# Fast Intersection of Sorted Lists Using SSE Instructions

Intersection of sorted lists is a cornerstone operation in many applications including search engines and databases because indexes are often implemented using different types of sorted structures. At GridDynamics, we recently worked on a custom database for realtime web analytics where fast intersection of very large lists of IDs was a must for good performance. From a functional point of view, we needed mainly a standard boolean query processing, so it was possible to use Solr/Lucene as a platform. However, it was interesting to evaluate performance of alternative approaches. In this article I describe several useful techniques that are based on SSE instructions and provide results of performance testing for Lucene, Java, and C implementations. I'd like to mention that in this study we were focused on a general case when selectivity of the intersection is low or unknown and optimization techniques like skip list are not necessarily beneficial.

## Scalar Intersection

Our starting point is a simple element-by-element intersection algorithm (also known as Zipper). Its implementation in C is shown below and do not require lengthy explanations:

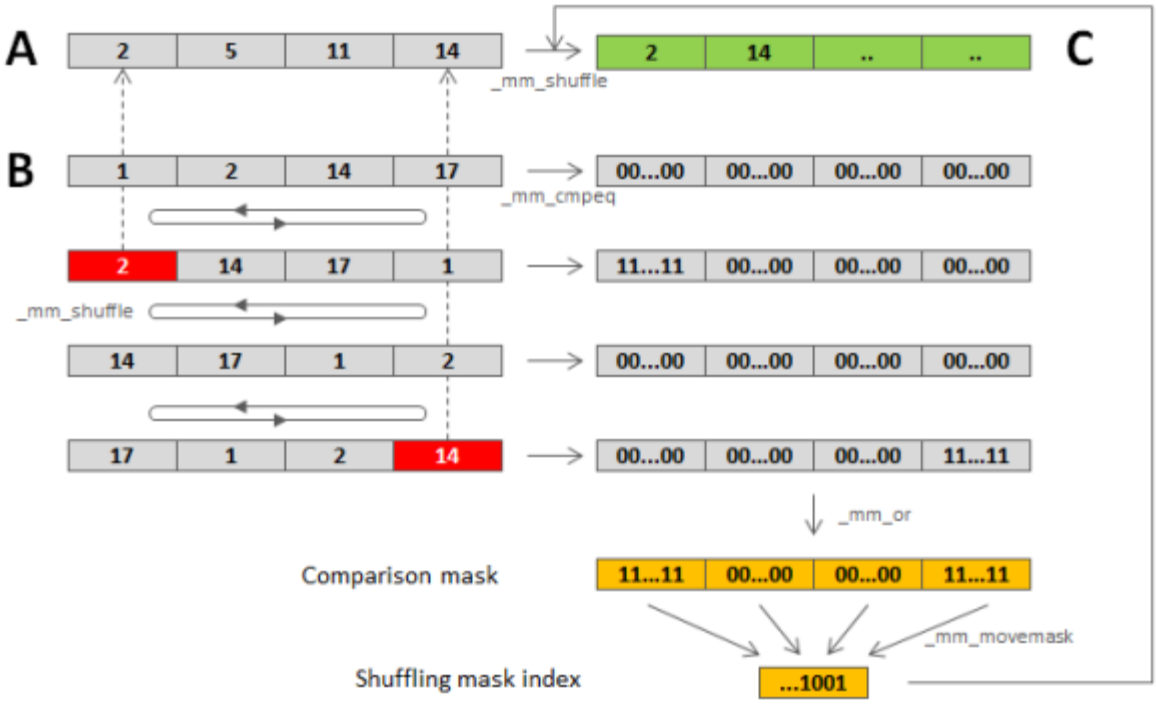
```
1   #define int32 unsigned int
2
3   // A, B - operands, sorted arrays
4   // s_a, s_b - sizes of A and B
5   // C - result buffer
6   // return size of the result C
7   size_t intersect_scalar(int32 *A, int32 *B, size_t s_a, size_t s_b,
8   int32 *C) {
9       size_t i_a = 0, i_b = 0;
10      size_t counter = 0;
11
12      while(i_a < s_a && i_b < s_b) {
13          if(A[i_a] < B[i_b]) {
14              i_a++;
15          } else if(B[i_b] < A[i_a]) {
16              i_b++;
17          } else {
18              C[counter++] = A[i_a];
```

```
19              i_a++; i_b++;
20          }
21      }
22      return counter;
    }
```

Performance of this procedure both in C and Java will be evaluated in the last section. I believe that it is possible to improve this approach using a branchless implementation, but I had no chance to try it out.

