

编号: CACR2022UXUS5G



作品类别: ☒ 软件设计 ☐ 硬件制作 ☐ 工程实践 ☐ 密码应用技术

2022 年第七届全国密码技术竞赛作品设计报告

题目: 轻量级客户端存储的频率隐藏保序加密方案

2022 年 10 月 26 日

中国密码学会

基本信息表

编号：CACR2022UXUS5G

作品题目：轻量级客户端存储的频率隐藏保序加密方案

作品类别：☒ 软件设计 ☐ 硬件制作 ☐ 工程实践 ☐ 密码技术应用

作品内容摘要：

目前密态数据库的应用引起了学术界和工业界的广泛关注，阿里云、华为高斯数据库都普遍开始对数据库进行加密。而对密态数据库的范围查询通常使用保序加密（Order-Preserving Encryption）算法来实现，该算法能够根据明文顺序输出对应编码。但由于明文的频率泄漏使得保序加密算法容易受到频率分析攻击，因此频率隐藏的保序加密方案（Frequency-Hiding Order-Preserving Encryption）意义非凡。

然而，现有方案要么需要大量的客户端存储（ $O(n)$ ），要么需要 $O(\log n)$ 轮额外交互（其中 n 是明文的总数），性能都存在着一定的影响。为此，我们提出了一种轻量级的隐藏频率的保序加密方案，该方案无需额外的客户端-服务器交互即可实现轻量客户端存储。具体来说，我们的方案实现了 $O(N)$ 客户端存储以及零额外交互（其中 N 是不同明文的数量）。在我们的方案中，我们设计了一个带有十分新颖且高效编码策略的B+树，用于高效检索密文和维护顺序信息。我们的编码策略显著减少了编码更新的频率，从而提高了将密文插入数据库的效率。

实验结果显示，我们的加密算法在加密时间、查询时间等方面均有着较为理想的表现，与过去传统的OPE加密方案相比，性能提升较为显著。

关键词（五个）：

保序加密；密态数据库；应用密码学；密文查询；数据安全

1.作品功能与性能说明

1.1 作品功能说明

本加密算法基于目前较为流行的保留顺序加密（OPE，Order Preserving Encryption）能在加密的同时保留明文的顺序信息，允许不可信服务器无需解密就可以完成比较以及较为范围查询（>, <）等操作，并且本算法能够融合进入市面上较为常用的关系型数据库系统，例如 MySQL、postGRESQL 等数据库。其对明文的加密方式的底层实现一般为对称加密算法（例如国密加密算法 SM4、国际通用的 AES 加密算法），而其密码原语的实现机制一般是对明文的顺序信息进行编码，例如 *mOPE* 以及 2015 年 Kerschbaum 等人提出的保留顺序且隐藏频率的保序加密算法（Frequency-Hiding Order Preserving Encryption），就是基于二叉树的编码方案实现的顺序比较。

本加密算法不仅能够实现在客户端存储隐私数据和在服务器端建立密文索引，便使服务器端在与客户端正常交互的情况下对密文进行范围检索，并能保证较高的安全性，而且能够实现较为轻量级的客户端存储量级和较高的安全性，充分保护用户的数据安全。

1.2 作品性能说明

针对本加密算法，将进行时间统计，收集和比较在不同明文域和密文域上进行加密、解密的时间；同时对性能测试以在 MySQL 中的直接操作和经过算法加密后的操作行为对比展开，分别测试插入、查询的平均操作时长，进行比较分析。

本加密算法在三个维度上的性能说明如下：

- **客户端存储：**在不同数据量的（1,000 - 1,000,000 条）的数据集上，本算法要求的客户端存储远远小于传统的保序加密算法，而且也优于现有的一些隐藏频率的保序加密算法。并且，本算法所需的客户端存储量随着数据量的增长也较为缓慢，可以支持较为庞大的数据集。
- **数据库查询性能：**实验数据显示，对于本轻量级客户端存储的频率隐藏的 OPE 加密算法，查询效率在数据频次信息明显的情况下与明文检索效率相差在最坏情况下仅有 30% 的开销增加。相较 DICT-OPE 等传统 OPE 加密算法而言，本算法具有查询、插入效率上的优势。
- **加、解密时间：**即使不进行任何优化，本保序算法也能比传统的 Kerschbaum 的 DICT-

OPE 加密算法在加解密的性能上更为高效；如果进行了算法优化，最坏情况下，性能也能提高约 8%，在一般情况下能够做到优化约 20% 的加、解密时间。因此本文提出的轻量级客户端存储的频率隐藏的保序加密算法能够达到较高的加、解密性能，从而实现节省存储密集型的应用程序在数据库插入的时间的目的，降低由安全性提高而带来的性能损失。

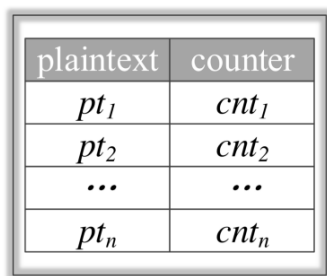
2. 设计与实现方案

2.1 实现原理

2.1.1 算法思路

算法的大致思路为在客户端存储每个数据出现的次数 *counter*，在进行加密的过程中，将数据及其出现的次数 *counter* 共同作为参数输入到加密算法中；在服务器端中，建立数据的编码索引树，以数据的位置为索引信息进行搜索。在客户端与服务器端进行交互时，是以数据的位置信息为桥梁进行沟通。

在本文提出的算法中，需要在客户端存储名为 *local table* 的辅助查询表，用于记录每个相同的明文出现的次数，初始值都为 0；在服务器端，建立编码树。如下图所示。



plaintext	counter
pt_1	cnt_1
pt_2	cnt_2
...	...
pt_n	cnt_n

图 2-1 客户端建立的查询表 *local table*

此外，为了减少服务器和客户端之间的交互过程，同时提高检索、插入等操作的效率，我们在服务器端建立了一个类似于 B+树的编码树结构。由于 B+树能够保证叶节点都在同一层，这样我们将密文保存在叶节点就可以保证每个节点的查询效率都一致；而且，每个外部节点都可以保存它们的子节点的信息，更加适应 OPE 的加密目标了，图 2-2 说明了不同节点类型的存储结构。

以 m 阶 B+树为例，其中：

- 内部节点 (Interval Node)：内部节点含有指向子节点的指针，以及每个子节点对应

的关键词 kwd 。这些关键词用以完成插入和搜索操作。

- 叶节点（Leaf Node）：叶节点存储每个已经插入的密文 ct 和它对应的编码信息 cd ，以及指向其兄弟节点的指针（用于合并、删除等操作）；此外，每个叶节点还含有一个区间 $[lower, upper]$ 用以表示编码的区间。当叶节点含有的密文数量大于 m 的时候，我们就需要对其进行分裂以保证树的平衡。
- 关键词（Keyword）：每个关键词 kwd 代表该关键词对应的子节点的所有后继节点的 OPE 密文的数量。
- 坐标（Position）：由客户端的加密函数生成。一旦有一个新的明文 pt 需要被加密，客户端先会查询本地的计数表用以获取该明文对应的重复次数 $T[pt]$ 。该明文对应的密文的坐标 pos 便从这些重复值的相对顺序中进行选取，即 $pos \leftarrow U\{\sum_{pt' < pt} T[pt'], \sum_{pt' < pt} T[pt'] + 1, \dots, \sum_{pt' \leq pt} T[pt']\}$ 。

