Lecture #06: Hash Tables

15-445/645 Database Systems (Fall 2019)
https://15445.courses.cs.cmu.edu/fall2019/
Carnegie Mellon University
Prof. Andy Pavlo

1 Data Structures

A DBMS uses various data structures for many different parts of the system internals:

- **Internal Meta-Data:** Keep track of information about the database and the system state.
- **Core Data Storage:** Can be used as the base storage for tuples in the database.
- **Temporary Data Structures:** The DBMS can build data structures on the fly while processing a query to speed up execution (e.g., hash tables for joins).
- **Table Indexes:** Auxiliary data structures to make it easier to find specific tuples.

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.

2 Hash Table

A hash table implements an associative array abstract data type that maps keys to values. It provides on average $O(1)$ operation complexity and $O(n)$ storage complexity.

A hash table implementation is comprised of two parts:

- **Hash Function:** How to map a large key space into a smaller domain. This is used to compute an index into an array of buckets or slots. Need to consider the trade-off between fast execution vs. collision rate.
- **Hashing Scheme:** How to handle key collisions after hashing. Need to consider the trade-off between the need to allocate a large hash table to reduce collisions vs. executing additional instructions to find/insert keys.

3 Hash Functions

A *hash function* takes in any key as its input. It then return an integer representation of that key (i.e., the "hash"). The function's output is deterministic (i.e., the same key should always generate the same hash output).

The DBMS does not want to use a cryptography hash function (e.g., SHA-256) because we do not need to worry about protecting the contents of keys. These hash functions are primarily used internally by the DBMS and thus information is not leaked outside of the system. For this lecture, we only care about the hash function's speed and collision rate.

The current state-of-the-art hash function is Facebook XXHash3.

4 Static Hashing Schemes

A static hashing scheme is one where the size of the hash table is fixed. This means that if the DBMS runs out of storage space in the hash table, then it has to rebuild it from scratch with a larger table. Typically the new hash table is twice the size of the original hash table.

To reduce the number of wasteful comparisons, it is important to avoid collisions of $\times$ hashed key. This requires hash table with twice the number of slots as the number of expected elements.

4.1 Linear Probe Hashing

This is the most basic hashing scheme. It is also typically the fastest. It uses a single table of slots. The hash function allows the DBMS to quickly jump to slots and look for the desired key.

- Resolve collisions by linearly searching for the next free slot in the table
- To see if value is present, go to slot using hash, and scan for the key. The scan stops if you find the desired key or you encounter an empty slot.

4.2 Robin Hood Hashing

This is an extension of linear probe hashing that seeks to reduce the maximum distance of each key from their optimal position in the hash table. Allows threads to steal slots from "rich" keys and give them to "poor" keys.

- Each key tracks the number of positions they are from where its optimal position in the table.
- On insert, a key takes the slot of another key if the first key is farther away from its optimal position than the second key. The removed key then has to be re-inserted back into the table.

4.3 Cuckoo Hashing

Instead of using a single hash table, this approach maintains multiple has tables with different hash functions. The hash function are the same algorithm (e.g., XXHash, CityHash); they generate different hashes for the same key by using different seed values.

- 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 all of the hash tables with new hash function seeds (less common) or rebuild the hash tables using larger tables (more common).

5 Dynamic Hashing Schemes

The static hashing schemes require the DBMS to know the number of elements it wants to store. Otherwise it has rebuild the table if it needs to grow/shrink in size.

Dynamic hashing schemes are able to resize the hash table on demand without needing to rebuild the entire table. The schemes perform this resizing in different ways that can either maximize reads or writes.

5.1 Chained Hashing

This is the most common dynamic hashing scheme. The DBMS maintains 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 that chain. The hash table can grow infinitely because the DBMS keeps adding new buckets.

5.2 Extendible Hashing

Improved variant of chained hashing that splits buckets instead of letting chains to grow forever. This approach allows multiple slot locations in the hash table to point to the same bucket chain.

The core idea behind re-balancing the hash table is to to move bucket entries on split and increase the number of bits to examine to find entries in the hash table. This means that the DBMS only has to move data within the buckets of the split chain; all other buckets are left untouched.

- The DBMS maintains a global and local depth bit counts that determine the number bits needed to find buckets in the slot array.
- When a bucket is full, the DBMS splits the bucket and reshuffle its elements. If the local depth of the split bucket is less than the global depth, then the new bucket is just added to the existing slot array. Otherwise, the DBMS doubles the size of the slot array to accommodate the new bucket and increments the global depth counter.

5.3 Linear Hashing

Instead of immediately splitting a bucket when it overflows, this scheme maintains a *split pointer* that keeps track of the next bucket to split. No matter whether this pointer is pointing to the bucket that overflowed, the DBMS always splits. The 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.