### Vectorized Intersection

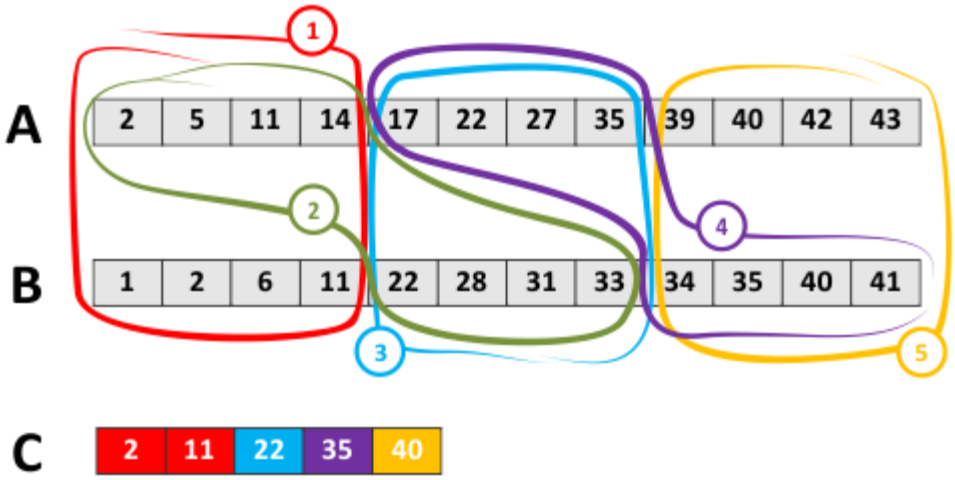
It is intuitively clear that performance of intersection may be improved by processing of multiple elements at once using SIMD instructions. Let us start with the following question: how to find and extract common elements in two short sorted arrays (let's call them segments). SSE instruction set allow one to do a pairwise comparison of two segments of four 32-bit integers each using one instruction (_mm_cmpeq intrinsic) that produces a bit mask that highlights positions of equal elements. If one has two 4-element registers, A and B, it is possible to obtain a mask of common elements comparing A with different cyclic shifts of B (the left part of the figure below) and OR-ing the masks produced by each comparison (the right part of the figure):




The resulting *comparison mask* highlights the required elements in the segment A. This 128-bit mask can be transformed to a 4-bit value (*shuffling mask index*) using _mm_movemask intrinsic. When this short mask of common elements

is obtained, we have to efficiently copy out common elements. This can be done by shuffling of the original elements according to the shuffling mask that can be looked up in the precomputed dictionary using the *shuffling mask index* (i.e. each of 16 possible 4-bit shuffling mask indexes is mapped to some permutation). All common elements should be placed to the beginning of the register, in this case register can be copied in one shot to the output buffer C as it shown in the figure above.

The described technique gives us a building block that can be used for intersection of long sorted lists. This process is somehow similar to the scalar intersection:




In this example, during the first cycle we compare first 4-element segments (highlighted in red) and copy out common elements (2 and 11). Similarly to the scalar intersection algorithm, we can move forward the pointer for the list B because the tail element of the compared segments is smaller in B (11 vs 14). At the second cycle (in green) we compare the first segment of A with the second segment of B, intersection is empty, and we move pointer for A. And so on. In this example, we need 5 comparisons to process two lists of 12 elements each.

