Lecture #07: Hash Tables

15-445/645 Database Systems (Spring 2024)
https://15445.courses.cs.cmu.edu/spring2024/
Carnegie Mellon University
Jignesh Patel

1 Data Structures

A DBMS uses various data structures for many different parts of the system internals. Some examples include:

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

There are two main design decisions to consider when implementing data structures for the DBMS:

1. Data organization: We need to figure out how to layout memory and what information to store inside the data structure in order to support efficient access.
2. Concurrency: We also need to think about how to enable multiple threads to access the data structure without causing problems, ensuring that the data remains correct and sound.

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 ($O(n)$ in the worst-case) and $O(n)$ storage complexity. Note that even with $O(1)$ operation complexity on average, there are constant factor optimizations which are important to consider in the real world.

A hash table implementation is comprised of two parts:

- **Hash Function:** This tells us how to map a large key space into a smaller domain. It is used to compute an index into an array of buckets or slots. We need to consider the trade-off between fast execution and collision rate. On one extreme, we have a hash function that always returns a constant (very fast, but everything is a collision). On the other extreme, we have a "perfect" hashing function where there are no collisions, but would take extremely long to compute. The ideal design is somewhere in the middle.
- **Hashing Scheme:** This tells us how to handle key collisions after hashing. Here, we need to consider the trade-off between allocating a large hash table to reduce collisions and having to execute additional instructions when a collision occurs.

3 Hash Functions

A *hash function* takes in any key as its input. It then returns 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 need not use a cryptographically secure 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. In general, 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 a larger hash table from scratch, which is very expensive. 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 hashed key. Typically, we use twice the number of slots as the number of expected elements.

The following assumptions usually do not hold in reality:

1. The number of elements is known ahead of time.
2. Keys are unique.
3. There exists a perfect hash function.

Therefore, we need to choose the hash function and hashing schema appropriately.

4.1 Linear Probe Hashing

This is the most basic hashing scheme. It is also typically the fastest. It uses a circular buffer of array slots. The hash function maps keys to slots.

For insertions, when a collision occurs, we linearly search the subsequent slots until an open one is found, looping around from the end to the start of the array if necessary. For lookups, we can check the slot the key hashes to, and search linearly until we find the desired entry. If we reach an empty slot or we iterated over every slot in the hashtable, then the key is not in the table. Note that this means we have to store both key and value in the slot so that we can check if an entry is the desired one.

Deletions are more tricky. We have to be careful about just removing the entry from the slot, as this may prevent future lookups from finding entries that have been put below the now empty slot. There are two solutions to this problem:

- The most common approach is to use "tombstones". Instead of deleting the entry, we replace it with a "tombstone" entry which tells future lookups to keep scanning. Note that insertions are able to insert into tombstone indices.
- The other option is to shift the adjacent data after deleting an entry to fill the now empty slot. However, we must be careful to only move the entries which were originally shifted. This is rarely implemented in practice as it is extremely expensive when we have a large number of keys.

Non-unique Keys: In the case where the same key may be associated with multiple different values or tuples, there are two approaches.

- Separate Linked List: Instead of storing the values with the keys, we store a pointer to a separate storage area that contains a linked list of all the values, which may overflow to multiple pages.
- Redundant Keys: The more common approach is to simply store the same key multiple times in the table. Everything with linear probing still works even if we do this.

Optimizations: There are several ways to further optimize this hashing scheme:

- Specialized hash table implementations based on the data type or size of keys: These could differ in the way they store data, perform splits, etc. For example, if we have string keys, we could store smaller strings in the original hash table and only a pointer or hash for larger strings.
- Storing metadata in a separate array: An example would be to store empty slot/tombstone information in a packed bitmap as a part of the page header or in a separate hash table, which would help us avoid looking up deleted keys.
- Maintaining versions for the hash table and its slots: Since allocating memory for a hash table is expensive, we may want to reuse the same memory repeatedly. To clear out the table and invalidate its entries, we can increment the version counter of the table instead of marking each slot as deleted/empty. A slot can be treated as empty if there is a mismatch between the slot version and table version.

Google's absl::flat_hash_map is a state-of-the-art implementation of Linear Probe Hashing.

4.2 Cuckoo Hashing

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

When we insert, we check every table and choose one that has a free slot (if multiple have one, we can compare things like load factor, or more commonly, just choose a random table). If no table has a free slot, we choose (typically a random one) and evict the old entry. We then rehash the old entry into a different table. In rare cases, we may end up in a cycle. If this happens, 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).

Cuckoo hashing guarantees $O(1)$ lookups and deletions, but insertions may be more expensive.

Professor's note: The essence of cuckoo hashing is that multiple hash functions map a key to different slots. In practice, cuckoo hashing is implemented with multiple hash functions that map a key to different slots in a single hash table. Further, as hashing may not always be $O(1)$, cuckoo hashing lookups and deletions may cost more than $O(1)$.

5 Dynamic Hashing Schemes

The static hashing schemes require the DBMS to know the number of elements it wants to store. Otherwise it has to 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. Keys which hash to the same slot are simply inserted into the linked list for that slot.

To look up an element, we hash to its bucket and then scan for it.

Bloom Filters: Lookup can be optimized by additionally storing bloom filters in the bucket pointer list, which would tell us if a key does not exist in the linked list, thus helping us avoid the cost of lookup in such cases. The bloom filter is a probabilistic data structure, a bitmap, that answers set membership queries. It allows false positives but never false negatives.

Assume we use bitmaps of size n and we employ k hash functions $h_1, h_2, ..., h_k$ to implement the bloom filters. Adding bloom filters modifies the hashing implementation in the following ways:

- Insertion: When we insert x into a bucket in our hash table, we set the indices $h_1(x)\%n$, $h_2(x)\%n$, ..., $h_k(x)\%n$ of the bucket's bitmap as 1s.
- Lookup: When we lookup x, which hashes to a certain bucket, we check whether all indices $h_1(x)\%n$, $h_2(x)\%n$, ..., $h_k(x)\%n$ of the bucket's bitmap are 1s. If they are, the bloom filter tells us that x *may* exist in the bucket linked list. If not, the bloom filter tells us that x *definitely* does not exist in the bucket linked list.

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. These bit counts determine the number of most significant bits we need to look at to find buckets in the slot array.
- When a bucket is full, the DBMS splits the bucket and re-distributes 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. Add a new slot entry and a new hash function, and apply this function to rehash the keys in the split bucket.
- If the original hash function maps to a slot that has previously been pointed to by the split pointer, apply the new hash function to determine the actual location of the key.
- When the pointer reaches the very last slot, delete the original hash function and move the pointer back to the beginning.

If the highest bucket below the split pointer is empty, we can also remove the bucket and move the split pointer in the reverse direction, thereby shrinking the size of the hash table.