Algorithm 1: Insert_stage1

Input: Item e
Output: Report whether the insertion of element e in stage1 was successful

```
for entry ∈ L0[hash0(e)] % bucket_num do
    if entry.id == null || entry.id == id then
        entry.id = e.id;
        return true;
    end
endfor
return false;
```

Algorithm 2: Insert_stage2

Input: Item e
Output: Report whether the insertion of element e in stage2 was successful

```
v1 = min_{1≤i≤d1} L1[i][hash1_i(e)].cnt;
if v1 < threshold_{L1} then
    for i from 1 → d1 do
        b_1 ← L1[i][hash1_i(e)];
        if b_1.cnt == v1 && b_1.flag == On then
            b_1.cnt ← b_1.cnt + 1;
            b_1.flag ← Off;
        end
    endfor
    return true;
end
                                              ▷ L1 reach threshold
v2 = min_{1≤i≤d2} L2[i][hash2_i(e)].cnt;
if v2 < threshold_{L2} then
    for i from 1 → d2 do
        b_2 ← L2[i][hash2_i(e)];
        if b_2.cnt == v2 && b_2.flag == On then
            b_2.cnt ← b_2.cnt + 1;
            b_2.flag ← Off;
        end
    endfor
    return true;
end
                                              ▷ L2 reach threshold
return false
```

Algorithm 3: Insert_stage3

Input: Item e

```
1  h3 ← hash3(e) % (length * (length − 1));
2  h3_1 ← h3 % length; h3_2 ← h3 ÷ (length − 1);
3  replace = null;
4  for i ∈ {1, 2} do
5  |  b3 ← L3[i][h3_i];
6  |  for entry ∈ b3 do
7  |  |  if
   |  |   entry.item == null || entry.item == e && entry.flag == On
   |  |   then
8  |  |  |  entry.item ← e;
9  |  |  |  entry.flag ← Off;
10 |  |  |  entry.cnt ← entry.cnt + 1;
11 |  |  |  return;
12 |  |  end
13 |  |  else
14 |  |  |  if replace == null || entry.cnt < replace.cnt then
15 |  |  |  |  replace ← entry;
16 |  |  |  end
17 |  |  end
18 |  endfor
19 endfor
20 if h3 % (replace.cnt + 1) == 0 then
21 |  replace.item ← e;
22 |  if replace.flag == On then
23 |  |  replace.cnt ← replace.cnt + 1;
24 |  |  replace.flag ← Off;
25 |  end
26 end
```

Algorithm 4: flush_L0

```
1        ▷ when new window arrives, flush items in L0 to the later stage
2  for entry ∈ L0 do
3  |  if entry ≠ null then
4  |  |  if !Insert_stage2(e) then
5  |  |  |  Insert_stage3(e);
6  |  |  end
7  |  end
8  endfor
```

Algorithm 5: Insert

Input: Item e

```
if Insert_stage1(e) then
    return;
end
if Insert_stage2(e) then
    return;
end
Insert_stage3(e);
return;
```

Algorithm 6: Query

Input: Item e
Output: The persistence of e

```
ret ← 0;
v1 ← min_{1≤i≤d1}(L1[i][hash1_i(e)]);
if v1 < threshold_{L1} then
    return v1;
end
ret ← ret + v1;
v2 = min_{1≤i≤d2}(L2[i][hash2_i(e)]);
if v2 < threshold_{L2} then
    return ret + v2;
end
ret ← ret + v2;
hash_num ← hash3(e) % length * (length − 1);
bucket_pos_1 ← hash_num % length;
bucket_pos_2 ← hash_num ÷ (length − 1);
for i ∈ {1, 2} do
    bucket ← L3[i][bucket_pos_i];
    for entry ∈ bucket do
        if entry.item == e then
            ret ← ret + entry.count;
        end
    endfor
endfor
return ret;
```

Algorithm 7: Scanning a bucket with 16 cells through the SIMD instructions

Input: The incoming item e and the start address p of the bucket
Output: Return the index of the matched key or -1

```
1  __m128i item = _mm_set1_ep132(e);
2  __m128i * keys_p = (__m128i *) p;
3  __m128i a_comp = _mm_cmpeq_epi32(item, keys_p[0]);
4  __m128i b_comp = _mm_cmpeq_epi32(item, keys_p[1]);
5  __m128i c_comp = _mm_cmpeq_epi32(item, keys_p[2]);
6  __m128i d_comp = _mm_cmpeq_epi32(item, keys_p[3]);
7  a_comp = _mm_packs_epi32(a_comp, b_comp);
8  c_comp = _mm_packs_epi32(c_comp, d_comp);
9  a_comp = _mm_packs_epi32(a_comp, c_comp);
10 matched = _mm_movemask_epi8(a_comp);
11 if match ≠ 0 then
12 |  return TZCNT(matched);
13 end
14 else
15 |  return −1;
16 end
```