$child_2$	...	$child_m$

内部节点

表 3: 内部节点结构示意

叶节点

该结构体的实现在 Node.h 中的 class LeafNode。叶节点中存储的是客户端输入密文和服务器对它的编码，其中包含 vector<string> cipher（密文数组）和 vector<long long> encoding（编码数组），根据下标一一对应。除此之外叶节点还需要存储 lower 和 upper，从而界定区间。另外叶节点还存储了指向其左右兄弟的指针，从而方便重平衡操作。

vector<string> cipher	ct_1	ct_2	...	ct_m
vector<long long> encoding	cd_1	cd_2	...	cd_m
$lbro$	$lower$	$upper$	$rbro$	

叶节点

表 4: 叶节点结构示意

3.2 插入操作

根据 2.4 节中对于服务器状态的叙述，插入操作的流程图可以被总结为图1。

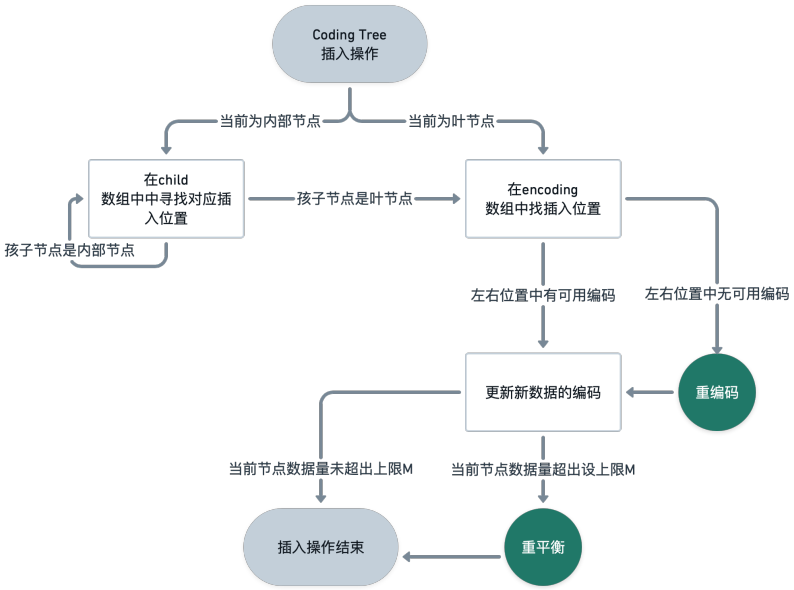


图 1: 插入操作流程图

可以看到插入操作涉及到所在叶节点寻找、编码、正常插入、重编码、重平衡等几个模块，以下对每个模块进行分别叙述。

所在叶节点寻找

所在叶节点寻找模块的输入为 int pos 和 string cipher。

对于内部节点来说，首先要找到，每个节点包含了各个子节点中所存储的数量。以图2作为示例，如果客户端输入的 pos 为 5，那么服务器从根节点开始，遍历关键词数组 child_num。此时 $pos = 5$，比 root 的第一个孩子节点的大小 4 更大，就将 pos 减去这个节点的大小 4，继续向下遍历。那么对于第二个节点来说，此时的 $pos = 1$，小于第二个节点的大小 2。此时将该孩子节点的大小增大 1，并向下一层递归，直到叶节点停止。

```
for (int  i = 0; i < this->child.size(); i++) {
  if (pos > this->child_num.at(i)) {
    pos = pos - this->child_num.at(i);}
  else {
    this->child_num.at(i)++;
    return this->child.at(i)->insert(pos, cipher);}
}
```

Listing 1: 寻找插入孩子节点位置

另一种情况是该节点的每个现有的孩子节点都已经满了，此时直接在最后一个孩子节点进行插入。

```
this->child_num.back() = this->child_num.back()++;
return this->child.back()->insert(pos, cipher);
```

注意到这个过程中可能会出现该节点的叶节点数目超过设定上限的情况，该文章采取的策略是先插入，在重编码和重平衡模块再对这些节点进行统一调整的策略。该函数的实现在 InternalNode 类中的成员函数 insert(int pos, string cipher) 。该函数为递归函数，由于 InternalNode 和 LeafNode 中都实现了 insert，因此终止情况存在于叶节点中，内部节点中仅做寻找路径操作。

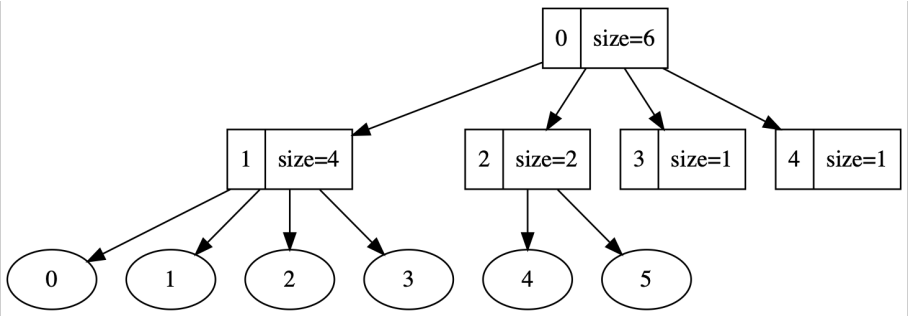


图 2: 所在叶节点寻找实例[1]

[1]此处给出的树初始状态不代表系统中存在该真实状态，仅做说明使用。

编码

该函数的实现在 LeafNode 类中的成员函数 Encode(int pos)。

编码模块的输入为 int pos，返回值为对该位置的编码。

当寻找到叶节点的时候，此时要在叶节点中对新数据进行插入。根据用户传入的 pos，服务器要对其进行编码。根据 3.4 节中叙述的编码策略，新节点的编码是其所在位置左右两侧节点的平均值。首先通过叶节点中的成员 lower 和 upper 分别作为左侧编码 left 和右侧编码 right，根据 pos 的大小找到新数据在编码数组中的位置，如果左/右有邻居，就将 left/right 修改为左/右邻居的编码值。此时判断左侧编码和右侧编码中是否还有位置，如果这两个值是否只相差 1 则触发重编码操作；否则就取两侧编码平均值上取整并进行插入。

```
long long left = (pos > 0 ? this->encoding.at(pos - 1):this->lower);
long long right =(pos < this->encoding.size() - 1? this->encoding.at(pos + 1):this->upper);
if (floor(right - left) < 2) {
Recode();//此处为缩略，代表进行重编码操作}
else{
    long long frag = (right - left)<<1;
    unsigned long long re = right - frag;
    this->encoding.at(pos) = re;}
```

重编码

该函数的实现在 Node.h 中的 Recode(vector<LeafNode*> node_list)。

重编码模块对输入为 vector<LeafNode*> node_list，即一个包含了叶节点的列表。该函数是一个递归函数。重编码的含义是在一个节点中存在间隔 =1 的两个相邻编码，导致节点未满而无法插入时，将所有编码进行均匀分布调整（注意：重编码只涉及到叶节点中编码的调整，与树中节点的布局无关）。

在任何触发重编码操作的地方，都需要该调用位置准备一个 node_list 并放入需要重平衡的节点。在重编码的递归过程中保证了 node_list 总是按序、连续的。

首先通过 node_list 中第一/最后一个节点的 lower/upper 确定其中叶节点的总左右区间 right_bound 和 left_bound，并且根据每个节点的 cipher 数组大小得到总数据量 total_cipher_num。

递归的终止条件为左右区间中可以给出所有数据的编码，即 right_bound - left_bound > total_cipher_num。此时只要顺序将所有编码修改为在此区间内均匀分布即可。由于递归的过程中 node_list 可能不止一个节点，需要将每个节点的上下界做相应修改。

```
long long frag = floor((right_bound - left_bound ) / total_cipher_num);
for (size_t i = 0; i < node_list.size(); i++) {
  node_list.at(i)->lower = cd;
  for (int j = 0; j < node_list.at(i)->encoding.size(); j++) {
    node_list.at(i)->encoding.at(j) = cd;
    update.insert(make_pair(node_list.at(i)->cipher.at(j), cd));
    cd = cd + frag;}
  node_list.at(i)->upper = cd;
}
```

