W4112 Database Systems Implementation, Spring 2016 Project 2, due April 11, 2016 (part 1) and April 28, 2016 (parts 2 and 3)

In this project, you will be asked to implement an advanced kind of traversal method for in-memory search trees. For analytic datasets that don't need to support updates, we can optimize the tree data structure to eliminate pointers. Parent-child relationships are implicit in the arrangement of the keys within nodes.

Unlike what we've covered in class so far, we're not aiming to optimize I/O. Instead, we're trying to optimize performance by reducing CPU-related latencies and by ultizing SIMD capabilities of modern CPUs.

The details of the advanced method you're going to implement are described in Section 3.5 of the reference document at `http://www.cs.columbia.edu/~orestis/sigmod14I.pdf`. You will implement the horizontal approach, using variations of the first code fragment from Section 3.5.1, and the code fragment in Section 3.5.2. You will use the first method shown in Section 3.5.1 to search a node and extend it for multiple tree levels as explained in Section 3.5.2.

There are three stages to the project, outlined below. Your code must be implemented in `C`. `C++` is not permitted.

1 Part 1 (40 points). Due 4/11/2016

You will implement code in `C` to build a pointer-free tree structure as described in the reference document, with an array of keys for each tree level. The tree contains keys only, and the result of a search is the identity of the range to which the search key belongs. So if there are n keys, there will be $n + 1$ ranges numbered $0, \dots, n$. (To do a real index traversal we would use the range identifier as an offset into an array of record-identifiers, but for this project we'll omit that step, and ignore record-identifiers altogether.)

To build this table, you'll need to randomly generate a set of unique 32-bit integer keys, sort them, and store them as outlined in the reference document. A parameter to your method will be the fanout at each level specified as a list of integers on the command line, You may assume that the fanout can be any integer between 2 and 17 inclusive, but we'll be particularly interested in fanouts of 5, 9 and 17 as discussed in the reference document. If there are too many build keys for the given fanout, then you should report an error. If there are too few keys, so that when you build the tree the root node is empty (and a shallower tree would suffice), then you should also report an error. If a tree node is not completely filled, you should pad it with extra keys containing the maximum representable number.

For example, suppose we are trying to build a 2-level tree with a fanout of 4 (i.e., 3 keys per node) at each level. Suppose there are nine keys to put in the tree: 10, 20, 30, 40, 50, 60, 70, 80, 90. Start by filling the leaf level node with 3 keys, 10, 20, 30. The next key 40 then goes in the parent node. The next 3 keys 50, 60, 70 go in the next leaf node. Key 80 goes in the parent node. Key 90 goes in the leaf node. This last leaf node is partially full, so it is padded with two instances of MAXINT. The parent node is also partially full, so it is padded with one instance of MAXINT. So the final leaf level array is [10,20,30,50,60,70,90,MAXINT,MAXINT]. Note that the array has length 9, not 12. (Had there been more keys we may have needed to allocate 12 slots in the array rather than 9.) The root level array is [40,80,MAXINT]. You will need to figure out the math for how to distribute keys to arrays in the general case. In a three-level tree, the grandparent node gets populated after the parent node fills, etc. Make sure the arrays are aligned at 16 byte boundaries by allocating them using the `posix_memalign` function. (This impacts performance because SSE aligned reads are faster than unaligned reads.)

Once you have built the tree, for this stage of the project you will simply do a binary search within each node to locate the appropriate child node. A set of probes is randomly generated, and the output is the corresponding sequence of range identifiers. Your code should be invoked as:

```
build K P 9 5 9
```

to build a 3-level 9-5-9 tree, or

```
build K P 9 5 5 9
```

to build a 4-level 9-5-5-9 tree, etc. K is the number of keys used to build the tree, and P is the number of probes to perform. We will provide code to generate random sets of integers.

You should implement the method in three distinct phases to facilitate timing measurements. In phase 1, the index should be built, and the probes loaded into an array of integers. In phase 2, the index is probed using binary search for each node and the range identifier of the match is appended to an array of output values. In phase 3, the output value array is written to stdout.