图 2-3 展示了客户端 *local table* 和服务端 *encoding tree* 编码树的总体架构和交互示意。

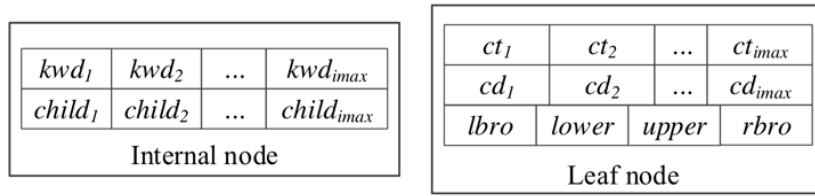


图 2-2 节点类型示意图

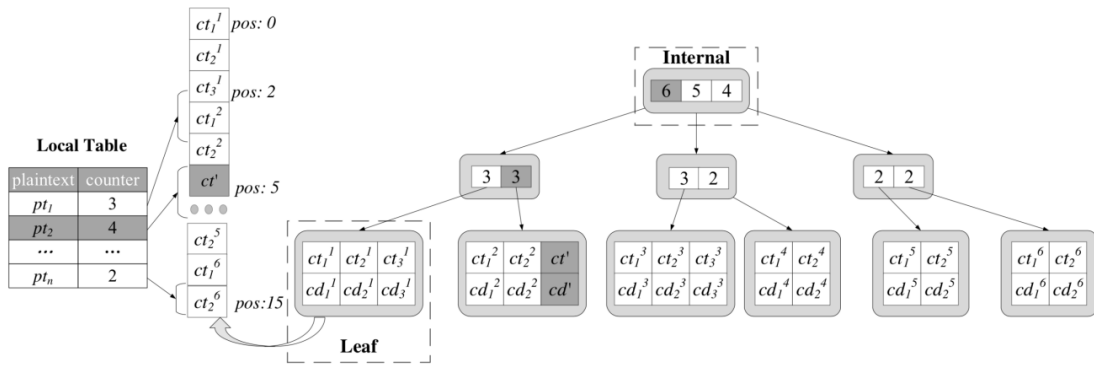


图 2-3 B+树和客户端对应明密文示意图

2.1.2 服务器端编码策略

由于服务器端有一个树结构用以保持密文之间的相对顺序，所以当我们想要插入或者变更密文的时候，就需要进行调整。但是，对于体积比较庞大的树结构而言，调整起来往往代价很高。因此，我们认为对密文的编码需要降低这种比较频繁而且较为昂贵的调整。本文在

此给出区间编码策略，用以抵消传统 OPE 编码的问题。

当我们想要将一个新的密文插入到叶节点中时，服务器会先查看其对应的叶节点的左右邻居的编码中值。如果这个密文恰好是叶节点中密文的首位或是末尾，那么我们会将这个叶节点对应的区间端点作为其左右邻居的编码。例如，当有一个新的密文 ct 要插入到介于编码 cd_1 和 cd_2 之间的时候，那么 ct 的编码 $cd = \frac{cd_1 + cd_2}{2}$ 。

这样编码的好处在于能够使得任意两个密文之间的相对顺序都能够得以保持，而且在对 B+树平衡的时候，顺序信息依旧能够得到保证。

为避免频繁更新编码，当明文发生更新的时候，我们只在必要情况下进行重新编码。当且仅当新的密文无法获得任何有效编码的时候才会发生重新编码的工作（事实上这种情况极少发生）：它临近的所有密文或者兄弟节点的所有密文都要均匀地进行重新编码。例如，若有个叶节点的对应区间为 $(0,8]$ ，且含有密文序列 $ct = \{ct_1, ct_2, ct_3\}$ ，且对应编码为 $cd = \{cd_1 = 1, cd_2 = 3, cd_3 = 4\}$ 。当密文 ct' 想要插入到 ct_2 和 ct_3 之间的时候， cd_2 和 cd_3 已经无法容纳新的编码了，那么整个节点都需要重新编码。这四个密文的编码会被重写成 $cd = \{2, 4, 6, 8\}$ 。

2.1.3 算法具体描述

1. 客户端

(1) 初始化

客户端在本地建立一张新的空表 *local table* 用以加密。

(2) 加密

注意服务器端的编码树需要密文的坐标 pos 才能完成对应操作，而这个数据仅能通过客户端进行，因此客户端想要搜索、插入、删除、更新数据的时候，必须先对明文进行一次加密才能找到对应服务器中的密文和坐标，这样才能完成所需操作。

因此客户端必须有加密函数，同时更新本地查询表。

伪代码如下：

Algorithm 6: Encryption on the client side

Input: sk, st_{Cl}, pt
Output: pos, ct

```

1  $ct \leftarrow \text{RNEnc}(sk, pt);$ 
2 get  $T$  from client  $st_{Cl}$ ;
3  $l \leftarrow \sum_{pt' < pt} T[pt'];$ 
4 if  $pt \in T$  then
5   uniformly sample  $pos \leftarrow \{l, l+1, \dots, l+T[pt]\};$ 
6    $T[pt] \leftarrow T[pt] + 1;$ 
7 else
8    $pos \leftarrow l;$ 
9   insert  $pt$  into  $T$ ;
10   $T[pt] \leftarrow 1;$ 
11 end
12 set  $T$  as  $st'_{Cl}$ ;
13 return tuple( $pos, ct$ );
```

代码2-4 客户端加密函数

(3) 解密

因为密文和编码无关，所以解密仅需调用本地解密函数 $RNDDec$ 即可。

算法伪代码如下：

Algorithm 7: Decryption on the client side

Input: sk, ct
Output: pt

```

1  $pt \leftarrow \text{RNDDec}(sk, ct);$ 
2 return  $pt;$ 
```

代码2-5 客户端解密函数

(4) 插入

客户端需要插入一个数据到表中时，它首先需要生成密文在本地表中的坐标信息 pos ，并且将密文 ct 也同时发送给服务器。这样后者就可以决定插入位置，该过程由服务器端的 UDF 实现，客户端实际上只是一个提供接口的包装器。算法伪代码如下：

Algorithm 8: Insert on the server side

Input: sk, pt, st_{Cl}

```

1  $pos, ct \leftarrow \text{Enc}(sk, st_{Cl}, pt);$ 
2  $\text{FHInsert}(pos, ct);$ 
```

代码2-6 客户端插入函数

(5) 查询

当客户端想要查询某个数据的时候，SQL 语句将会被改写。假设原始语句为

SELECT * FROM University WHERE ID > ptmin AND ID <= ptmax;

它会被改写成

```

SELECT *
FROM University
WHERE encoding >  $\text{FHSearch}(pos\_min)$ 
```

AND

encoding < FHSearch(pos_max);

其中 pos_min 和 pos_max 分别代表了该明文在本地表中出现的区间的首尾范围；严格来说，我们有

$$pos_{min} = \sum_{pt' \leq pt_{min}} T[pt'] ,$$

以及

$$pos_{max} = \sum_{pt' < pt_{max}} T[pt'] + 1 .$$

其中函数 FHSearch 是服务器端的自定义函数，详细可见下节服务器编码树算法介绍。

2. 服务器端

为了更好地支持服务器端构建的编码树，我们需要为其定义诸多 UDF(用户自定义函数)，并以动态链接库的形式存储到远程的服务器中。同时，为了更好地说明本文介绍的加密算法过程，在本小节也会以图的形式来说明。

(1) 插入

该算法输入为编码树的根节点 *root*，由客户端加密得到的密文 *ct* 以及其对应的坐标 *pos*。服务器会不停地将所有节点的关键词和 *pos* 进行比较以定位密文 *ct* 所在的叶节点的位置。确定好叶节点之后，服务器会调用编码函数对叶节点新插入的密文作编码，编码也是通过 Encode(leaf, pos + 1) 进行实现。若叶节点发生了分裂，那么 B+ 树会做一次分裂并调整；若叶节点编码无法实现，则重置叶节点的所有编码。最后，该算法会返回密文对应的编码，存储到表中。客户端仅需调用一次插入函数的接口即可。伪代码如下：