调整编码均匀分布

如果左右区间中容纳不下所有数据，那么此时通过 node_list 中首节点的 left_bro 指针和尾节点的 right_bro 指针找到整个 node_list 的左右兄弟，并令其分别成为 node_list 的新首/尾。由于根据编码树的策略，总是在前一个节点满后才会对下一个节点进行插入，因此如果此时没有右兄弟，那么就说明叶节点层已经全满，则对整棵树进行扩容。

重平衡

该函数的实现分别在 InternalNode 和 LeafNode 中的 rebalance()。

重平衡的意义是当一个节点中的数据量超过设定的上限 M 时，此时需要对整棵树的布局进行调整。

首先根据当前节点类型创建一个新的内部/叶节点，并将自己的孩子节点中较大的一半分给新节点。并调整分出那一半孩子节点的父节点指针。

```
InternalNode* new_node = new InternalNode();
  int middle = floor(this->child.size()*0.5);
  while (middle > 0) {
    new_node->child.insert(new_node->child.begin(), this->child.at(this->child.size()-1));
    new_node->child_num.insert(new_node->child_num.begin(), this->child_num.at(this->child_num.size()-1)
        );
    this->child.pop_back();
    this->child_num.pop_back();
    middle--;
  }
```

以内部节点为例的创建新节点操作

如果该节点为根节点，那么就需要对根节点进行调整。创建一个内部节点作为新的根节点，并将该节点和新创建的节点作为根节点的孩子插入。如果并非根节点，那么就对双亲节点中的 child_num 进行调整，并且插入新节点。

如果该节点为叶节点，那么还需要对原节点和新节点的其他域 (lower, upper, right_bro, left_bro) 进行调整。对于新节点区间的调整策略是使用新节点中的第一个编码作为左边界、原节点的 upper 作为右边界。与之对应的，原节点使用 lower 和原节点中最后一个编码作为边界。

插入叶节点

该函数的实现在 LeafNode 类中的成员函数 insert(int pos, string cipher) 。

对于叶节点来说，首先根据传入 pos 直接将 ct_{new} 插入到 cipher 数组的对应位置，并且在 encoding 数组的对应位置给出编码。此时判断叶节点中的数量是否超过设定的 M 值，如果超过则触发重平衡操作。

3.3 搜索/遍历操作

对于内部节点来说，搜索操作和 3.2.1 节中所述类似，仅将其中的修改操作去掉。对于叶节点来说，当 pos 传入时直接在 cipher 数组中进行下标寻找即可。由于客户端保持了本地表，因此传入的 pos 总是符合范围内的。该函数的实现在 InternalNode 和 LeafNode 类中的成员函数 search(int pos)。

遍历操作实际上就是 B+ 树的遍历操作，从根节点开始递归每个孩子节点，当到达叶节点的时候顺序输出叶节点中的每个编码。该函数的实现在 InternalNode 和 LeafNode 类中的成员函数 traverse()。

4 代码运行插入操作可视化结果

相对搜索/遍历操作来说，插入操作涉及的流程较复杂、代码较多，因此本节以一棵 4 节点、初始区间 $[0, 8]$、每次扩容 2^3 大小的编码树来进行具体说明。为了使得理解更直观，本节采用 log 文件和树可视化结果结合进行说明。本测试只关注插入操作，所采取的插入 pos 为随机生成，能较好的模拟各种情况。

下图3是对编码树可视化的一个结果示例，其中上层为内部节点，其中的 size 代表孩子节点的数量。下层为叶节点，其中 size 代表该节点中存放的密文的数量，size 下方则是节点内存储密文的部分信息。方括号 [%d] 中的数字代表的是该节点为第 i 个插入的节点，括号后的数字则是服务器对该密文给出的编码。

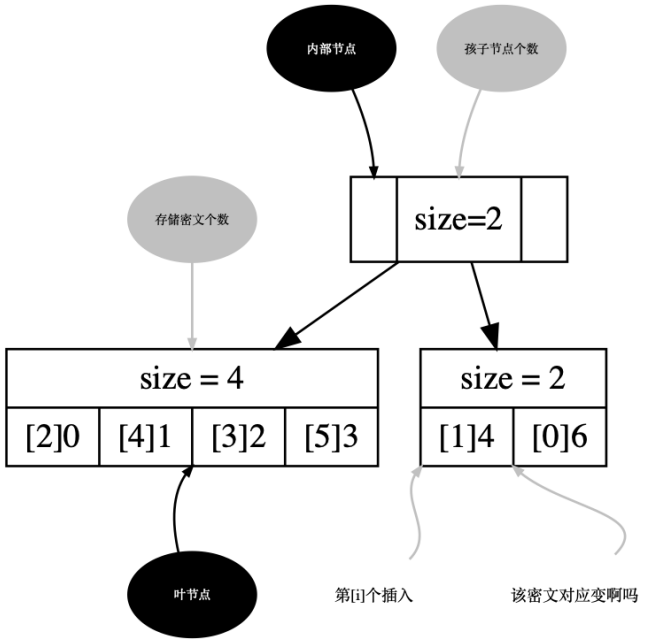


图 3: 编码树可视化基本说明

4.1 初始化/中位数编码

由于客户端存储了关于当前每个不同明文对应多少个计数，因此在插入第一个节点时，本地表中为空，插入 pos 一定为 0。此时的根节点为叶节点，因此进行直接插入。log 文件的输出表示了该插入节点为第 0 个节点，插入位置 $pos = 0$，$Leafpos$ 代表路径寻找到叶节点时，新节点在该叶节点中的位置。在该此时的左右邻居都为空，因此 $[left, right]$ 直接采用左右区间 0、8。而 now 代表着当前节点插入的位置，此时该位置的编码为 4.

Node 1	Pos = 0,Node Pos = 0 [left,now,right] = 0 4 8,Encoding: 4

表 5: 插入第 1 个节点

size = 1
[0]4

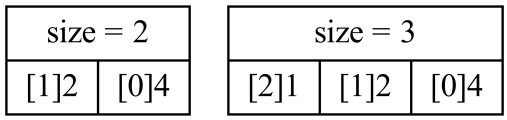
图 4: 插入第 1 个节点的编码树

[2]该节点是第几个插入的节点在编码树结构中并没有实际存储，此处为了方便说明而附带该部分信息

接下来在插入 2、3 个节点的时候，都是简单的取中位数的操作。第 2 个插入节点的位置为 0，因此选取 $lower = 0$ 和第一个节点的编码 4 做平均值，取得编码 2。同样的有第 3 个节点的编码为 0 和 2 的平均值 1。

Node 2	Pos = 0,Leaf Pos = 0 [left,now,right] = 0 2 4,Encoding: 2
Node 3	Pos = 0,Leaf Pos = 0 [left,now,right] = 0 1 2,Encoding: 1

表 6: 插入第 2、3 个节点



(a) 第 2 个节点　　(b) 第 3 个节点

图 5: 插入第 2、3 个节点的编码树

4.2 重编码/重平衡操作

叶节点内有足够的位置容纳新插入密文

在插入第 4 个节点时，通过 log 文件可以看到插入的位置为 1，从上一个节点的状态5(b)可以看出此时左右的编码分别为 1、2，没有足够的位置来放置新密文，此时进入重平衡的过程。表7中的第 2 行显示此时该节点的范围是 [0,8]，节点内共有 4 个密文，即节点内有足够的范围来容纳。第 4 行的 cd 代表起始节点，frag 代表均匀分布的间隔，可以看到该节点分配后 4 个密文的编码分别变为 0 2 4 6。因此第四个节点的编码为此时该节点中 $pos = 1$ 的编码，即 2。

