# Assignment - 7

# Vijaya Krishna Sameeraj Jonnavithula

# Student ID : 005029574

# Course : 2024 Fall - Algorithms and Data Structures (MSCS-532-B01) - Second Bi-term

# Date : November 24, 2024

Github : <https://github.com/VijayaKrishnaSameerajJonnavithula/Assignment-6>

**Hash Functions and Their Impact**

Creating Useful Hash Functions

The following traits should be present in a decent hash function:

* Uniform Distribution: To reduce clustering and collisions, keys should be dispersed throughout the hash table as equally as feasible.
* Deterministic: The hash value should always be the same when using the same key.
* Efficiency: Even for big datasets, hash value computation should be quick.
* Low Collision Rate: Reduce the number of times that different keys hash to the same value.

Instances of Bad Hash Function Architecture:

* If table\_size is not carefully chosen (e.g., a power of two), using simple modulus operations (e.g., key % table\_size) may result in clustering.
* Actual situation: Performance for lookups dropped from O(1) to O(n) due to frequent collisions caused by a badly designed hash function in a database indexing system.

Strategies for Mitigation:

* For crucial applications, use well-known hash algorithms such as SHA-family hashes or MurmurHash.
* To reduce worst-case scenarios, use universal hashing, in which the hash function is selected at random from a family of hash functions.
* When employing simple hash algorithms, choose prime numbers for the table size to minimise clustering.
* Keeping Complexity and Speed in Check

Hash quality and computing complexity are inherently traded off:

* Simple hash functions, such as bitwise or modulo operations, are quick but might cause collisions with keys that are not widely distributed.
* Cryptographic hashes such as MD5 or SHA-256 are examples of complex hash functions. sluggish and resource-intensive, but extremely collision-resistant.

Case Study: Google's Bigtable and Hash Function Design

Background: Google's Bigtable is a distributed storage system that powers services like Gmail, Maps, and Google Search. It is used to manage enormous amounts of structured data. Bigtable uses hash functions extensively for load balancing, key partitioning, and guaranteeing consistent data distribution across servers in order to attain high efficiency.

Difficulties: Equitable Distribution

Bigtable uses millions of keys to hold large datasets. Some servers can experience overloading while others continue to be underutilized in the absence of a well-designed hash algorithm. System performance and query latency are impacted by this imbalance.

Scalability

The hash function must preserve consistent distribution as more servers are added without necessitating extensive data redistribution, which is computationally costly.

Velocity:

Every second, Bigtable processes billions of requests. Lightweight hash calculation is necessary to prevent bottlenecks.

**Hash Function Design in Bigtable:**

Bigtable uses the following tactics to balance speed and complexity in order to overcome these obstacles:

Custom Hash Function for Lightweight Data:

Bigtable combines bitwise manipulations with basic arithmetic operations. This guarantees a nearly uniform distribution while maintaining the hash function's computational affordability.

Regular Hashing:

Bigtable uses consistent hashing in order to manage scalability. This technique makes sure that only a portion of the keys need to be rehashed when a new server is added or removed. This reduces interference and makes it possible for the system to grow effectively.

Dynamic Load Distribution:

Bigtable employs dynamic partitioning to modify the ranges of keys allocated to each server while keeping an eye on server loads. Because the hash function is intended to deal with these divisions, flexibility is possible without having to recalculate the hash values for every key.

Effect in the Real World: Enhancement in Performance:  
  
Bigtable achieves great throughput with little computational effort by utilizing a lightweight hash algorithm. As a result, the system can handle billions of enquiries with latency times of less than a millisecond.

Both scalability and reliability:  
Regular hashing guarantees that the system can accommodate frequent cluster size adjustments (such as adding servers during periods of high demand) without experiencing appreciable performance deterioration.

Knowledge Acquired:  
It is crucial to balance hash complexity with speed. Bigtable shows how customized hash functions can greatly improve system performance and scalability by catering to the unique requirements of distributed systems, where uniformity and speed are more important than cryptographic security.

**Open Addressing vs. Separate Chaining :**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| | **Feature** | | --- | | | **Open Addressing** |  | | --- | --- | | Separate Chaining |
| Efficiency | -Performs well with low load factors. | Lookup time increases as the table becomes fuller due to clustering. - Performance remains relatively stable as load factors increase, since linked lists are used to handle collisions. |
| Memory Usage | - Uses a single block of contiguous memory, making it more memory-efficient for low load factors.  - Wastes memory if table size is much larger than the number of keys. | - Requires extra memory for linked lists (or other structures) to store colliding elements. |
| Complexity | - Simple to implement but can suffer from primary clustering and secondary clustering.  - Deletion requires careful handling (e.g., marking slots as deleted). | - More complex due to additional pointers or memory management for linked lists. |
| Cache Friendliness | - Better cache performance as all data is stored in contiguous memory. | - Poorer cache performance due to potential random memory access for linked list nodes. |
| Scalability | - Performance deteriorates as load factor exceeds ~0.7. | - Can handle higher load factors without severe performance degradation. |

Favourite Situations for Open Addressing:

* Applications with low load factors and strict memory limits (such as embedded systems and in-memory key-value stores) are the best candidates for this scenario.
* DNS caching (low memory, quick lookup requirements) is one example.
* Redis's open-addressed hash tables (for tiny datasets' compact storage).

Distinct Chaining:

* Scenarios: Ideal for applications requiring flexibility in collision management or resizing due to high or variable load factors.
* Database indexing is one example, where several entries may hash to the same key.
* Java hash maps (which employ independent chaining with balanced trees or linked lists).

Performance in Practice

Open Addressing:

* Effect of Load Factor: When the load factor rises above 0.7, performance quickly deteriorates. Longer probing sequences result in longer lookup and insertion times for load factors near 1.
* Useful Points to Remember: Resize proactively and steer clear of load factors greater than 0.75.

Distinct Chaining:

Effect of Load Factor: Performance is not adversely affected when the load factor is greater than 1. Larger linked lists, however, result in longer average lookup times.

Practical Considerations: Rebalancing linked lists (e.g., converting them to balanced trees) increases complexity, and the use of pointers may cause memory fragmentation.

Real-World Compromises:

* Because of its compactness, open addressing is usually preferred in embedded systems with little memory.
* Because of its capacity to withstand high load factors and adjust to changing workloads, separate chaining is recommended in high-throughput systems such as web servers or distributed hash tables.

REFERENCE:

Books:

Introduction to Algorithms by Cormen, Leiserson, Rivest, and Stein (CLRS):

Chapters on hash tables and collision resolution strategies (covers open addressing and separate chaining in detail).

The Art of Computer Programming by Donald Knuth:

Volume 3 focuses on searching and hashing, providing theoretical insights.

Official Documentation and Libraries:

Python dict Implementation:

Explains how Python's dictionary uses open addressing with dynamic resizing.

Java HashMap Documentation:

Discusses Java's use of separate chaining with linked lists and their transition to balanced trees.

Online Tutorials and Articles:

GeeksforGeeks: Open Addressing vs Separate Chaining:

Detailed comparison with examples and use cases.

Hashing in Redis:

Description of open addressing in Redis’s in-memory data structures.

Research Papers:

A Fast Hash Table Implementation for General Purpose Use by Andrew Kuchling:

Explores optimizations for open addressing in practical systems.

Universal Hashing by Carter and Wegman (1979):

Foundational work on hash function design for collision minimization.