Algorithm 9: Insert on the server side

```

Input: node, pos, ct
Output: cd, lower, upper
1 if node is a leaf node then
2   insert ct into node as the (pos + 1)-th ciphertext;
3   node.cdpos+1, lower, upper ← Encode(node, pos+1);
4   if node contains more than m elements then
5     Rebalance(node);
6   end
7 else
8   i ← 1;
9   while pos > node.kwdi do
10    pos ← pos - node.kwdi;
11    i ← i + 1;
12  end
13  Insert(node.childi, pos, ct);
14 end
15 return node.cdpos+1, lower, upper;

```

代码2-7 服务器端B+树插入函数

Algorithm 10: Rebalance

Input: *node*
Output: *root*

```

1 if node is leaf node then
2   create a new leaf node' ;
3   move this node's last  $\lfloor \frac{m}{2} \rfloor$  ciphertexts with their encodings to node';
4   insert node' between node and node.rbrother;
5 else
6   create a new internal node node';
7   move node's last  $\lfloor \frac{m}{2} \rfloor$  children with their keywords to node';
8 end
9 if node.parent = NULL then
10  create a new internal node root;
11  insert node and node' into root;
12  node.parent, node'.parent  $\leftarrow$  root, root;
13  set root as the root of the encoding tree;
14 else
15  insert node' into node.parent as node.brother;
16  node'.parent  $\leftarrow$  node.parent;
17  if node.parent contains more than m nodes then
18    Rebalance(node.parent);
19  end
20 end
21 return root

```

代码2-8 服务器端B+树平衡函数

(2) 平衡

该算法将以 B+树的根节点 $root$ 作为输入。如果某个节点 $node$ 超过了 B+树的最大节点容量，即 m 阶 B+树中若有节点超过 m 个（为方便期间我们定义节点中的最大容量为 $imax$ ， $node.imax > m$ ），那么算法会以回溯的形式不断地将容量超过最大规定容量的节点进行分裂并向上传递，直至根节点为止。如果节点 $node$ 需要被分裂，那么它的后 $\lfloor \frac{m}{2} \rfloor$ 的节点会被移动到新的节点，叫做 $node'$ ；随后 $node'$ 会被插入到 $node$ 的父节点中，作为 $node$ 的右兄弟。另外需要指出的是，若分裂的节点 $node = root$ ，那么该算法会创建一个新的根节点作为服务器的编码树的根节点。算法的伪代码如图 2-8 所示。

(3) 重新编码

该算法会接受一个叶子节点 $leaf$ 作为输入。如果编码过程触发了重新编码函数，那么该函数会重新分配叶子节点 $leaf$ 和其兄弟节点的编码。如果该叶子节点 $leaf$ 超过了 $imax$ ，服务器会均匀地为它们分配编码。否则，在编码过程中， $leaf$ 的兄弟节点会逐渐聚合堆积成一个以块为单位的链表。值得注意的是，若最右端的 $leaf$ 的区间范围已经不足以为编码分配任何空间，服务器会自动地扩展区间能够承受的上限到原来的两倍。最后，该算法会输出编码范围 $(lleaf.lower, rleaf.upper]$ 作为 $leaf$ 端的区间范围。图 2-11 和图 2-12 展示了插入密文和这一重新编码的具体过程。

算法伪代码如 2-9 所示。

(4) 获取编码

该算法会以编码树的根节点 $root$ 和某个待加密的明文的位置 pos 作为输入。服务器会不断地将 pos 和当前节点的关键词进行比较，以定位到 pos 所属的叶节点中。最后该算法会输出 pos 对应的编码 cd 。

算法伪代码如 2-10 所示。

Algorithm 11: Recode	
Input: $leaf$	
Output: $lower, upper$	
1	$lleaf, rleaf \leftarrow leaf, leaf;$
2	$imax \leftarrow leaf.imax;$
3	if $rleaf.upper - lleaf.lower \geq imax$ then
4	$frag \leftarrow \lfloor \frac{rleaf.upper - lleaf.lower}{imax} \rfloor;$
5	$cleaf \leftarrow lleaf;$
6	$cd \leftarrow lleaf.lower;$
7	for $i \in [cleaf.imax]$ do
8	$cd \leftarrow cd + frag;$
9	$cleaf.cd_i \leftarrow cd;$
10	if $cleaf \neq rleaf$ then
11	$cleaf.upper \leftarrow cd;$
12	$cleaf \leftarrow cleaf.rbrother;$
13	$cleaf.lower \leftarrow cd;$
14	continue;
15	else
16	break;
17	end
18	end
19	else
20	if $lleaf.lbrother \neq NULL$ then
21	$lleaf \leftarrow lleaf.lbrother;$
22	$imax \leftarrow imax + lleaf.imax;$
23	end
24	if $rleaf.rbrother \neq NULL$ then
25	$rleaf \leftarrow rleaf.rbrother;$
26	$imax \leftarrow imax + rleaf.imax;$
27	else
28	$rleaf.upper \leftarrow rleaf.upper \times 2;$
29	end
30	goto 3;
31	end

代码2-9 Recode函数

Algorithm 12: GetCode

Input: $node, pos$
Output: cd

```

1 if  $node$  is leaf node then
2    $cd \leftarrow node.cd_{pos};$ 
3 else
4    $i \leftarrow 1;$ 
5   while  $pos$  is bigger than  $node.kwd_i$  do
6      $i \leftarrow i + 1;$ 
7      $pos \leftarrow pos - node.kwd_i;$ 
8   end
9    $cd \leftarrow GetCode(node, child_i, pos);$ 
10 end
11 return  $cd;$ 
    
```

代码2-10 GetCode函数

(5) 示例

该小节会给出一个加密的示例，用以说明插入密文的时候 Recode 和 Rebalance 之间的关系是怎样的。考虑一个取值只有两种情况的案例：性别。显然，该属性仅有“男”和“女”两种可能，在此表示为男（1），女（0）。假设服务器端构建了一棵 3 阶 B+树作为编码树，且根节点 $root$ 对应的区间为 $(0, 16]$ 。另外假设现有一组明文序列待加密，为 $\{0, 1, 0, 1, 0, 1, 0, 1\}$ ，并且已经插入了 $\{0, 1, 0, 1, 0, 1\}$ 到编码树中了。目前的状态——本地存储的计数表 *local table* 和编码树在图 2-8(a)中给出，为方便起见，图中的 $E(*)$ 代表了加密函数。

下面将展示服务器和客户端如何共同完成加密剩余明文 $\{0, 1\}$ 的过程。

首先需要加密明文 0。服务器查询本地计数表得到一个随机位置 $pos = 0$ ，以及 0 的对应密文 $E(0)$ 至服务器，随后服务器插入 $E(0)$ 到第一个叶子节点，并为其分配了编码 2。图 2-12(b) 描述了这一过程，并且该过程出发了一次 Rebalance，因为该叶子节点的密文数量超过了 3（该示例中为 3 阶 B+编码树）。图 2-12(c) 展示了平衡后的 B+树，但是密文的编码并未改变。

接下来需要加密明文 1。客户端首先查询本地计数表得到一个随机位置 $pos = 6$ ，以及 1 的对应密文 $E(1)$ 至服务器，随后服务器插入 $E(1)$ 到第三个叶子节点，并为其分配了编码 13。图 2-12(d) 描述了这一过程，并且该过程触发了一次 Rebalance，因为该叶子节点的密文数量超过了 3。图 2-12(e) 展示了平衡后的 B+树，注意到平衡函数调用了两次。