Node 4	Pos = 0
	Scale = [0 8],CipherNums = 4 -> Enough codes. cd = 0, frag = 2 Node0:0 2 4 6
	Leaf Pos = 0,Encoding: 0

表 7: 插入第 4 个节点

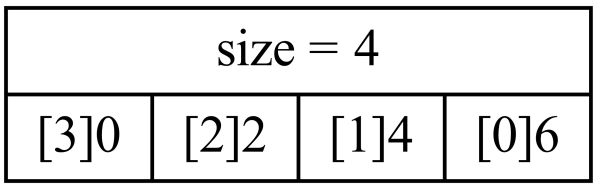


图 6: 插入第 4 个节点的编码树

重平衡操作/创建新节点

在插入第 5 个密文的时候 $pos = 1$，根据6中左右编码分别为 0、2，因此首先给出该密文的编码为 1。但由于该编码树的节点上限为 4，根节点没有足够的空间容纳新密文，因此此时需要进入重平衡过程。由于此时原节点为根节点，因此需要新建一个 InternalNode 作为新的根节点，并且根据分割策略将原节点中的 5 个密文平衡到两个孩子节点中去，于是形成了如7中所示的编码树结构。

Node 5	Pos = 1,Leaf Pos = 1 [left,now,right] = 0 1 2,Encoding: 0
	******Enter rebalance: Is root: ******Out rebalance.

表 8: 插入第 5 个节点

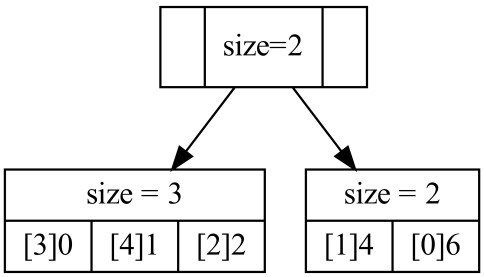


图 7: 插入第 5 个节点的编码树

叶节点内没有足够的位置容纳新插入密文

接下来在插入 6 个节点的时候，插入位置为 2，此时左右编码分别为 0、2，没有可用编码，于是进入重编码过程。可以看到 log 文件中显示，左侧孩子的范围为 [0,4)，而密文数为 4 个。由于定义中可用编码需要严格大于密文数量，因此此时认为没有足够的编码可用，于是将右侧孩子节点也放入 Node_list 中，递归进入下一次重编码。此时区间和密文数分别变为 [0,8)、6，足够将编码均匀分布。此时起始点为 0，间隔为 1，从左到右的密文编码为 0 1 2 3 4 5。可以看到这正是文章中所说的，重编码只涉及到节点中 $[lower, upper)$ 区间和密文编码的调整，不涉及到节点的重新布局。

Node 6	Pos = 2,Leaf Pos = 1 **enter recode
	Scale = (0 4],CipherNums = 4 -> No enough codes.
	**enter recode Scale = (0 8],CipherNums = 6 -> Enough codes. cd = 0, frag = 1 Node0:0 1 2 3 Node1:4 5 **out recode
	**out recode
	Pos = 2,[left,now,right] = Encoding: 2

表 9: 插入第 6 个节点

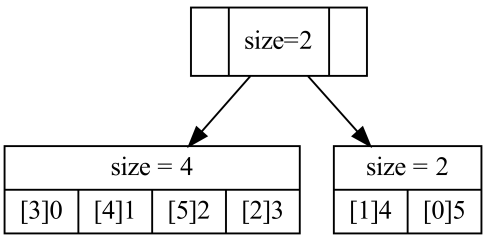


图 8: 插入第 6 个节点的编码树

4.3 结果展示

以上就是几个基本操作的可视化展示，具体结果可以在 logfh.txt 和每个插入节点的 dot 文件中进行查看。为了简要展示编码树较大时的状态，在图9中展示了插入 35 个节点的编码树。以上可视化文件的代码在 lcope_test.c 中。

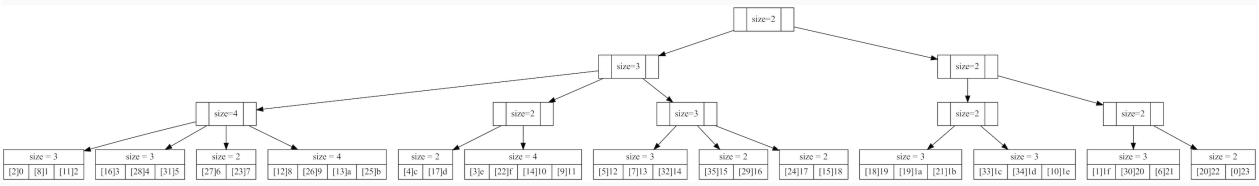


图 9: 插入第 35 个节点的编码树

5 MYSQL 上代码实际操作

5.1 创建并调用 MYSQL 接口

创建 MYSQL 接口

正常向 mysql 数据库进行插入和搜索时，使用的为 INSERT 和 SELECT 操作。这两个操作在该系统中需要被替换为以上 FHOPE 中所定义的插入和搜索操作（3.2、3.3 节）。在本小节中介绍如何将上述 C 代码打包为 MYSQL 的拓展接口 UDF（用户自定义函数），该节中所有函数的实现都在 lcope.cpp 文件中。

对于 MYSQL 提供的结构体 UDF_ARGS，定义 static char * getba(UDF_ARGS *const args, int i, double&len) 来获取 SQL 语句返回值的第 i 个元素。

```
static char *getba(UDF_ARGS *const args, int i, double&len)
{
    len = args->lengths[i];
    return args->args[i];
}
```

对于插入操作，定义 UDF 接口 FHInsert。在初始化上述编码树的根节点（初始化时根节点类型为叶节点）后，该函数从 SQL 语句中读取插入的位置 pos 和对应的密文 $cipher$，并调用叶节点方法 insert 进行插入，即返回 FHOPE 给出的编码。

```
long long FHInsert(UDF_INIT *initid, UDF_ARGS *args, char *is_null, char *error)
{
    int pos=*(int*)(args->args[0]);
    double keyLen;
    char *const keyBytes = getba(args, 1, keyLen);
    const std::string cipher = std::string(keyBytes, keyLen);
    long long start_update = -1;
    long long end_update = -1;
    update.clear();
    long long re = root->insert(pos, cipher);
    return re;
}
```

对于搜索操作，读取搜索位置 pos 后，调用根节点 search 方法向下进行搜索。

```
long long
FHSearch(UDF_INIT *const initid, UDF_ARGS *const args,char *const result, unsigned long *const length,
    char *const is_null, char *const error)
{
    int pos=*(int*)(args->args[0]);
    if(pos == -1) return 0;
    return root->search(pos);
}
```

调用如下命令将该 lcope.cpp 编译为 lcope.so 共享库文件，并放入 mysql 的插件目录中。

```
g++ -c -o ope.o -std=c++11 -I </path/to/your/mysql/include/lib> -fPIC -Wall lcope.c;
g++ -shared -o lcope.so lcope.o;
sudo mv obj/ope.so /usr/lib/mysql/plugin;
```

下图为创建 UDF PInsert 成功的实验截图，并且尝试调用其中的接口 PInsert 和 PSplit，均能够成功。

```
mysql> create function PInsert RETURNS INTEGER SONAME 'pope.so';
Query OK, 0 rows affected (0.02 sec)

mysql> select PInsert('2');
+--------------+
| PInsert('2') |
+--------------+
|            0 |
+--------------+
1 row in set (0.00 sec)

mysql> select PInsert('3');
+--------------+
| PInsert('3') |
+--------------+
|            0 |
+--------------+
1 row in set (0.00 sec)

mysql> select PSplit('2','4');
+-----------------+
| PSplit('2','4') |
+-----------------+
|               0 |
+-----------------+
1 row in set (0.00 sec)
```

