Database System Implementation

Project 2

**Teammates**: ChaozhongLian( cl3190 ), Jinxi Zhao ( jz2540 )

**List of Files**:

* SplashTable.java //java source code
* probe.c //c source code
* splash //shell script for running part 1
* Makefile //Makefile to compile java and c codes
* ReadMe // The file you are reading

**How to run**:

1. Type "make"

This compile the "SplashTable.java" and "probe.c", and generate "SplashTable.class" and "probe" files.

2. Type "./splash B R S h inputfiledumpfile<probefile>resultfile"

This run the java program with given B R S h and take standard input from probefile, then generate the resultfile

3. Type "./probe dumpfile<probefile>resultfile"

This run the c program, taking dumpfile generated from part 1 as argument, taking probefile as standard input, and output to resultfile

**Internal Design**

**Part 1:**

The <key, payload> for the splash table is stored as a Pair class object, a bucket stores a LinkedList of Pairs, thus the whole splash table is represented as an ArrayList of LinkedList of Pairs. The splash table can be initialized with B, R, S, h values.

When inserting a <key, payload> pair, we first check if the key is in the table already, if so, replace the old payload with the new payload. If it is not in the table, we check if the table is full, if so, insertion fails; if not full, we try to insert it into the table. First of all, we hash the key values using all h hash functions and generate h hashcodes, which means the candidate bucket numbers. The bucket to be inserted is randomly chosen from the candidate buckets with max free slots. If all candidate buckets are full, re-insertion process is activated. The first pair of the chosen bucket is removed, the new pair is inserted to the end of the bucket. Then the removed pair needs to be re-inserted into the table, which goes through the same insertion process. If insertion cannot succeed until it reaches the maximum re-insertion number, insertion fails and the program stops.

When probing a key, we need to use h hash functions to get h candidate buckets, and then search all candidate buckets to check if the key exists. If it exists, print the <key, payload> pair; if not, ignore the key and continue to probe next key. To avoid branch misprediction, we uses “and, or” operators in stead of “if” statement. Please see source code for detail step-by-step explaination.

Note that we use multiplicative hashing described in Knuth's "The Art of Computer Programming" volume 3.For the special case of a 32 bit key K, and a 32-bit odd multiplier M, one performs a 32-bit multiplication M\*K.  This 32-bit result R drops the high-order 32 bits that would be visible if we did a full 64-bit multiplication.  So R is M\*K mod 2^32.  The high-order bits of R are used for hashing, so that if we need 10 hash bits we can shift R right by 22 bits.Also note that in our implementation, M value is generated using Random.nextInt(), which is ranged from -2^31 to 2^31-1.

**Part 2:**

In probe.c, we store the keys and their corresponding payloads in two separate arrays, these two arrays are just one dimension array, upon receiving the probing value, we will calculate the bucket number and so reaching the corresponding value.

The main use of SSE instructions is in the probe function, the algorithm is as described in the paper. We first construct a pack of 4 target value using \_mm\_set1\_epi32 (SIMD COPY), then this targetPack is multiply with the 2 hash multiplication numbers using \_mm\_mullo\_epi32 (SIMD MULT), the result is then once again multiply with the size of the bucket(value of B), and so the "index" to the two buckets we needed are generated.

Using these two index, we fetch the two buckets of keys, we use \_mm\_cmpeq\_epi32 (SIMD cmp-eq) to compare the keys to the targetPack, it there is a match, then the corresponding field will be all 1s, then we use \_mm\_and\_si128 (SIMD and) to do the and operation with their corresponding payloads, if there is a match, now the two packs will not all be 0s, the correct payload will be in there. So the last step we do is to use \_mm\_or\_si128 (SIMD or) to generate one pack with the correct payload in it. Because there is no SIMD or-across available, so we extract the four values one by one, and return the "or" of these four value. If there is a find, the return value will not be zero, if no find, then it should be zero.