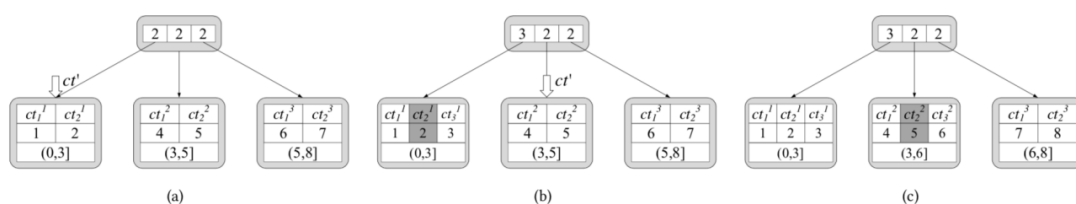


图2-11 重新编码例程

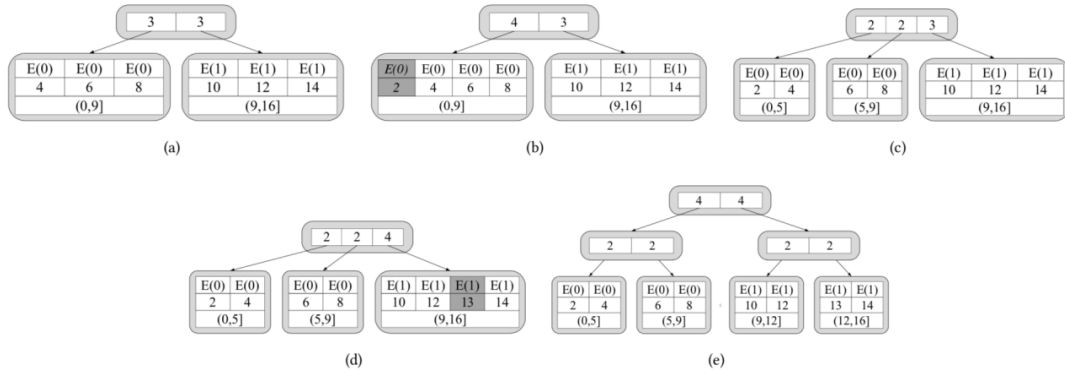


图2-12 插入和平衡示例

图 2-7 说明了何时会发生编码的重置。

当插入一个新的密文到第一个叶节点的密文 ct_1 和 ct_2 时（见图 2-12(a)所示），编码树会为其分配一个编码；然而由于该节点的编码为 1 和 2，已经没有更小的编码能分配了（见图 2-12(b)所示）。因此，编码树会重置该层所有的叶节点的编码，重置后的结果如图 2-12(c)所示，此时新插入的密文已经被分配到第三个叶节点之中了。

(6) 搜索

B+树是一类特殊的搜索树，而服务器端保留了保序的编码，因此搜索过程仅需获取 *encoding* 的最大最小值即可。例如客户端想要做一次查询“SELECT * FROM table WHERE ID >= rmin AND ID < rmax”，该语句仅需被改写成“SELECT * FROM table WHERE ID >= Search(posmin) AND ID < Search(posmax)”。

3. 优化

如果在每次插入后都要检查 B+树中各个密文的编码是否满足条件，那么在明文插入次数和重复次数过多的时候，会导致客户端加密效率的下降。考虑到这一点，基于 B+树的编码方式的 OPE 加密算法的更新频次需要降低。由于密文暂时失去保序性质在客户端进行查询之前是允许的，因此该 OPE 算法的更新操作可以与客户端查询绑定，即查询之前发起一次 Update 请求，对 B+树整体的编码进行更新。然而，如果插入不更新的话，也会导致查询效率的降低。从这个角度来看，插入函数需要设置一个更新上界 *upper_bound*，若插入次数超过该计数之后就要对 B+树进行整体更新。

2.2 运行结果

由于本加密算法的主要应用场景为数据库系统，当用户想要查询数据的时候就可以采用

算法定义的查询。

本文选取实现的数据库为 Oracle 公司的 MySQL 以及其提供的 MySQL-Proxy 代理服务器作为密态代理和 SQL 解释器的运行环境。将在服务器端的操作以用户自定义函数（UDF）的方式写入数据库，将数据库中的具体数据和索引树相连，这样就可以实现对密文的搜索，同时加速在数据库中的搜索操作。当服务器端的函数链接完成后，客户端的密态模块就能够调用这些加密接口函数了。

```
g++ -c -o ope.o -std=c++11 -I usr/include/mysql -fPIC -Wall ope.cc
g++ -shared -o ope.so ope.o
```

如上所示，我们对密码算法文件进行编译。使用UDF编写的接口文件代码如下所示。

```
extern "C"
{
    //typedef double longlong;
    //插入
    my_bool FHInsert_init(UDF_INIT *initid, UDF_ARGS *args, char *message);
    double FHInsert(UDF_INIT *initid, UDF_ARGS *args, char *is_null, char *error);
    //搜索
    my_bool FHSearch_init(UDF_INIT *const initid, UDF_ARGS *const args, char *const message);
    double FHSearch(UDF_INIT *const initid, UDF_ARGS *const args,
        char *const result, unsigned long *const length,
        char *const is_null, char *const error);

    //更新
    my_bool FHUpdate_init(UDF_INIT *initid, UDF_ARGS *args, char *message);
    double FHUpdate(UDF_INIT *initid, UDF_ARGS *args, char *is_null, char *error);
    //更新范围
    my_bool FHStart_init(UDF_INIT *initid, UDF_ARGS *args, char *message);
    double FHStart(UDF_INIT *initid, UDF_ARGS *args, char *is_null, char *error);
    my_bool FHEnd_init(UDF_INIT *initid, UDF_ARGS *args, char *message);
    double FHEnd(UDF_INIT *initid, UDF_ARGS *args, char *is_null, char *error);
}
```

图2-13 UDF声明

```
CREATE FUNCTION FHInsert RETURNS REAL SONAME 'ope.so';
CREATE FUNCTION FHUpdate RETURNS REAL SONAME 'ope.so';
CREATE FUNCTION FHSearch RETURNS REAL SONAME 'ope.so';
CREATE FUNCTION FHEnd RETURNS REAL SONAME 'ope.so';
CREATE FUNCTION FHStart RETURNS REAL SONAME 'ope.so';
```

图2-14 添加用户自定义函数UDF

我们尝试加密采用的数据集来自于实际生活中的数据，中国移动手机用户与服务商 TalkingData 的交互信息（下称“中国移动”）以及 DBLP 公开数据集（下称“DBLP”），将本文的加密算法通过 User-Defined Function（UDF）的形式链接到 MySQL 数据库中，通过频率隐藏的保序加密算法进行加密“年龄”数据列后，能够获得如下的密文和编码数据：

plaintext	ciphertext（通过 base64 编码）	encoding
35	frvblFS8nktkqcASGQ5B/Xreh0TlyJeiOGM51BMvles=	1.09951E+12
35	Uy2USx8Rot3gtUcJK1F7Nnreh0TlyJeiOGM51BMvles=	1.64927E+12
35	cH8i5n2YWKGG8P9PNf07lXreh0TlyJeiOGM51BMvles=	1.92415E+12
30	df5MNgxeg5oosu9qnUdoy3reh0TlyJeiOGM51BMvles=	5.49756E+11