图 10: 创建 UDF PInsert 并使用其中接口

调用 MYSQL 接口

在程序中使用 python 代码进行服务器操作，并利用 pymysql 库在 mysql 数据库上进行操作。首先连接到指定数据库，然后创建共享库中所定义的 UDF。以下是部分代码节选，同理对 FHSearch、FHUpdate 等其他函数做类似操作。

```
db = pymysql.connect(host="", user="",database="", password="", database="", password="", ssl={'ssl':{}})#连接到指定数据库
cursor = db.cursor() #创建数据库指针
sql = """drop function if exists FHInsert;"""#创建sql语句，如果存在FHInsert则抛弃
cursor.execute(sql)#执行sql语句
sql = """create function FHInsert RETURNS INTEGER SONAME 'lcope.so';""" #创建sql语句，调用lcope.so共享库中创建FHInsert
cursor.execute(sql)#执行sql语句
```

特别的，根据之前定义的 FHInsert 函数，其返回值为编码树所给出的编码，因此此时需要将该密文按该编码进行插入。为该功能创建 UDF pro_insert，以下是实现该功能的 SQL 语句

```
create procedure pro_insert(IN pos int, IN ct varchar(512))//创建pro_insert，参数为整数pos和512位长密文ct
BEGIN
DECLARE i BIGINT default 0;  #声明类型为BIGINT的变量i，默认值为0
SET i = (FHInsert(pos,ct));   #将FHInsert(pos,ct)给出的关于ct的编码存储在i中
insert into example values (i,ct); #向表example插入记录(i,ct)
if i = 0 then #如果使用到了编码0
    update example set encoding = FHUpdate(ciphertext) where (encoding >= FHStart() and encoding < FHEnd()) or (encoding = 0);
end if;
END
```

Listing 2: pro_insert

```
mysql> call pro_insert(1,'Cipher1');
Query OK, 1 row affected (0.01 sec)

mysql> call pro_insert(2,'Cipher2');
Query OK, 1 row affected (0.01 sec)

mysql> call pro_insert(3,'Cipher3');
Query OK, 1 row affected (0.00 sec)

mysql> call pro_insert(1,'Cipher4');
Query OK, 1 row affected (0.00 sec)

mysql> call pro_insert(2,'Cipher5');
Query OK, 1 row affected (0.00 sec)

mysql> call pro_insert(3,'Cipher6');
Query OK, 1 row affected (0.00 sec)

mysql> select * from example;
+-----------------+------------+
| encoding        | ciphertext |
+-----------------+------------+
| 316659348799488 | Cipher1    |
| 334251534843904 | Cipher2    |
| 343047627866112 | Cipher3    |
| 299067162755072 | Cipher4    |
| 307863255777280 | Cipher5    |
| 312261302288384 | Cipher6    |
+-----------------+------------+
```

图 11: UDF pro_insert 功能测试

在 mysql 中对 pro_insert 进行简单的测试，插入六条记录，得到的结果如图11所示。可以看到用户只要简单调用该接口，传入该密文的 pos 和密文本身，pro_insert 就会通过调用 FHInsert 来给出编码。

5.2 客户端状态

按照前文中的方案叙述，在客户端中需要保存一个较小的字典结构，Key 为所有不同的密文，对于 Value 为该密文的数量。因此需要定义一个客户端结构，在进行插入操作的时候对当前的客户端状态进行修改。该结构的实现在/client/clientSide/client.py 文件中的结构体 OpeClient。

客户端计数器更新

OpeClient 类中有成员列表 sortedkey（保存不同关键字，该列表按关键字大小有序排列）和列表 cnt（关键字对应计数），以及字典结构 search_map。在每次插入时需要调用 OpeClient 的成员函数 insert_udf，参数为需要插入的明文 pt。当插入明文时，首先使用 pt 在 $sortedkey$ 进行二分搜索，找到小于 pt 关键词的最大值所在位置 i。如果此时找到的 $sortedkey[i] == pt$（即关键词已存在的情况），直接在当前 $sortedkey[i]$ 所对应的 $cnt[i]$ 上加 1。其他情况下则插入当前密文，并在对应 cnt 位置插入新计数 1。

```
    def insert_udf(self, pt):
        inser_pos=bisect.bisect_left(self.sortedkey,pt)
        if len(self.sortedkey) == inser_pos or self.sortedkey[inser_pos] != pt:
            self.sortedkey.insert(inser_pos,pt)
            self.cnt.insert(inser_pos,1)
```

```
        else:
            self.cnt[inser_pos] = self.cnt[inser_pos] + 1
        total = sum(self.cnt[0:inser_pos]) + random.randint(0, self.cnt[inser_pos]-1)
        cipher = self._crypt.encode_undet(pt, self.cnt[inser_pos])
        return (total, cipher)
```

该操作较为简单，在此处直接给出客户端进行插入操作时的 $sortedkey$ 和 cnt 实例。

```
[ChristinedeMacBook-Pro:clientSide christinelin$ python]
3 client.py
************  Insert 10 distinct cipher  ************
Sortedkey:   1  2  3  4  5  6  7  8  9 10
Cnt      :   1  1  1  1  1  1  1  1  1  1

****** Insert 100 random cipher(range [1,10]) *******
Sortedkey:   1  2  3  4  5  6  7  8  9 10
Cnt      :   7 12 13 11  8  9 11  9 10 10
```

图 12: Client 端 insert_udf 函数实例

明文在编码树中对应位置寻找

此时很容易就能得到该明文在编码树中对应的位置，在 pt 前共有 $\sum_{k=1}^{i-1} cnt[k]$ 个元素，和 pt 相同的共有 $cnt[i]$ 个，那么在 $\left[\sum_{k=1}^{i-1} cnt[k]+1, \sum_{k=1}^{i} cnt[k]\right]$ 中随机选择一个位置插入，就能够保证密文的顺序。

```
    def Search(self, key):
        inser_pos=bisect.bisect_left(self.sortedkey,key)
        left_pos = sum(self.cnt[0:inser_pos])
        left_pos = left_pos -1
        return left_pos
```

以下给出寻找客户端在本地表中搜索密文的实例。

```
[ChristinedeMacBook-Pro:clientSide christinelin$ python]
3 client.py

****** Insert 100 random cipher(range [1,10]) *******
Sortedkey:   1  2  3  4  5  6  7  8  9 10
Cnt      :  12  8 12 11 11 11 11  6  9  9
Cipher  1  found at  -1
Cipher  2  found at  11
Cipher  3  found at  19
Cipher  4  found at  31
Cipher  5  found at  42
Cipher  6  found at  53
Cipher  7  found at  64
Cipher  8  found at  75
Cipher  9  found at  81
```

图 13: Client 端 search 函数实例

明文加密及密文解密

该部分的实现在/client/clientSide/cipher.py 中为了保证每个明文（可能相同）给出的密文是不相同的，在加密中采用将明文和当前明文对应计数进行级联后，填充至 16 的倍数，然后使用 AES 加密的方式（由于每次新插入当前明文对应计数都发生变化，因此不存在相同密文）。具体实现在 connect_plain_count 函数和 encode_undet 函数。

解密的过程和加密相反，首先经过 AES 解密后剥离填充部分，再对明文和计数进行分离后输出。具体实现在 decode_undet 函数。

5.3 MYSQL 上插入操作

对于数据库中的每一条记录，首先调用客户端状态中的 insert_udf 成员函数，将记录作为参数得到对应密文和插入位置 $pos, ciphertext$，然后调用 sql 语句 call pro_insert($pos, ciphertext$) 进行编码树的插入。

```
pos, ciphertext = opec.insert_udf(insertkey)
sql = "call pro_insert(%d,\"%s\");" %(pos, str(ciphertext))
cursor.execute(sql)
```

