**Deep Bodra**

**5801 1841**

**DBI Week 8(2/24 - 2/28)**

**Detailed explanation of the topics covered in class**

**2/25/2020**

1. Hash tables
   1. Array of buckets
   2. Each array index has buckets connected in the form of a linked list
   3. The records in this linked list of buckets are similar in the sense that a hash of the subset of attributes is same
2. MCT and tmalloc [secret to speeding up your programs]
   1. The default memory allocator that comes with C++ is not so fast
   2. Whether you allocate small amount of memory or a large amount of memory the difference in allocation time is not that significant
   3. So, better use custom allocator like MCT
   4. tmalloc is also a better choice
   5. Tmalloc makes use of thread global variables
3. Hash functions
   1. [Which is the best hash function?](https://softwareengineering.stackexchange.com/questions/49550/which-hashing-algorithm-is-best-for-uniqueness-and-speed) [link to the stackexchange question discussed in class]
      1. There are tons of hash functions out there each designed for a different reason
      2. Murmur II is pretty decent as it is fairly good in terms of randomness, uniqueness and speed (not the best but a good trade off amongst all aspects)
4. Size of hash table?
   1. +, -, \*, are cheap operations requiring just one clock cycle
   2. / takes about 30-40 cycles
   3. An alternative to x/2 is right shift of x by 1
   4. So we maintain hash table size that is some power of 2 or sum of powers of 2
   5. For hash tables we require a large prime number
   6. So, we need a large prime number that is closest to a power of 2
   7. The Mersenne primes are a set of primes one less than a power of 2
   8. Typically, we use (261 - 1)
5. Congruent Linear Hash
   1. (ax+b)modp
   2. p is prime
   3. a ≈ p
   4. Mod operation involves division which is expensive
   5. So use Mersenne primes

**2/27/2020**

1. Complexity of a mod N
   1. O(No of 1 bits in N)
2. TLB and large hashes
   1. TLB has about 1024 entries
   2. Assuming 4KB pages, about 4MB data
   3. For large hashes, it will be a disaster as a result of misses
   4. Solution: Large pages 2MB
3. Linear Hashing
   1. It is fantastic for large memory access
   2. It minimizes random memory access
   3. We have array of slots instead of array of buckets
   4. Each slot has 64b meta data and 64b data
   5. The access is done as follows: Random access followed by a sequential access
   6. Because of the sequential access, we benefit from prefetching
   7. Just 10 bits for linked list of slots. But we have sophisticated additional mechanism incase of overflows
4. Semaphore granularity and performance
   1. Intuitively, we might think that having semaphores for the smallest unit possible (say a bucket) is best. But fine grain semaphores result in bad performance
   2. Coarse grain semaphores give remarkable performance
5. 2 Stage hash



* 1. Stage 1: Compute hash
     1. Each thread generates a medium size hash by processing a chunk of size 2M tuples
  2. Stage 2: Merge hashes
     1. Hashes computed by threads are merged into one hash
  3. Two threads shouldn’t hash at the same time
  4. This method is still bad and doesn’t scale for more than 4 cores as times is wasted in getting the lock for the big hash (merged hash space)

1. Segmented hash
   1. The former is bad because time gets wasted in obtaining lock for the large merge hash (hash on the left in the diagram)
   2. So, we instead split the hash into segments and threads work on separate segments
   3. This way they can work in parallel and this can be scaled for any number of cores
   4. For allocating threads to segments, BitSet is used
      1. For 128 cores we have an array of 128 bits
      2. Processor i turns bit i from 0 to 1 to assign segment i to a thread

--THANK YOU--