30	337KcAhXz3uh81xWT9OZEHreh0TlyJeiOGM51BMvles=	8.24634E+11
24	yRfCLBEz5bWtpX8YaPFmmHreh0TlyJeiOGM51BMvles=	2.74878E+11
36	jKxMC3MulYNM9ou1mMHh1Hreh0TlyJeiOGM51BMvles=	2.06158E+12
38	ohimLatIMMTAjEq67WnhG3reh0TlyJeiOGM51BMvles=	2.1303E+12
33	Ti0FyXT1cSYeWvWpn1Pq2Hreh0TlyJeiOGM51BMvles=	9.62073E+11
36	IaymPMGSur6B4VC1EQr/p3reh0TlyJeiOGM51BMvles=	2.09594E+12

表 2-1 通过 OPE 加密后的部分数据信息

可以看到，保序性质是成立的，而且也不存在重复的密文。

2.3 技术指标

本加密算法评估的技术指标主要有以下三个方面：

- 加密时间：可以通过数据库插入时间来评估；
- 查询时间：可以通过数据库密文检索效率来评估；
- 客户端存储：可以通过客户机中 *local table* 的容量（以字节计）来评估

3. 系统测试与结果

3.1 测试方案

我们以较为典型的客户端-服务器模型为基本的测试模型，其中客户端为普通的用户应用程序，而服务器端部署了具有加密算法的数据库系统，其中算法以 C++ 语言实现。我们将我们的加密算法和其他较为著名的频率隐藏的保序加密算法进行对比，其中有 *mOPE* 和 Kerschbaum 提出的 FH-OPE 方案。

第一，在功能实现部分，本文会分别对前文所述的两个频率隐藏的加密方案各项功能进行审查。方案一主要包括对改进后的算法在不同的明文域和密文域下进行加密、解密测试，从而来验证保序性和对称性；方案二主要包括 SQL 的插入、搜索两项基本操作，并分客户端、服务器端两个方面进行：对客户端加解密进行测试，即是否可以做到隐藏频率的加密，是否可以正确的解密服务器返回的密文；对服务器支持的操作进行测试，即是否可以在密文的状态下进行插入、搜索、删除和更新的操作。

第二，在功能测试过程中，本文将进行时间统计，收集性能测试的数据。方案一将统计对不同明文域和密文域的算法进行加密、解密的平均时间；方案二对算法的性能测试以在

MySQL 中的直接操作和经过算法加密后的操作行为对比展开，分别测试插入、查询的平均操作时长，进行比较分析。

第三，本文将对数据库系统的采用的新型加密算法的安全性进行测试。由于这一测试联系到现实生活中的数据安全性，因此本文中采用的都是真实世界中的数据集，并且采用国际上较为认可的针对频率泄露的攻击手段来模拟敌手窃取加密数据的场景。

第四，实验环境设置。本文所述的数据库系统和加密算法用 C++ 语言实现（标准为 C++20 标准），而与之相关的实验均在阿里云 EC2 服务器上进行，其中 CPU 配置为 8 核心 Intel(R) Xeon W-2245 GPU @3.9GHz，内存容量为 32 GB，操作系统为 Ubuntu 20.04 LTS。我们使用国家密码局发布的国家标准 SM4 对称加密算法作为保序加密的底层加密算法，并用 Base64 编码。服务器端的编码树为 128 路 B+ 树。本实验仅使用单线程。

第五，数据集选取。本实验采用的数据集均来自于实际生活中的数据，包括新冠疫情 COVID-19 各国数据统计（下称“新冠疫情”）以及 DBLP 公开数据集（下称“DBLP”）。

3.2 功能测试

这里主要展示了本 OPE 算法在 MySQL 中对于密文的一些基本操作的实现。测试代码如下：

```
DELIMITER //
create procedure PRO_INSERT(IN pos int,IN ct varchar(128))
BEGIN
    insert into example values(ct,FHInsert(pos,ct));
END //
DELIMITER ;

DELIMITER //
create procedure PRO_DELETE(IN posMin int,IN posMax int)
BEGIN
    delete from example where encoding>FHSearch(posMin) and encoding<=FHSearch(posMax);
END //
DELIMITER ;

DELIMITER //
create procedure PRO_UPDATE(IN pos int,IN ct varchar(128))
BEGIN
    update example SET ciphertext=ct where encoding=FHSearch(pos);
END //
DELIMITER ;

DELIMITER //
create procedure PRO_QUERY(IN posMin int,IN posMax int)
BEGIN
    select * from example where encoding>FHSearch(posmin) and encoding<=FHSearch(posmax);
END //
DELIMITER ;
```

```
mysql> source udf.sql;
Query OK, 0 rows affected (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

Query OK, 0 rows affected (0.01 sec)

Query OK, 0 rows affected (0.00 sec)
```

图3-1 添加测试过程函数

(1) 插入数据

客户端通过调用 **PRO_INSERT(pos int,ciphertext varchar(128))**过程即可实现数据的插入。第一个参数 pos 表示插入的位置，第二个参数 ciphertext 表示插入的密文。下图给出了向“example”表中插入一条数据的结果。

```
mysql> call PRO_INSERT(0,"frvb1FS8nktkqcASGQ5B/Xreh0TlyJeiOGM51BMvles=");
Query OK, 1 row affected (0.01 sec)

mysql> select * from example;
+-----+-----+
| ciphertext | encoding |
+-----+-----+
| frvb1FS8nktkqcASGQ5B/Xreh0TlyJeiOGM51BMvles= | 9.223372036854776e18 |
+-----+-----+
1 row in set (0.00 sec)
```

图 3-2 插入数据测试结果

(2) 查询数据

客户端通过调用 **PRO_QUERY(posMin int, posMax int)**过程即可实现数据的查询。第一个参数 posMin 表示查询位置的最小值，第二个参数 posMax 表示查询位置的最大值，区间满足“左开右闭”的特性。下图给出了查询“example”表中 1 号位置和 2 号位置数据的结果。

```
mysql> select * from example;
+-----+-----+
| ciphertext | encoding |
+-----+-----+
| frvb1FS8nktkqcASGQ5B/Xreh0TlyJeiOGM51BMvles= | 9.223372036854776e18 |
| Uy2USx8Rot3gtUcJK1F7Nnreh0TlyJeiOGM51BMvles= | 1.3835058055282164e19 |
| ch8i5n2YwKGG8P9PNf071Xreh0TlyJeiOGM51BMvles= | 1.152921504606847e19 |
| df5MNgxeg5oosu9qnUdoy3reh0TlyJeiOGM51BMvles= | 4.611686018427388e18 |
| 337KcAhXz3uh81xWT90ZEhreh0TlyJeiOGM51BMvles= | 6.917529027641082e18 |
| yRfCLBEz5bWtpX8YaPFmmHreh0TlyJeiOGM51BMvles= | 2.305843009213694e18 |
+-----+-----+
6 rows in set (0.00 sec)

mysql> call PRO_QUERY(0,2);
+-----+-----+
| ciphertext | encoding |
+-----+-----+
| df5MNgxeg5oosu9qnUdoy3reh0TlyJeiOGM51BMvles= | 4.611686018427388e18 |
| 337KcAhXz3uh81xWT90ZEhreh0TlyJeiOGM51BMvles= | 6.917529027641082e18 |
+-----+-----+
2 rows in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)
```