2 Part 2 (40 points). Due 4/28/2016

In this part of the project, you will re-implement the probe routine in `C` using Intel's SSE instruction set. Your code should more closely match the code fragments from the reference document. You can find a guide to SSE instructions (also known as "intrinsics") at `http://software.intel.com/sites/landingpage/IntrinsicsGuide/`. These instructions should be recognized by the gcc compiler as long as you use the `-msse4.2` compilation flag and include the following header files:

```
<xmmintrin.h> SSE
<emmintrin.h> SSE2
<pmmintrin.h> SSE3
<tmmintrin.h> SSSE3
<smmintrin.h> SSE4.1
<nmmintrin.h> SSE4.2
<ammintrin.h> SSE4A
```

Your program should be invokable using the same interface as before, but you only need to support fanouts of 5, 9, and 17 (i.e., 4, 8, 16 keys, respectively). The output from parts 1 and 2 should be identical.

While your method should be fully general, you should also hardcode one particular tree structure, namely the one with fanouts of 9, 5, and 9 on successive levels (see the reference document). Hardcoding means that certain overheads such as dereferencing function pointers or checking the fanout at each level can be avoided during probes. The hardcoded probe function for a given key should have no `if` or `while` or `for` statements. Almost all of the statements in the probe routine should use SSE intrinsics.

The hardcoded version should incorporate two additional optimizations. First, the root node of the index should be explicitly loaded into register variables so that it is not re-read from the array for each search. Second, probes should be done four at a time. To load and broadcast the 4 keys from the input to register variables, you might use:

```
__m128i k = _mm_load_si128((__m128i*) &probe_keys[i]);
register __m128i k1 = _mm_shuffle_epi32(k, _MM_SHUFFLE(0,0,0,0));
register __m128i k2 = _mm_shuffle_epi32(k, _MM_SHUFFLE(1,1,1,1));
register __m128i k3 = _mm_shuffle_epi32(k, _MM_SHUFFLE(2,2,2,2));
register __m128i k4 = _mm_shuffle_epi32(k, _MM_SHUFFLE(3,3,3,3));
```

These 4 keys must be processed in an unrolled fashion: access level 1 for all four, then access level 2 for all four etc. This optimization allows work to happen for one probe while another has a data access stall, for example.

Again, you should implement the method in three distinct phases to facilitate timing measurements. In phase 1, the index should be built, and the probes loaded into an array of integers. In phase 2, the index is probed using the more advanced method and the range identifier of the match is written to an array of output values. In phase 3, the output value array is written to stdout.

3 Part 3 (20 points). Due 4/28/2016

In this part of the project, you will compare the time performance of the two approaches (part 1 and part 2) by measuring the time taken in phase 2 of each implementation. You will need to evaluate the two methods on the following three tree structures:

- 9-5-9
- 17-17

- 9-5-5-9

For the “9-5-9” tree structure, include both the hardcoded probe implementation, and the non-hardcoded probe implementation in the evaluation. Make sure that the probe data used is large enough so that the overheads of setting up the experiment are small relative to the probing time, but not so large that the experiment takes too long to complete. You should submit a brief report describing the performance differences observed for your chosen data. The report should include, for each dataset, one chart plus one or two paragraphs of explanation (including the number of build and probe keys used). The report should fully explain how your experiments were performed, including:

1. Which machines were used?
2. What are the machine’s characteristics (clock frequency, amount of RAM, etc.)?
3. Were the machines idle apart from the experiment?
4. What compiler did you use to compile the code? What compiler options were provided?
5. How did you instrument the code to measure the time taken just for phase 2?

Administrative Notes

Projects are to be done in teams of two. Unless you tell the TAs otherwise, we’ll assume you’ll keep the same teams as in project 1. If for some reason you are working alone, you are expected to do parts 1 and 2, but not part 3 of the project. However, such cases will be limited to situations where the TAs are unable to find a partner for a student, or where a project partner drops the class unexpectedly.

The deadline for part 1 is 4/11, and the deadline for parts 2 and 3 is 4/28. Submit using your dropbox on courseworks2 your code, a comprehensive README file, a Makefile, and the brief report. As usual, the code should be well-documented, and clearly organized.