Dorina Ismaili & Deniz Gaiser
02.12.2024

Mining Massive Datasets
Homework 5

Exercise 1

List common mechanisms for resolving hash collisions and explain how key-value dictionaries in C++, Java, and Python handle hash collisions, respectively.

When a hash collision occurs (i.e., two distinct keys k_1 and k_2 hash to the same index), various strategies can resolve the conflict:

a) Chaining: Each bucket in the hash table stores a linked list. If multiple keys hash to the same bucket, they are stored in the linked list of that bucket.

b) Open Adressing: instead of storing multiple items in one bucket, the hash table searches for the next available bucket (slot) in the table.

Some programming languages also use different variations of handling hash-collisions:

C++: Uses chaining for collision resolution, When collisions occur, keys that hash to the same bucket are appended to the linked list of that bucket, $O(1)$ average-case lookup, insertion, and deletion.

Java: In the case of HashMap, java uses chaining with linked lists. When the number of items in a bucket exceeds a threshold (default is 8), the linked list is replaced with a binary search tree (BST) for better performance.

Python: In pytjon's dictionaries open addressing with quadratic probing is used. If a collision occurs, Python searches for the next available slot using a quadratic probing sequence.

Exercise 2

a) Explain MurmurHash and how the seed value creates independent hash functions.
In MurmurHash, the function takes the input AND a seed, and combines those with various elementary operations. If the seed changed, the gives result will change as well, as the seed determines the hashing process that are performed on the input.

b) Is MurmurHash suitable for applications where collision resistance is critical?
MurmurHash is not suitable for such applications, as the results are not necessarily cryptographically secure. Since MurmurHash is deterministic for fixed seeds, that feature can be exploited to find inputs that can re-generate an existing hash entry.

Exercise 3

Suppose our stream consists of the integers 3, 1, 4, 1, 5, 9, 2, 6, 5. Our hash functions will all be of the form $h(x) = (ax + b) \bmod 32$ for some a and b.
The Flajolet-Martin algorithm's estimate depends on the hash function used and the trailing zeros it produces, highlighting the importance of good hash function design.

- $h(x) = (2x + 1) \bmod 32$; Max Tail length = 0, Estimate: $2^0 = 1$
- $h(x) = (3x + 7) \bmod 32$; Max Tail length = 4, Estimate: $2^4 = 16$
- $h(x) = 4x \bmod 32$; Max Tail length =4, Estimate $2^4 = 16$

	$h(x) = (2x + 1) \bmod 32$	$h(x) = (3x + 7) \bmod 32$	$h(x) = 4x \bmod 32$
3	7; Binary = 00111; Tail Length = 0	16;Binary = 10000; Tail Length = 4	12; Binary = 01100; Tail Length = 2
1	3; Binary = 00011; Tail Length = 0	10; Binary = 01010; Tail Length = 1	4; Binary = 00100; Tail Length = 2
4	9; Binary = 01001; Tail Length = 0	19; Binary = 10011; Tail Length = 0	16; Binary = 10000; Tail Length = 4
1	3; Binary = 00011; Tail Length = 0	10; Binary = 01010; Tail Length = 1	4; Binary = 00100; Tail Length = 2
5	11; Binary = 01011; Tail Length = 0	22; Binary = 101101; Tail Length = 1	20; Binary = 10100; Tail Length = 2
9	19; Binary = 10011; Tail Length = 0	6; Binary = 00110; Tail Length = 1	4; Binary = 00100; Tail Length = 2
2	5; Binary = 00101; Tail Length = 0	13; Binary = 01101; Tail Length = 0	8; Binary = 01000; Tail Length = 3
6	13; Binary = 01101; Tail Length = 0	25; Binary = 11001; Tail Length = 0	24; Binary = 11000; Tail Length = 3
5	11; Binary = 01011; Tail Length = 0	22; Binary = 101101; Tail Length = 1	20; Binary = 10100; Tail Length = 2

The provided hash function has certain limitations since the modulo operation is biased by truncating trailing zeros for larger numbers. The modulo operation with $2k$ ensures the hash values are limited, and it increases the risk of collision. Values that are multiples of a certain number may hash to the same or closely related buckets, which undermines the uniform distribution expected of a good hash function.
Advices to use the hash function would be:

1. Randomly select a and b such that $a \neq 0$. Random coefficients reduce correlations with input data and help achieve more uniform hash distributions.

2. If a is even, certain patterns in input values might hash to the same buckets. Using an odd a avoids this issue.

3. 3. Analyze the hash function on sample data to ensure a uniform distribution of outputs.

Exercise 4

Suppose we are given the stream 3, 4, 1, 3, 4, 2, 1, 2 to which we apply the Alon-Matias-Szegedy Algorithm to estimate the k-th moment.

i	$X_i.el$	$X_i.val$
1	3	2
2	4	2
3	1	2
4	3	1
5	4	1
6	2	2
7	1	1
8	2	1

X_i.el is the element in the stream at position i, and X_i.val is the count of that element starting from position i.
No, it does not make sense to maintain X_i for each stream position in practice. That is due to memory usage, random sampling and variance reduction.

To calculate AMS algorithm more efficiently, we would calculate $X.el$ and $X.val$ differently by focusing on random sampling rather than determining them for every position in the stream.

Exercise 5

a) The stream is $[1, 2, 3, 2, 4, 1, 3, 4, 1, 2, 4, 3, 1, 1, 2]$. The code in Python.

b) To compute the exact third moment $k = 3$, we use the formula $M = \sum_i m_i^3$, where m_i is the frequency of each distinct element in the stream. Frequency of each element: 1: Appears 5 times; 2: Appears 4 times; 3: Appears 3 times; 4: Appears 3 times.

$$M = 5^3 + 4^3 + 3^3 + 3^3 = 243.$$

c) The parameter v in the Alon-Matias-Szegedy (AMS) algorithm determines the number of auxiliary variables (X) used to estimate the k-moment. The parameter v controls the trade-off between accuracy and computational resources. Larger v reduces the variance of the AMS estimate, improving accuracy, but at the cost of increased memory and processing requirements.

Exercise 6

If we wanted to estimate the fourth moments the Alon-Matias-Szegedy Algorithm, how would we convert X.val to an estimate of the fourth moment, i.e. how does the function $f(X)$ looks like in this case?

To estimate the fourth moment using the Alon-Matias-Szegedy (AMS) Algorithm, the function $f(X)$ must be adapted for the fourth power. For the k-th moment, the AMS algorithm estimates $F_k = \sum m_i^k$ using

$$f(X) = n * X.val^k,$$

where n is the total number of elements in the stream, and X.val represents the frequency of the randomly selected element. For $k = 4$, the fourth moment estimate is:

$$f(X) = n * X.val^4.$$

The final estimate is the average of these contributions $F_4 = \frac{1}{k} \sum_{j=1}^{k} f(X_j)$.

Exercise 7

See "Task 7.py"