以下两图为客户端和服务器的结果。在图14(a)中可以看到给出的对每个密文给出 pos，图14(b)中则可以看到编码树所给出的 $encoding$，可以看出 pos 小所对应的 $encoding$ 也更小，符合保序加密的定义。并且对 $encoding$ 做间隔分析的话，可以看到 $encoding$ 之间大致是等距离分布的，只是由于编码树的重编码性质导致的。

```
now we are starting state/shuffle/ other
0 ZmY3YjRlZmRmODRiMDNlOTNkZGM5MDEwNjAzODNhYWI=
0 Zjk1ZjhkMWFjZjNhZDU5M2NhYTU1ZjUwM2Y3NmU3Mjc=
1 NzQ0NDYyYTYyZDVlNzAyMmJlYTdmOGZmMjJjYmRhYWM=
2 Y2ExYmRmYmZiYmRkNjAwMjBjOWI3YTI2NzdkYTRjZTY=
2 YWJhYWRkNWFkYmUxYjY2ZWNhNzdjYjM0MzYzMjY3OWI=
2 MzY5MGNhYTBiMTk0NjRhNTBiNWM1NTM5M2YwNDRjYWU=
5 NzcwNDE1MGE4YzY0YWMyY2Q4MGZhYjQyY2JlMWUzZDY=
5 YzVmM2QzZmJjOGMzY2Y2N2ZkNzE4Mzk2ZTYyYTVlM2U=
6 YjU2NDg4MTFlODM3NzEwZTYxNTMxYzIxODRkNTNhMTM=
5 MmNiZDcyOGIyMDBlYTE2NzI1NmMwZjkwZmI1ODAzODY=
0 YTg5NzljY2FiOTczNzU1ZmY3ZDM2OThkOTcyMmU5OTU=
8 YTdjZGI0NWNhOTkxNTAzM2Y2YzlkYzU5MTU4N2I5YjI=
11 OTcwYzk1NGEyMTMzMjUwMTNlYTBjMmU4NjBlZWVkOTY=
```

(a) 客户端 $pos, ciphertext$

```
mysql> select * from example;
+--------------------+------------------------------------------+
| encoding           | ciphertext                               |
+--------------------+------------------------------------------+
| 576460752303423488 | ZmY3YjRlZmRmODRiMDNlOTNkZGM5MDEwNjAzODNhYWI= |
| 288230376151711744 | Zjk1ZjhkMWFjZjNhZDU5M2NhYTU1ZjUwM2Y3NmU3Mjc= |
| 432345564227567616 | NzQ0NDYyYTYyZDVlNzAyMmJlYTdmOGZmMjJjYmRhYWM= |
| 504403158265495552 | Y2ExYmRmYmZiYmRkNjAwMjBjOWI3YTI2NzdkYTRjZTY= |
| 468374361246531584 | YWJhYWRkNWFkYmUxYjY2ZWNhNzdjYjM0MzYzMjY3OWI= |
| 450359962737049600 | MzY5MGNhYTBiMTk0NjRhNTBiNWM1NTM5M2YwNDRjYWU= |
| 540431955284459520 | NzcwNDE1MGE4YzY0YWMyY2Q4MGZhYjQyY2JlMWUzZDY= |
| 522417556774977536 | YzVmM2QzZmJjOGMzY2Y2N2ZkNzE4Mzk2ZTYyYTVlM2U= |
| 531424756029718528 | YjU2NDg4MTFlODM3NzEwZTYxNTMxYzIxODRkNTNhMTM= |
| 513410357520236544 | MmNiZDcyOGIyMDBlYTE2NzI1NmMwZjkwZmI1ODAzODY= |
| 144115188075855872 | YTg5NzljY2FiOTczNzU1ZmY3ZDM2OThkOTcyMmU5OTU= |
| 526921156402348032 | YTdjZGI0NWNhOTkxNTAzM2Y2YzlkYzU5MTU4N2I5YjI= |
| 558446353793941504 | OTcwYzk1NGEyMTMzMjUwMTNlYTBjMmU4NjBlZWVkOTY= |
+--------------------+------------------------------------------+
```

(b) 服务器 $encoding, ciphertext$

图 14: 插入操作实例

5.4 MYSQL 上搜索操作

编码树支持范围搜索，即在云端搜索并同时返回位置 [i,j] 间的所有编码。在无编码方案的正常 SQL 搜索语句中使用 where 后加布尔条件的语句，而在本方案中只需要简单的调用 FHSearch(pos) 就能够得到对应 $encoding$，并使用该 $encoding$ 进行布尔条件设置。

```
sql = "select ciphertext from example where encoding > FHSearch(%d) and encoding < FHSearch(%d);" %(opec
    .Search(left), opec.Search_right(right))
```

以下给出范围搜索的实例，左右两张图分别是有序插入和无序插入时的数据库状态，可以看到在给定范围时会返回所有符合要求的密文。

```
mysql> select * from example;
+----------------+------------+
| encoding       | ciphertext |
+----------------+------------+
|  9895604649984 | Cipher 1   |
| 10445360463872 | Cipher 2   |
| 10720238370816 | Cipher 3   |
| 10857677324288 | Cipher 4   |
| 10926396801024 | Cipher 5   |
| 10960756539392 | Cipher 6   |
| 10977936408576 | Cipher 7   |
| 10986526343168 | Cipher 8   |
| 10990821310464 | Cipher 9   |
| 10992968794112 | Cipher 10  |
+----------------+------------+
10 rows in set (0.00 sec)

mysql> select ciphertext from example where encoding
> FHSearch(3) and encoding < FHSearch(8);
+------------+
| ciphertext |
+------------+
| Cipher 4   |
| Cipher 5   |
| Cipher 6   |
| Cipher 7   |
+------------+
4 rows in set (0.00 sec)
```

```
mysql> select * from example;
+----------------+------------+
| encoding       | ciphertext |
+----------------+------------+
| 10582799417344 | Cipher 1   |
|  9345848836096 | Cipher 2   |
| 10892037062656 | Cipher 3   |
| 10982231375872 | Cipher 4   |
| 10980083892224 | Cipher 5   |
| 10874857193472 | Cipher 6   |
| 10909216931840 | Cipher 7   |
|  9620726743040 | Cipher 8   |
| 10900626997248 | Cipher 9   |
| 10896332029952 | Cipher 10  |
+----------------+------------+
10 rows in set (0.00 sec)

mysql> select ciphertext from example where encoding
> FHSearch(6) and encoding < FHSearch(10);
+------------+
| ciphertext |
+------------+
| Cipher 3   |
| Cipher 6   |
+------------+
2 rows in set (0.00 sec)
```

图 15: Range Search 结果

6 效率测试及对比

6.1 数据集准备及处理

由于文章中的方案综合考虑了客户端存储量、服务器存储量、通信交互轮数和查询复杂度等多个方面，因此在实验模拟中使用不同的数据集来模拟在不同数据规模和数据分布下方案的总体效率。总共使用三个数据集来进行实验，包括两个人工合成分布和一个真实数据集。

数据集设置

两个合成分布分别是均匀分布（表示为 Distribution 1）和正态分布（表示为 Distribution 1）。给定数据集总大小 n，Distribution 1 通过改变范围来构建了一个包含 N 个不同明文的数据集。同样地可以通过改变标准差来构建数据集 Distribution 2，使得数据集中不同明文量/总数据集大小为 N/n。在 n 大小的选择上采取 $n = 10^5$ 和 10^6 两种设置，而 N/n 上则采取 $0.00001, 0.0001, 0.001, 0.01, 0.1, 0.5$ 六种设置。

真实数据集基于加利福尼亚州公开雇员薪水数据集，数据集大小为 240530，选用其中的 “Job Title” 和 “Other Pay” 字段。对其中不同明文数量进行计数，得到下表所示的结果。

Field	Distinct Plaintext	Dataset Size	N/n
Job Title	3923	240530	0.016
Other Pay	97297	240530	0.405

```
Job_title: n = 240530,N = 3923,N/n =0.016309815823390014
Other_pay: n = 240530,N = 97297,N/n =0.4045108718247204
```

