Lecture 08: Hash Tables

15-445/645 Database Systems (Fall 2017)
Carnegie Mellon University
Prof. Andy Pavlo

Data Structures

- Can be used for internal meta-data about different components.
- Can be used as the base storage for tuples in the database
- Can be used as a temporary data structure to execute different relational operators in a query plan
- Lastly, we can use these data structures as indexes to speed up query processing
- Design Decisions
  1. Data organization: How we layout memory and what information to store inside the data structure
  2. Concurrency: How to enable multiple threads to access the data structure without causing problems

Hash Table

- **A hash table implements an associative array abstract data type that maps keys to values**
- It uses a **hash function** to compute an index into an array of buckets or slots
- Static hash Table
  - Giant array with one slot for every element. Mod the key by number of elements to find the offset in the array
  - For variable length elements, array holds pointers to elements
  - Problematic assumptions
    1. You know the number of elements ahead of time
    2. Each key is unique
    3. Perfect hash function (if key1 != key2 then hash(key1) != hash(key2))
- **Chained, open addressing, and cuckoo hash tables assume knowing the number of elements you want to store ahead of time**
- You typically **don't** want to use a hash table for a table index

Chained Hashing

- Maintain a linked list of buckets for each slot in the hash table
- Resolves collisions by placing elements with same hash key into the same bucket
- If bucket is full, add another bucket to list
- Downside: the hash table can grow infinitely because you keep adding new buckets
- To handle concurrency, you only need to take a latch on each bucket
- Approaches for non-unique keys
    1. Separate linked list: stores values in separate storage area
    2. Store in bucket: Store duplicate keys in the same buckets (store values with their keys)

Open Addressing Hashing

- Single giant table of slots
- Resolve collisions by linearly searching for the next free slot in the table
- To see if value is present, go to offset using hash, and scan for the key
- To reduce the number of wasteful comparisons, it is important to avoid collisions of $\times$ hashed key. This requires hash table with 2 the number of slots as the number of expected elements

Cuckoo Hashing

- Maintain multiple has tables with different hash functions
- On insert, check every table and pick anyone that has a free slot
- If no table has free slot, evict element from one of them, and rehash it to find a new location
- If we find a cycle, then we can rebuild the entire hash tables with new hash functions

Extendible Hashing

- Chained-hashing approach with buckets.
- Instead of letting the linked list of buckets grow indefinitely, we're going to split them incrementally
- When a bucket is full, we split the bucket and reshuffle its elements
- Uses global and local depths to determine buckets
- Hash table doubles in size to allow for more buckets

Linear Hashing

- Maintain a pointer that tracks the next bucket to split
- Overflow criterion is left up to the implementation
- When any bucket overflows, split the bucket at the pointer location by adding a new slot entry, and create a new hash function
- If hash function maps to slot that has previously been pointed to by pointer, apply the new hash function
- When pointer reaches last slot, delete original hash function and replace it with new hash function

Hash Functions

- We don't need a cryptographic hash function because we don't need to get back key from hash
- We only care about speed and collision rate