The complete implementation of the described techniques is shown below:

```
1    static __m128i shuffle_mask[16]; // precomputed dictionary
2
3    size_t intersect_vector(int32 *A, int32 *B, size_t s_a, size_t s_b, int32 *C)
4    {
5        size_t count = 0;
6        size_t i_a = 0, i_b = 0;
7
8        // trim lengths to be a multiple of 4
9        size_t st_a = (s_a / 4) * 4;
```

```
10      size_t st_b = (s_b / 4) * 4;
11
12      while(i_a < s_a && i_b < s_b) {
13          //[ load segments of four 32-bit elements
14          __m128i v_a = _mm_load_si128((__m128i*)&A[i_a]);
15          __m128i v_b = _mm_load_si128((__m128i*)&B[i_b]);
16          //]
17
18          //[ move pointers
19          int32 a_max = _mm_extract_epi32(v_a, 3);
20          int32 b_max = _mm_extract_epi32(v_b, 3);
21          i_a += (a_max <= b_max) * 4;
22          i_b += (a_max >= b_max) * 4;
23          //]
24
25          //[ compute mask of common elements
26          int32 cyclic_shift = _MM_SHUFFLE(0,3,2,1);
27          __m128i cmp_mask1 = _mm_cmpeq_epi32(v_a, v_b);    // pairwise
28  comparison
29          v_b = _mm_shuffle_epi32(v_b, cyclic_shift);       // shuffling
30          __m128i cmp_mask2 = _mm_cmpeq_epi32(v_a, v_b);    // again...
31          v_b = _mm_shuffle_epi32(v_b, cyclic_shift);
32          __m128i  cmp_mask3  =  _mm_cmpeq_epi32(v_a,  v_b);     //  and
33  again...
34          v_b = _mm_shuffle_epi32(v_b, cyclic_shift);
35          __m128i  cmp_mask4  =  _mm_cmpeq_epi32(v_a,  v_b);     //  and
36  again.
37          __m128i cmp_mask = _mm_or_si128(
38                  _mm_or_si128(cmp_mask1, cmp_mask2),
39                  _mm_or_si128(cmp_mask3, cmp_mask4)
40          ); // OR-ing of comparison masks
41          // convert the 128-bit mask to the 4-bit mask
42          int32 mask = _mm_movemask_ps((__m128)cmp_mask);
43          //]
44
45          //[ copy out common elements
46          __m128i p = _mm_shuffle_epi8(v_a, shuffle_mask[mask]);
47          _mm_storeu_si128((__m128i*)&C[count], p);
```

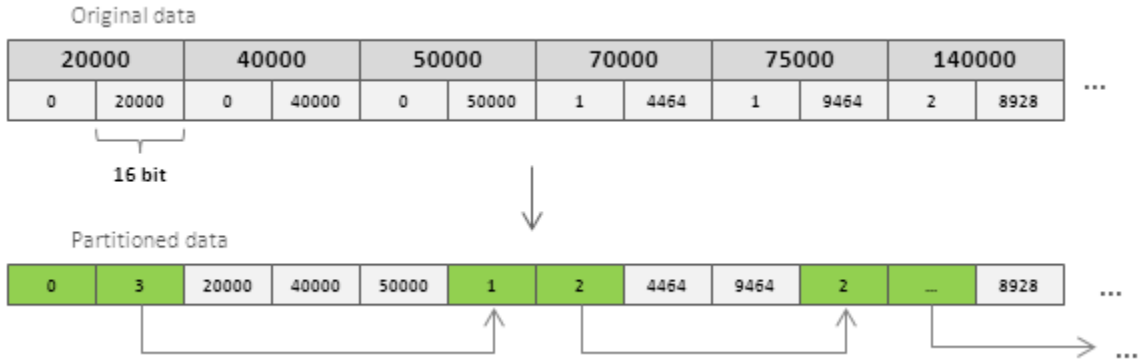
```
48          count += _mm_popcnt_u32(mask); // a number of elements is a
49  weight of the mask
50          //]
51      }
52
        // intersect the tail using scalar intersection
        ...

        return count;
    }
```

The described implementation uses the *shuffle_mask* dictionary to map the mask of common elements to the shuffling parameter. Building of this dictionary is straightforward (each bit in the mask corresponds to 4 bytes in the register):

```
1   // a simple implementation, we don't care about performance here
2   void prepare_shuffling_dictionary() {
3       for(int i = 0; i < 16; i++) {
4           int counter = 0;
5           char permutation[16];
6           memset(permutation, 0xFF, sizeof(permutation));
7           for(char b = 0; b < 4; b++) {
8               if(getBit(i, b)) {
9                   permutation[counter++] = 4*b;
10                  permutation[counter++] = 4*b + 1;
11                  permutation[counter++] = 4*b + 2;
12                  permutation[counter++] = 4*b + 3;
13              }
14          }
15          __m128i mask = _mm_loadu_si128((const
16  __m128i*)permutation);
17          shuffle_mask[i] = mask;
18      }
19  }
20
21  int getBit(int value, int position) {
22      return ( ( value & (1 << position) ) >> position);
    }
```

SSE 4.2 instruction set offers PCMPESTRM instruction that allows one to compare two segments of eight 16-bit values each and obtain a bit mask that highlights common elements. This sounds like an extremely efficient approach for intersection of sorted lists, but in its basic form this approach is limited by 16-bit values in the lists. This is not the case for many applications, so a workaround was recently suggested by Benjamin Schedel et al. in this article. The main idea is to store indexes in the partitioned format, where elements with the same most significant bits are grouped together. This approach also has limited applicability because each partition should contain a sufficient number of elements, i.e. it works well in case or very large lists or favorable distribution of the values.