图 3-3 查询数据测试结果

(3) 更新数据

客户端通过调用 **PRO_UPDATE(pos int, ciphertext varchar(128))**过程即可实现数据的更新。第一个参数 pos 表示更新数据的位置，第二个参数 ciphertext 表示更新数据的值。下图给出了更新“example”表中 0 号位置数据的结果。

```
mysql> call PRO_QUERY(0,1);
+-----+-----+
| ciphertext                                | encoding                                |
+-----+-----+
| df5MNgxeg5oosu9qnUdoy3reh0TlyJeiOGM51BMvles= | 4.611686018427388e18 |
+-----+-----+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

mysql> call PRO_UPDATE(1,"jKxMC3MulYNM9ou1mMHh1Hreh0TlyJeiOGM51BMvles=");
Query OK, 1 row affected (0.00 sec)

mysql> call PRO_QUERY(0,1);
+-----+-----+
| ciphertext                                | encoding                                |
+-----+-----+
| jKxMC3MulYNM9ou1mMHh1Hreh0TlyJeiOGM51BMvles= | 4.611686018427388e18 |
+-----+-----+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)
```

图 3-4 更新数据测试结果

(4) 删除数据

客户端通过调用 **PRO_DELETE(posMin int, posMax int)**过程即可实现数据的删除。第一个参数 posMin 表示删除位置的最小值，第二个参数 posMax 表示删除位置的最大值，区间满足“左开右闭”的特性。下图给出了删除“example”表中 1 号位置和 2 号位置数据的结果。

```
mysql> call PRO_QUERY(0,2);
+-----+-----+
| ciphertext                                | encoding                                |
+-----+-----+
| jKxMC3MulYNM9ou1mMHh1Hreh0TlyJeiOGM51BMvles= | 4.611686018427388e18 |
| 337KcAhXz3uh81xwT9OZEHreh0TlyJeiOGM51BMvles= | 6.917529027641082e18 |
+-----+-----+
2 rows in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

mysql> call PRO_DELETE(0,2);
Query OK, 2 rows affected (0.01 sec)

mysql> select * from example;
+-----+-----+
| ciphertext                                | encoding                                |
+-----+-----+
| frvb1FS8nktkqcASGQ5B/Xreh0TlyJeiOGM51BMvles= | 9.223372036854776e18 |
| Uy2USx8Rot3gtUcJK1F7Nnreh0TlyJeiOGM51BMvles= | 1.3835058055282164e19 |
| cH8i5n2YwKGG8P9PNf07lXreh0TlyJeiOGM51BMvles= | 1.152921504606847e19 |
| yRfCLBEz5bWtpX8YaPFmmHreh0TlyJeiOGM51BMvles= | 2.305843009213694e18 |
+-----+-----+
4 rows in set (0.00 sec)
```

图 3-5 删除数据测试结果

3.3 性能测试

为了评估 OPE 加密算法的效率，本实验将针对本小组设计的 OPE 算法在不同数据集和不同数据量下的加密效率，并将其插入到 MySQL 数据库中。并且，为了评估本 OPE 算法和其他原先提出的 OPE 加密算法的性能差异，本小节的实验将会与轻量级客户端存储的频率隐藏的 OPE 加密方案与 Kerschbaum 提出的 OPE 加密算法进行性能上的比较。方便起见，实验中记本小组提出的 OPE 算法为“FH-OPE”，优化后的 OPE 算法为“FHOPT-OPE”，而 Kerschbaum 提出的利用字典压缩的 OPE 算法为“DICT-OPE”。

此外，为了评估实际使用的效果，本小节将加密过程和插入过程同时进行。

（1）“感染人数”数据集加密性能测试

“感染人数”仅有将近 280000 条数据记录。因此本实验设计最多到 200000 条记录的插入，以此说明本加密算法的性能效果，以及和原有的频率隐藏的 OPE 加密算法的比较。实验将对明文域内所有明文进行一次加密，记录加密所需总时间，并以所得数据计算平均一次加密所需时间。图 3-6 给出了在不同明文数据量输入的情况下加密和插入的平均时间。

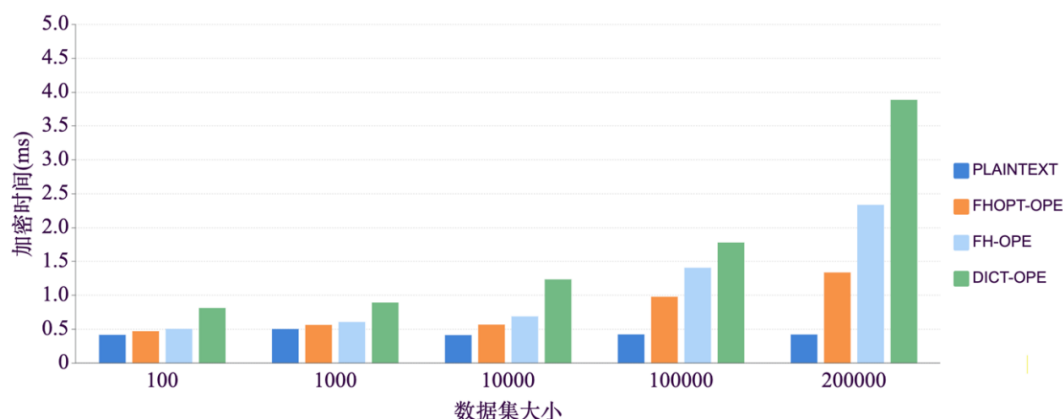


图3-6 利用不同OPE加密算法加密“感染人数”平均时间

（2）“年龄”数据集加密性能测试

“年龄”仅有将近 70000 条数据记录。因此本实验设计最多到 50000 条记录的插入，以此说明本加密算法的性能效果，以及和原有的频率隐藏的 OPE 加密算法的比较。实验将对明文域内所有明文进行一次加密，记录加密所需总时间，并以所得数据计算平均一次加密所需时间。图 3-7 给出了在不同明文数据量输入的情况下加密和插入的平均时间。

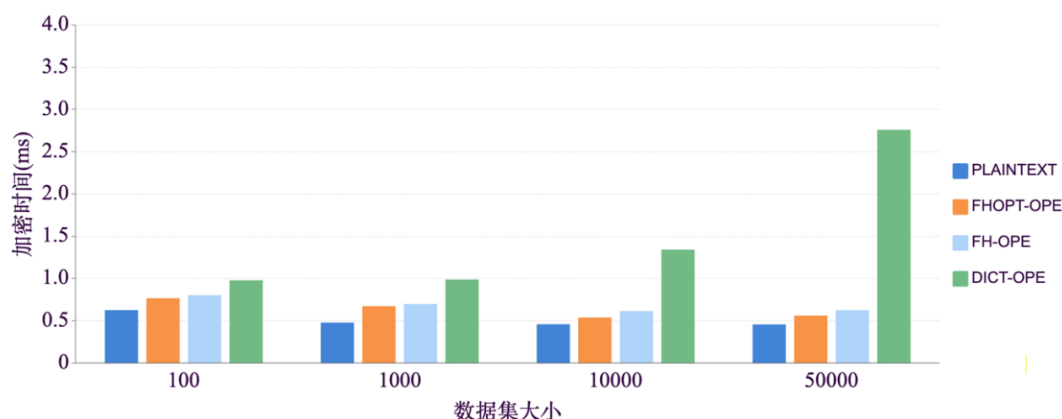


图3-7 利用不同OPE加密算法加密“年龄”平均时间

（3）“年份”数据集加密性能测试

本实验最多涉及到 100000 条记录的插入，以此说明本加密算法的性能效果，以及和原有的频率隐藏的 OPE 加密算法的比较。实验将对明文域内所有明文进行一次加密，记录加密所需总时间，并以所得数据计算平均一次加密所需时间。图 3-8 给出了在不同明文数据量输入的情况下加密和插入的平均时间。