图 16: Distribute of ”Job Title” and ”Other Pay” filed

数据集处理

当对效率进行测试的时候，输入顺序对方案效率可能

6.2 存储量

在本实验的对比中使用客户端方案 Kerschbaum’s scheme（以下简称 K’s scheme）和以 [2] 方案进行压缩的 Kerschbaum’s scheme（以下简称 Compress K）进行对比。在对比中从插入数据数量 n 和不同数据占比 N/n 两个维度进行考虑。首先对真实数据集 Job Title 和 Other pay 进行测试，经过实验后得到了如表10的数据。将该数据进行可视化处理得到的结果如图17和18所示。

Dataset	paper’s scheme		K’s scheme		compressed K	
Size	Job Title	Other Pay	Job Title	Other Pay	Job Title	Other Pay
100	0.0007	0.0009	0.0011	0.0011	0.0014	0.0014
1000	0.0015	0.0062	0.0114	0.0114	0.0063	0.0063
10000	0.0091	0.1139	0.1144	0.1144	0.0563	0.2066
100000	0.0441	0.8475	1.1444	1.1444	0.4792	0.9147

表 10: Client Storage Cost(Mbyte)

数据集处理

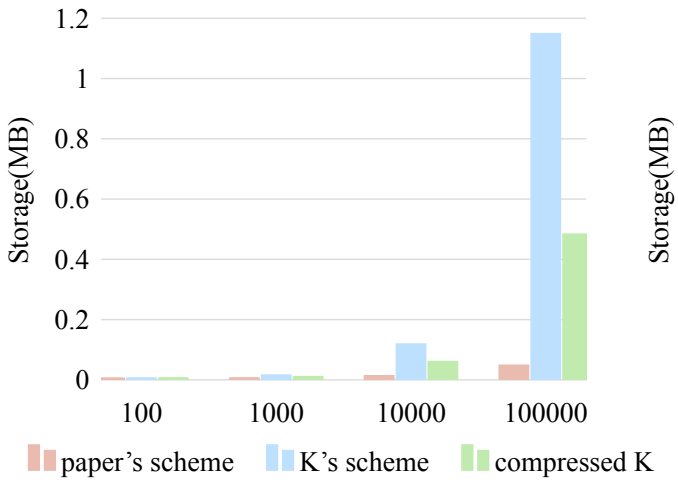


图 17: Storage of Job Title

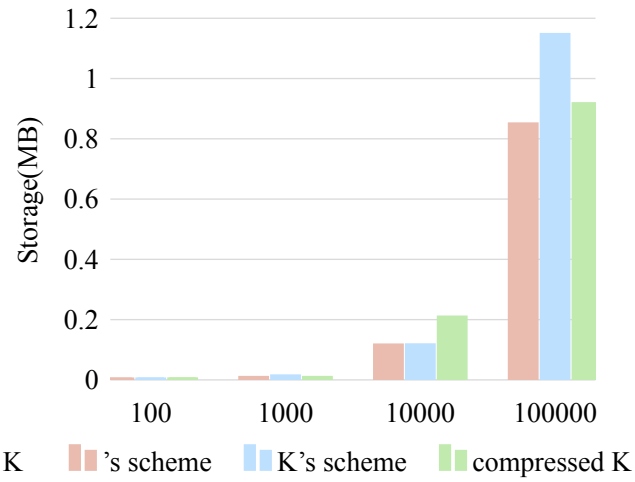


图 18: Storage of Other pay

对于 K’s scheme 和 Compress K 来说，由于方案中要求存储每条数据对应的编码，存储量和数据集的大小成正比，而与 N/n 无关。同时可以看到，当数据集较大的时候，这两个方案的存储量增长在实际中是很难应用的。该文章所述的方案优势在于当 N/n 较小时仍能保持相对较好的性质。根据数据集

处理中的描述，Job Title 的 $N/n = 0.16$，也就是数据集中只有 $0.16N$ 条不同彼此不同的数据。此时从图17中可以看到，在从 100∼100000 的范围内，本文方案的客户端存储量都有着较大优势。而对于 Other Pay 数据集的 $N/n = 0.45$，即数据集中有将近一半的不同数据，此时本文方案和其他方案相比也有相对的降低。

由于在真实数据插入中，数据集很有可能存在多条数据重复，因此该文章的改进方向还可以进一步用生成数据所模拟的真实情况进行评估。在 Distribution 1 和 Distribution 2 中分别使用的均匀分布和正态分布很大程度上能反应大部分真实数据集的情况。对以上数据进行评估，得到的结果如下图所示。可以从图中看到对于三种方案都对于不同分布的改变不敏感，而在两种分布、不同 N/n 设置下该文章的方案都有着相当的优势，能够在实际中运用。

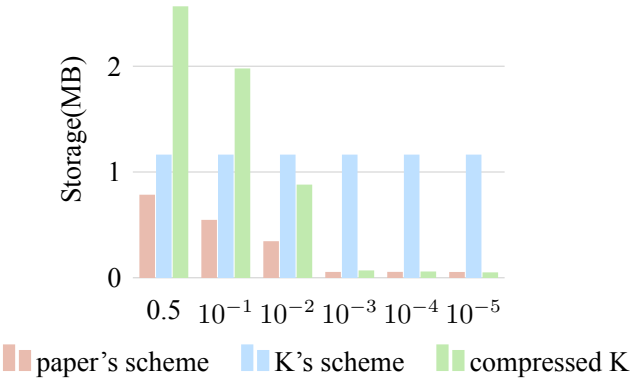


图 19: Storage of Distribution 1(n = $1e5$)

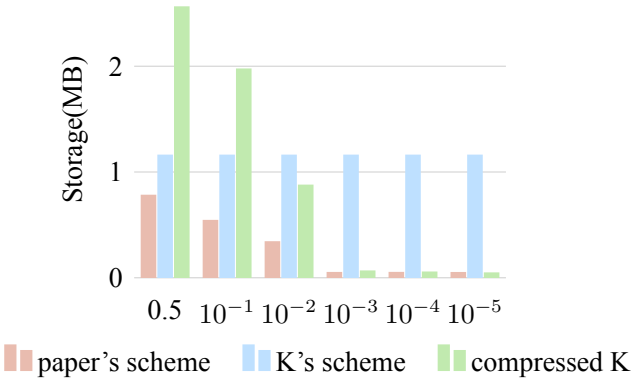


图 21: Storage of Distribution 2(n = $1e5$)

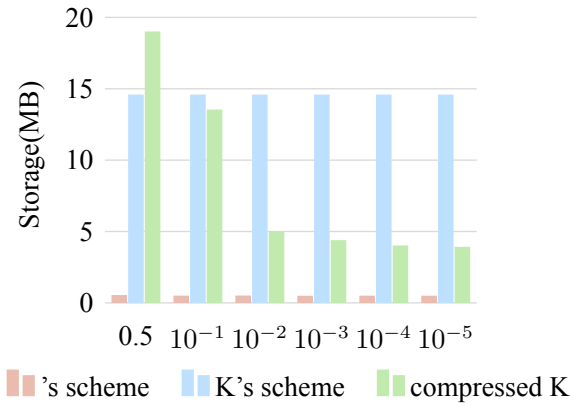


图 20: Storage of Distribution 1(n = $1e6$)

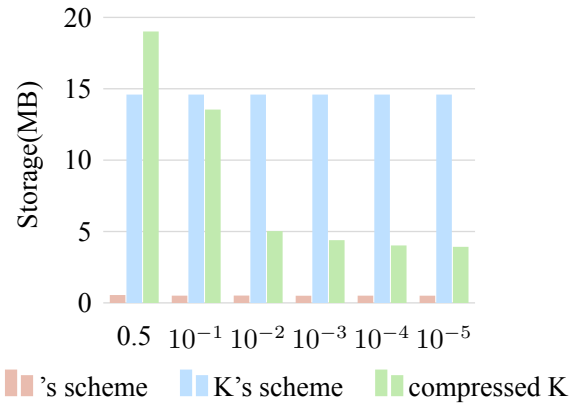


图 22: Storage of Distribution 2(n = $1e6$)