Each partition has a header that includes a prefix which represents most significant bits that are common for all elements in the partition and the number of elements in the partition. The following figure illustrates the partitioning process:




The partitioning procedure that coverts 32-bit values into 16-bit values is shown in the code snippet below:

```
1    // A - sorted array
2    // s_a - size of A
3    // R - partitioned sorted array
4    size_t partition(int32 *A, size_t s_a, int16 *R) {
5        int16 high = 0;
6        size_t partition_length = 0;
7        size_t partition_size_position = 1;
8        size_t counter = 0;
9        for(size_t p = 0; p < s_a; p++) {
10           int16 chigh = _high16(A[p]); // upper dword
11           int16 clow = _low16(A[p]);   // lower dword
12           if(chigh == high && p != 0) { // add element to the current
13   partition
14               R[counter++] = clow;
```

```
15              partition_length++;
16          } else { // start new partition
17              R[counter++] = chigh; // partition prefix
18              R[counter++] = 0;     // reserve place for partition size
19              R[counter++] = clow;  // write the first element
20              R[partition_size_position] = partition_length;
21              partition_length = 1; // reset counters
22              partition_size_position = counter - 2;
23              high = chigh;
24          }
25      }
26      R[partition_size_position] = partition_length;
27
28      return counter;
    }
```

A pair of partitions can be intersected using the following procedure that computes a mask of common elements using _mm_cmpestrm intrinsic and then shuffles these elements similarly to the vectorized intersection procedure what was described in the previous section.

```
1  size_t intersect_vector16(int16 *A, int16 *B, size_t s_a, size_t s_b, int16
2  *C) {
3      size_t count = 0;
4      size_t i_a = 0, i_b = 0;
5
6      size_t st_a = (s_a / 8) * 8;
7      size_t st_b = (s_b / 8) * 8;
8
9      while(i_a < st_a && i_b < st_b) {
10         __m128i v_a = _mm_loadu_si128((__m128i*)&A[i_a]);
11         __m128i v_b = _mm_loadu_si128((__m128i*)&B[i_b]);
12
13         __m128i res_v = _mm_cmpestrm(v_b, 8, v_a, 8,
14                 _SIDD_UWORD_OPS|_SIDD_CMP_EQUAL_ANY|_SIDD_BIT_MASK);
15         int r = _mm_extract_epi32(res_v, 0);
16         __m128i p = _mm_shuffle_epi8(v_a, shuffle_mask16[r]);
17         _mm_storeu_si128((__m128i*)&C[count], p);
18         count += _mm_popcnt_u32(r);
19
```

```
20      int16 a_max = _mm_extract_epi16(v_a, 7);
21      int16 b_max = _mm_extract_epi16(v_b, 7);
22      i_a += (a_max <= b_max) * 4;
23      i_b += (a_max >= b_max) * 4;
24   }
25
26   // intersect the tail using scalar intersection
27   ...
28
29   return count;
  }
```

The whole intersection algorithm looks similarly to the scalar intersection. It receives two partitioned operands, iterates over headers of partitions and calls intersection of particular partitions if their prefixes match:

```
    // A, B – partitioned operands
1   size_t intersect_partitioned(int16 *A, int16 *B, size_t s_a, size_t s_b,
2   int16 *C) {
3       size_t i_a = 0, i_b = 0;
4       size_t counter = 0;
5
6       while(i_a < s_a && i_b < s_b) {
7           if(A[i_a] < B[i_b]) {
8               i_a += A[i_a + 1] + 2;
9           } else if(B[i_b] < A[i_a]) {
10              i_b += B[i_b + 1] + 2;
11          } else {
12              C[counter++] = A[i_a]; // write partition prefix
13              int16 partition_size = intersect_vector16(&A[i_a + 2], &B[i_b +
14  2],
15                      A[i_a + 1], B[i_b + 1], &C[counter + 1]);
16              C[counter++] = partition_size; // write partition size
17              counter += partition_size;
18              i_a += A[i_a + 1] + 2;
19              i_b += B[i_b + 1] + 2;
20          }
21      }
22      return counter;
    }
```