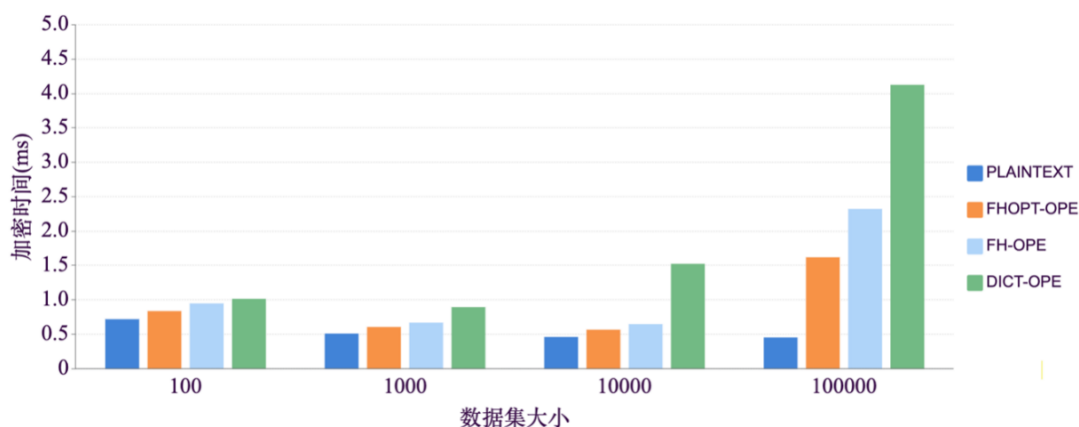


图3-8 利用不同OPE加密算法加密“年份”平均时间

3.4 存储性能测试

（1）“感染人数”数据集客户端存储测试

针对该加密算法，实验将评估本小组提出的OPE算法（FH-OPE）和传统的Kerschbaum提出的OPE算法的客户端存储，且选定数据集大小为至多200000条数据量。其测试结果如图

3-9所示。

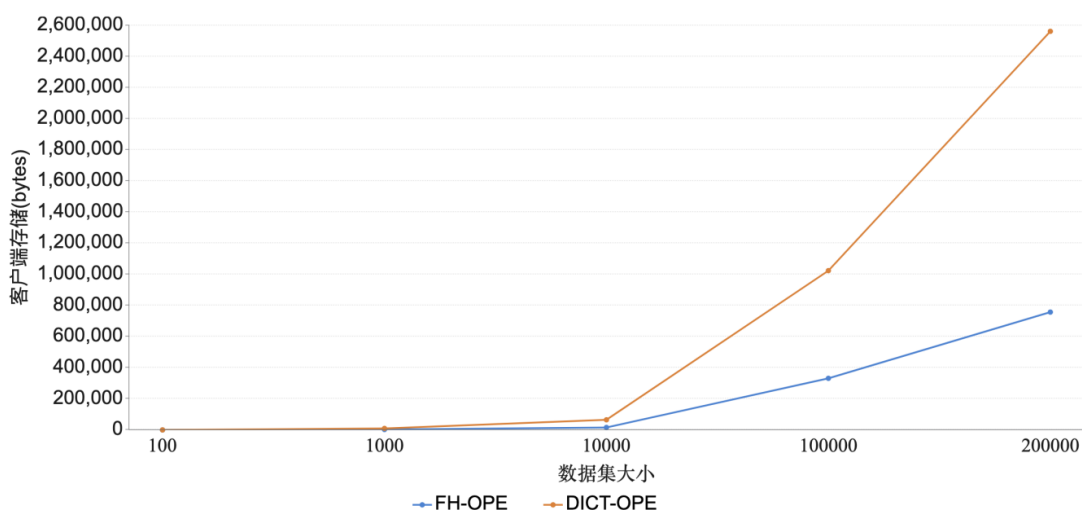


图3-9 利用不同OPE加密算法加密“感染人数”数据集的客户端存储差异

(2) “年份”数据集客户端存储测试

针对该加密算法，实验将评估本文提出的 OPE 算法（FH-OPE）和传统的 Kerschbaum 提出的 OPE 算法的客户端存储，且选定数据集大小为至多 100000 条数据量。由于差异过大，用图难以表示，故结果采用表格形式给出。其测试结果如表 3-1 所示。

数据集大小	客户端存储 （bytes）	
	FH-OPE 方案	DICT-OPE 方案
100	216	576
1000	468	5880
10000	540	60124
100000	660	64868

表3-1 利用不同OPE加密算法加密“年份”数据集的客户端存储

3.5 查询效率测试

(1) “感染人数”数据集加密性能测试

分别利用不同类型的 OPE 加密算法对明文域内所有明文进行一次加密。随后进行全部记录的查询，记录所需总时间，并以所得数据计算平均一次查询所需时间。其中图 3-10 给出了不同数据集下的平均查询时间。

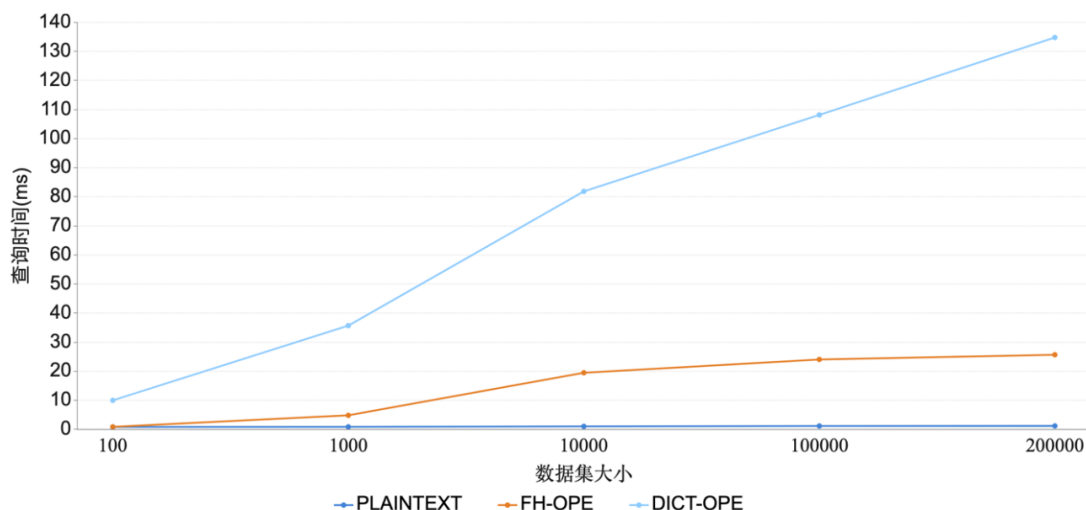


图3-10 利用不同OPE加密算法加密“感染人数”数据集的查询时间差异

(2) “年份”数据集加密性能测试

分别利用不同类型的 OPE 加密算法对明文域内所有明文进行一次加密。随后进行全部记录的查询，记录所需总时间，并以所得数据计算平均一次查询所需时间。其中图 3-11 给出了不同数据集下的平均查询时间。