6.3 编码更新频率

由于该文章的方案涉及到客户端和服务器两方面的更新，因此同时使用 mOPE（服务器方案）的插入树结构次数和 K's scheme（客户端方案）的插入明文量同时进行比较。通过对三个方案运行如下设置的参数，所得到的结果下表11所示，其可视化结果为图23∼26。

表 11: Recoding Frequency on Real Dataset(10^6 times)

Case	LCOPE		K's scheme		mOPE	
	Job Title	Other Pay	Job Title	Other Pay	Job Title	Other Pay
$-best$	0	0	0	0	5.1	5.2
$-worst$	1.1	1.8	38.2	179.8	7.3	7.7
$-natural$	0	0	0	0	4.9	5.3

从下图中可以看到，无论是在 Best Case 还是 Worst Case 下 mOPE 方案的编码更新量都和明文数量成正比，因此对比其余两个方案啊，mOPE 方案的编码更新量是其最大劣势。

而在 K'scheme 和该文章提出方案的对比中，在 Best Case 下两种方案都不涉及到编码树的重构，因此都几乎没有任何编码更新。然而在 Worst Case 中，可以看到 K'scheme 的编码更新量有了显著增加，甚至在一些情况下超过了 mOPE。这是由于当最差情况的时候数据集中的每条数据呈现顺序排列，也就是每一条新的数据都是紧贴上一条数据进行插入，这样的情况下树出现极端不平衡，因此涉及到大量的重平衡。而在该文章提出的方案中则增加了重编码操作，这使得在很多情况下只需要局部调整编码，而不涉及到树结构的重新平衡。因此可以看到在 Worst Case 中该文章所提出的方案要优于剩余两种方案。

另外在 Worst Case 下对于 Distribution 1 和 2 的对比中，可以看到该文章方案在均匀分布下表现出类似于线性增加的趋势，并且在数据量过大的时候逼近于 K'scheme 的编码更新量。这是因为在均匀分布下大量相同的数据依次进入树结构中，经常会指向同一个节点，从而导致对相同节点的反复插入，重复消耗该节点编码，因此更容易进入重平衡。而在正态分布下，由于每个不同数据的数据量存在差异，相较于均匀分布时所寻找到的节点更为分散，因此整体表现更优。考虑到在现实世界中正态分布为更常见的数据集，因此该方案有着极强的实用价值。

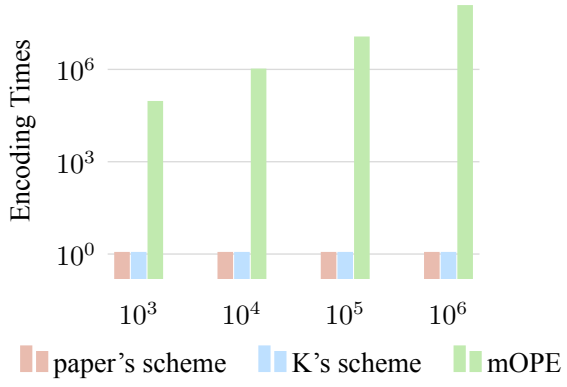


图 23: Best Case(Distribution 1)

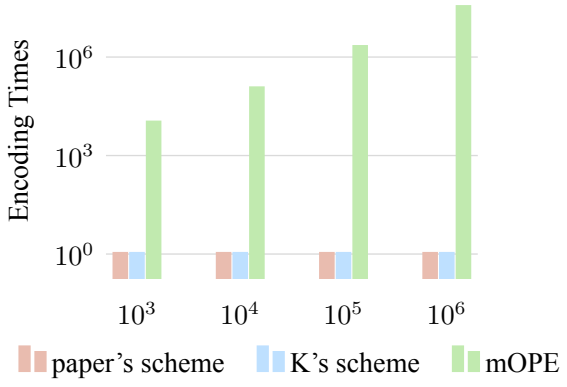


图 24: Best Case(Distribution 2)

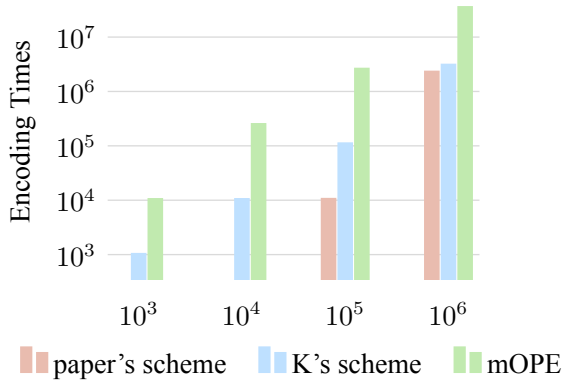


图 25: Worst Case(Distribution 1)

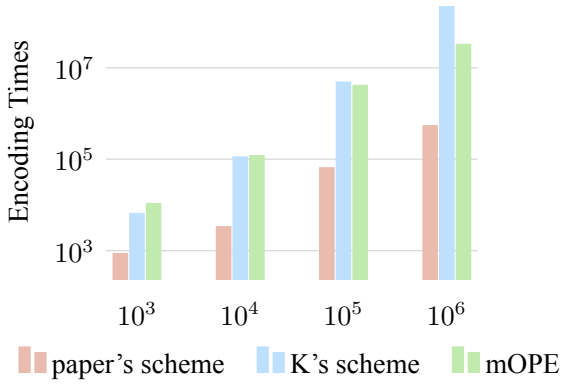


图 26: Worst Case(Distribution 2)

6.4 交互轮数

由于在各个代码的实现中使用了 mysql 作为交互对象，因此对于插入一条数据的的交互轮数可以用 mysql 用户定义函数"PRO_INSERT"2来评估；而对于查询一条数据的交互轮数可以使用调用 mysql 语句"SELECT" 来评估。在对于查找交互轮数的测评中，将数据集大小和服务器状态纳入综合考虑。其中服务器状态指的是在服务器端存储的数据是否是有序状态，将无序称为 cold 状态，有序称为 warm 状态。

插入交互轮数

对于该文章方案、K's scheme 和 compress K 来说，由于插入位置都是在客户端进行寻找，随后发送到服务器进行插入，不涉及到额外交互，因此对于每条数据交互轮数都为 1，总交互论述为。而对于 mOPE 方案来说，理论交互轮数为 $O(\log n)$，因此随着数据的逐条插入，交互轮数逐渐增加。从图中可以看出完全基于服务器的存储方案在插入交互轮数上的劣势。

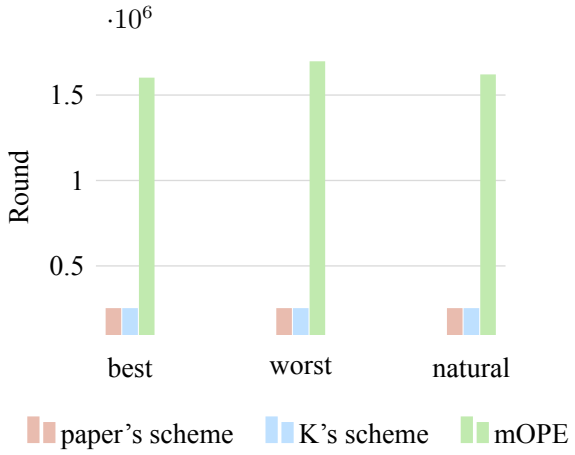


图 27: Insertion Interaction Rounds of Job Title

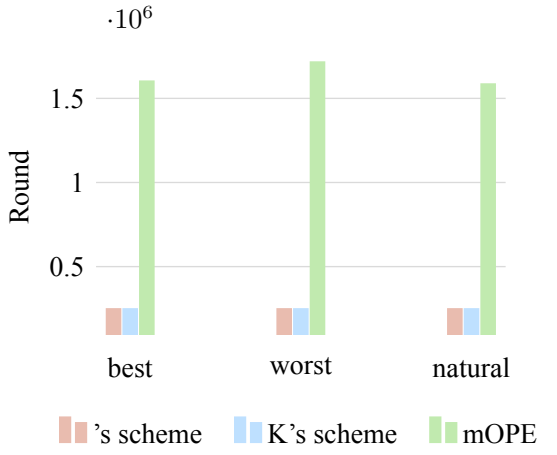


图 28: Insertion Interaction Rounds of Other pay

查找交互轮数

对于查找而言，由于该文章方案、K's scheme 和 compress K 是在客户端上寻找 pos，因此不需要与服务器进行额外交互，只需要进行一次发送，而服务器就能发回所有符合查询范围的条件。并且相对而