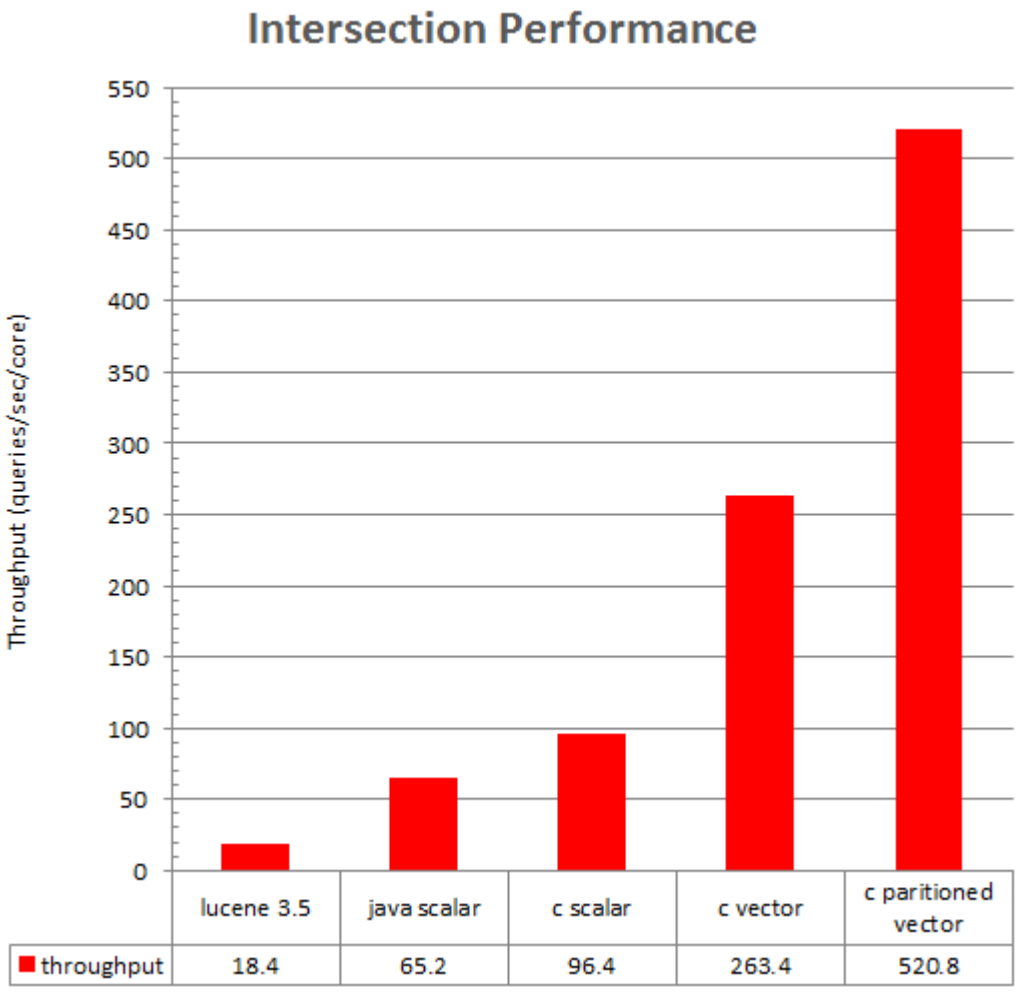
The output of this procedure is also a partitioned vector that can be used in further operations.

## Performance Evaluation

Performance of the described techniques was evaluated for intersection of sorted lists of size 1 million elements, with average intersection selectivity about 30%. All evaluated methods excepts partitioned vectorized intersection do not require specific properties of the values in the lists. For partitioned vectorized intersection values were selected from range [0, 3M] to provide relatively large partitions.

In case of Lucene, a corpus of documents with two fields was generated to provide the mentioned index sizes and selectivity; RAMDirectory was used. Intersection was done using standard Boolean query with top hits limited by 1 to prevent generation of large result set. Of course, this not a fair comparison because Lucene is much more than a list intersector, but it is still interesting to try it out.

Performance testing was done on the ordinary Linux desktop with 2.8GHz cores. JDK 1.6 and gcc 4.5.2 (with -O3 option) were used.






Source: http://highlyscalable.wordpress.com/2012/06/05/fast-intersection-sorted-lists-sse/