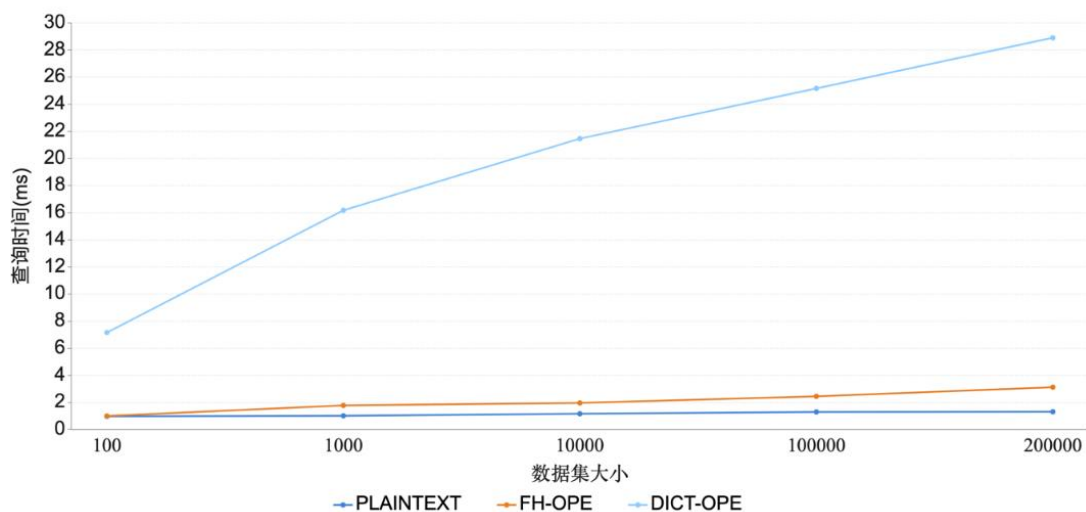


图3-11 利用不同OPE加密算法加密“年份”数据集的查询时间差异

3.6 测试数据与结果

针对我们的加密算法，实验将采用不同的数据集进行综合测试。同时，为了测试本算法在不同分布频率上的体现，实验将以“新冠”数据集的“感染人数”（下称“感染人数”）作为频率分布不明显的数据集，将以“中国移动”的“年龄”（下称“年龄”）和 DBLP 的“年份”（下称“年份”）作为频率分布十分明显的数据集。在本实验中，定义平均重复率 avg 来衡量总体明文和去重后

的明文数量的比值。表 3-2 给出了这些数据的信息。

数据集	去重后的明文数量	总体明文数量	<i>avg</i>
感染人数	98504	285307	2.8964
年龄	85	74645	878.176
年份	53	100000	1886.79

表3-2 数据集信息

结论1：对于本文提出的OPE加密算法，实验显示，即使不进行任何优化，本小组的FH-OPE算法也能比传统的Kerschbaum的DICT-OPE加密算法性能高效。如果进行了算法优化，最坏情况下，性能也能提高约8%，在一般情况下都能优化约20%的时间。因此本文提出的轻量级客户端存储的频率隐藏的OPE加密算法能够达到较高的加密效率。

结论2：加密方案实现了在密文状态下进行“增、删、改、查”四种操作，该方案只需要在客户端存储少量隐私数据和在服务器端建立密文索引，便使服务器端在与客户端正常交互的情况下对密文进行范围检索，也实现了加密的保序性特点，实现了安全性增强功能。

结论3：选定数据集大小为至多200000条数据量时前提下，当数据重复率较低时，FH-OPE算法所占用的存储空间远低于传统的Kerschbaum提出的DICT-OPE算法占用的存储空间。当数据量到达200000时只有DICT-OPE算法的1/4。当数据重复率较高时，FH-OPE算法的轻量级特性得到了充分的体现，数据量到达200000时占用的存储空间只有DICT-OPE算法1/100。

结论4：实验数据显示，FH-OPE算法查询数据重复率较低的数据时，查询性能也优于DICT-OPE算法，数据集达200000时也只有明文查询时间的3倍，DICT-OPE算法的1/6。当查询数据重复率较高的数据时，查询性能和明文查询几乎无差别，更是远远优于DICT-OPE算法，数据集达200000时，查询时间不足其1%。

4.应用前景

在实际的数据库使用场景中，现有的保序加密方案不够完善，很难单纯依靠这些加密算法构建出高效可靠的密态数据库系统：有的 OPE 算法由于频繁使用索引结构查询，导致额外性能开销增大，效率大幅度降低；有的 OPE 算法可能会泄露数据顺序，泄露数据信息过多，容易遭受频率攻击；有的 OPE 算法进行查询时 client-server 的交互为线性级别，通讯带宽过大。本轻量级客户端存储的频率隐藏的 OPE 加密方案是无索引结构的加密方案，与已有的保序加密方案相比，此方案效率较高，运行时间稳定，而且改进后也保证了不直接泄露明文分布信息，解决了数据加密领域中的大部分问题。首次通过编码机制来优化保序加密数据库索

引树调整带来的额外开销，与典型的 mOPE 等方案相比，所提出的编码优化机制可以减少大约 5-6 倍的 OPE 编码调整量；客户端存储量由记录条数相关降低到有效数值个数，极大降低了客户端存储量，对于年龄、地区、工资等以频率隐藏为主要安全目标的字段起到良好的实用效果；经过实验检验，在数据库查询性能方面，相较于 Kerschbaum 等经典的保序加密算法也有了较大的提升。

上述的性能提升，在密态数据库应用领域内具有较为深刻的意义。其一，加密性能的提升能够极大地提升算力效率；其二，加密开销降低有利于密态数据库系统的广泛部署；其三，密态数据库加密开销的降低，能够有效降低使用成本，提高用户体验等。

5. 结论

为了解决传统保序加密算法不能隐藏频率以及部分频率隐藏的保序加密算法性能开销过大的问题，本文提出了基于 B+编码树的保序加密算法，同时实现了轻量级客户端存储以及频率的隐藏，规避了过去算法存在的诸多问题和不足。

实验结果显示，本文提出的轻量级客户端存储的频率隐藏的 OPE 加密算法能够达到较高的加密效率，同时客户端存储代价也仅在 MB 级别，大大优于 DICT-OPE 的近 GB 级别的存储量。而在查询效率方面相比较明文检索，时间损失也不是很明显，而且时间的开销增长曲线较为缓慢。由此可见本加密算法具有较为理想的性能。

最后，本加密算法也具有较为广阔的应用前景，密态数据库系统中最为常用的返回检索便可依托于本加密算法实现，从而降低开销，提高效率。

参考文献

- [1] Oracle Inc. MySQL. <https://www.mysql.com/> [A/OL]. 2021-10.
- [2] PostgreSQL Global Development Group. PostgreSQL: The World's Most Advanced Open Source Relational Database. <https://www.postgresql.org/> [A/OL]. 2021-10.
- [3] GB/T 32907-2016 信息安全技术—SM4 分组密码算法[S].
- [4] Daemen, Joan, Vincent Rijmen. AES proposal: Rijndael. [J] 1999.
- [5] Jin Li, Zheli Liu, L-EncDB: A Lightweight Framework for Privacy-Preserving Data Queries in Cloud Computing, Knowledge-Based Systems, 2015, SCI, EI, 第一标注.
- [6] Zheli Liu, Jian Weng, Jin Li, Cloud-based Electronic Health Record System Supporting Fuzzy Keyword Search, Soft Computing, 2015.5.12, 1(1): 1~13, SCI, 第一标注.
- [7] Popa R A, Li F H, Zeldovich N. An ideal-security protocol for order-preserving encoding[C]//2013 IEEE Symposium on Security and Privacy. IEEE, 2013: 463-477.
- [8] Popa R A, Redfield C M S, Zeldovich N, et al. CryptDB: Protecting Confidentiality with Encrypted Query Processing[C]//Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011.
- [9] Kerschbaum F. Frequency-hiding order-preserving encryption[C]//Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. 2015: 656-667.
- [10] 李经纬, 贾春福, 刘哲理, 李进. 可搜索加密技术研究综述, 软件学报, 2015.1.1, 26: 109~128, EI, 第一标注.