言该文章方案在客户端上的寻找代价也更小。对于 mOPE 方案来说同样需要 $O(\log n)$ 交互轮数。POPE 方案则对服务器状态敏感，当第一次查询时服务器中数据无序，此时交互轮数将会更多；而第一次查询后数据被按序排列，交互轮数相对减少，但总体而言依旧是 $O(\log n)$ 复杂度。并且可以看到在两个数据集上几种方案的表现都是相似的，这说明交互论述对于数据集特征变化并不敏感，只和数据集大小有关。

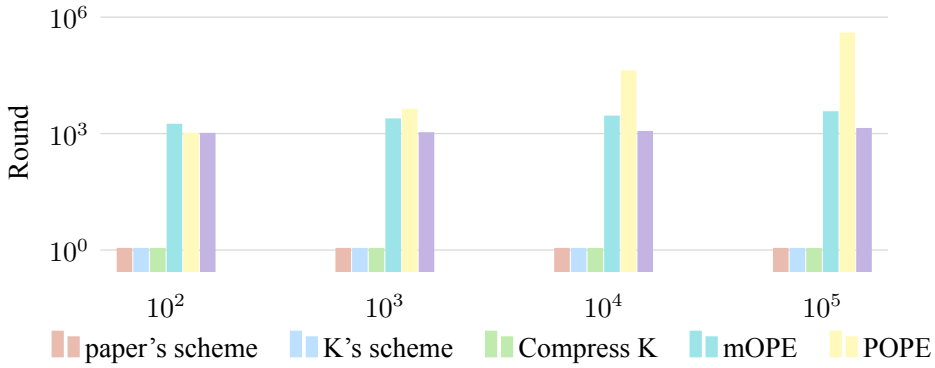


图 29: Interaction Rounds for "Job Title"

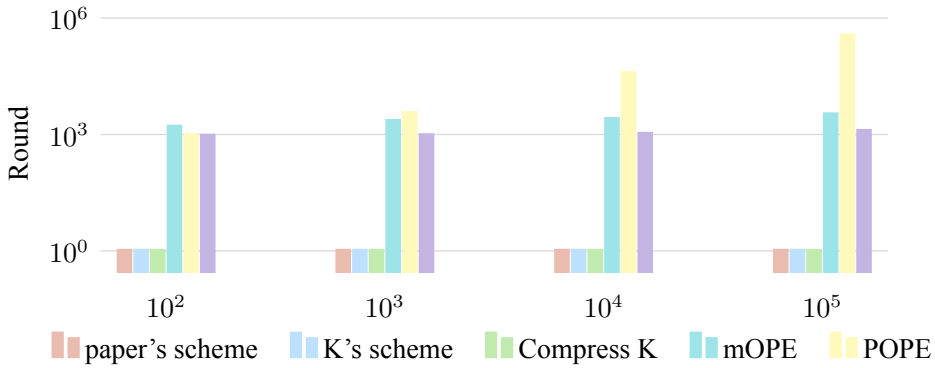


图 30: Interaction Rounds for "Job Title"

6.5 总体应用时间效率

最后对各个方案在 MYSQL 实际应用上的时间进行评估。

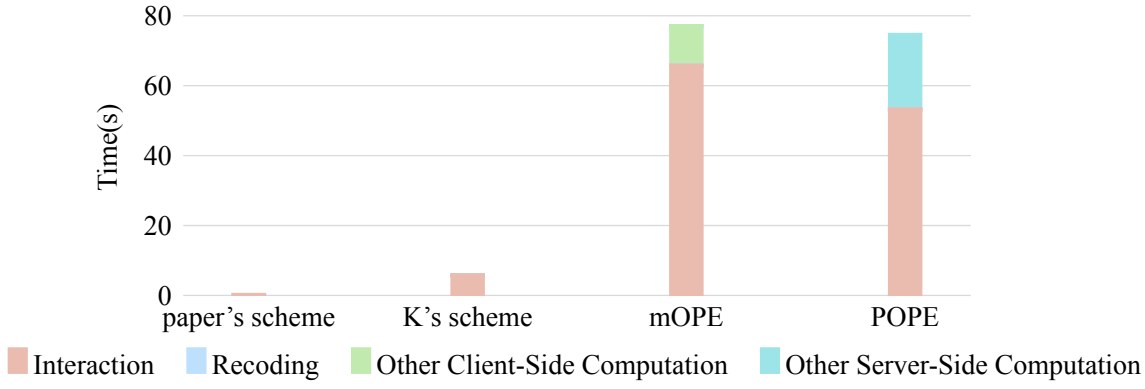


图 31: Insertion Interaction Rounds of Other pay

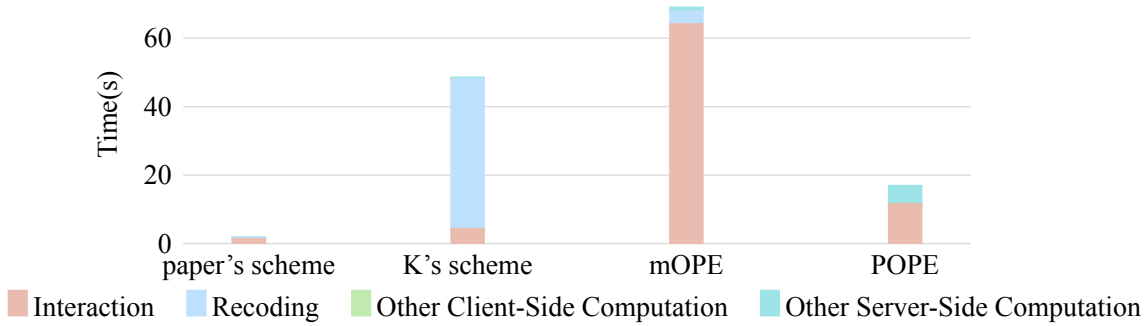


图 32: Insertion Interaction Rounds of Other pay

7 Data

7.1 Client Storage Cost(Generate Data Set)

n(Data Nums)	N/n	Scheme		
		lcope	K'scheme	Compress
10000	0.5	0.7651	1.1444	2.5471
	0.01	0.5271	1.1444	1.9592
	0.001	0.3252	1.1444	0.8607
	0.0001	0.0346	1.1444	0.0488
	0.00001	0.0351	1.1444	0.0387
	0.000001	0.0342	1.1444	0.0304
100000	0.5	0.4007	14.4444	18.8641
	0.01	0.3576	14.4444	13.3941
	0.001	0.3636	14.4444	4.8796
	0.0001	0.3516	14.4444	4.2445
	0.00001	0.3561	14.4444	3.8742
	0.000001	0.3541	14.4444	3.7776

References

[1] Raluca Ada Popa, Frank H. Li, and Nickolai Zeldovich. “An Ideal-Security Protocol for Order-Preserving Encoding”. In: 2013. DOI: 10.1109/SP.2013.38.

[2] Florian Kerschbaum. “Frequency-Hiding Order-Preserving Encryption”. In: 2015. URL: https://doi.org/10.1145/2810103.2813629.

[3] Daniel S. Roche et al. “POPE: Partial Order Preserving Encoding”. In: 2016. DOI: 10.1145/2976749.2978345. URL: https://doi.org/10.1145/2976749.2978345.

[4] Dongjie Li et al. “Frequency-Hiding Order-Preserving Encryption with Small Client Storage”. In: (2021). ISSN: 2150-8097. DOI: 10.14778/3484224.3484228. URL: https://doi.org/10.14778/3484224.3484228.