Logistics updates

• Poll for final exam alternative date for those with conflicts
• 2-day extension to project 2
  • Code due on 3/10, 11:59 pm. Write-up due on 3/12, 11:59 pm.
  • Project 3 will still be released on 3/10
In this lecture

• Composite key in hash index
• How to design a good hash function?
Composite keys in hash index

• Composite key: multiple fields as the key (f1, f2, ..., fk)

• How to handle composite keys in hash index?
  • Combine the hash values of each field together
  • Many libs available, e.g., boost::hash_combine, absl::Hash::combine(), etc ...
    e.g., in boost:
    
    seed ^= hash_value + 0x9e3779b9 + (seed<<6) + (seed>>2);

• Search with composite keys
  • Must specify all the keys, equality search only
  • Can’t perform partial key search
    • e.g., hash index on (sid, login)
      • may be used for predicate sid = 12345 AND login = ‘alice’
      • but not sid = 12345, nor logging = ‘Alice’
What might go wrong with hashing?

• Too many items with the same key
  • Extendible hashing and linear hashing will also *fail* when that happens

• Why can that happen?
  • Too many entries with the same key?
    • Not much that we can do, but we can try to incorporate other fields to make the keys distinct if it’s possible from the user’s perspective
    • Alternatively, consider using other types of index

• Hash collision
  • Some hash functions are prone to too many hash collisions
    • For instance, you’re hashing pointers of int64_t,
      • using modular hashing $h(x) = x \mod m$ with $m = 2^d$ for some $d$ is going to leave many buckets completely empty
Designing Good Hash Functions

• Formal set up: let $U = [N]$ denote the numbers $\{0, 1, 2, \ldots, N - 1\}$. For any set $S \subseteq U$, where $|S| = n$, we want to support:
  • add(x): add the key x to S
  • query(x): is the key $q \in S$?
  • delete(x): remove the key x from S

  efficiently!

*We consider the static case here (fixed set S). Note that even though S is fixed, we don’t know S ahead of time. Imagine it’s chosen by an adversary from $\binom{N}{n}$ possible choices*

*Our hash function needs to work well for any such (fixed) set S.*
Static vs Dynamic

• Static: Given a set \( S \) of items, we want to store them so that we can do lookups quickly. E.g., a fixed dictionary.

• Dynamic: here we have a sequence of insert, lookup, and perhaps delete requests. We want to do these all efficiently.
Hash Function Basics

• We will perform inserts and lookups by an array $A$ of $M$ buckets, and a hash function $h : U \rightarrow \{0, \ldots, M - 1\}$ (i.e., $h : U \rightarrow [M]$). Given an element $x$, the idea of hashing is we want to store it in $A[h(x)]$.
  - If $N = |U|$ is small, this problem is trivial. But in practice, $N$ is often big.

• Collision happens when $x \neq y \land h(x) = h(y)$
  - Open hashing with linked list/overflow pages
  - Extendible/linear hashing can be used to alleviate the problem but can’t handle it well if there is skewness in hash values
Desirable Properties

• Small probability of distinct keys colliding: if \( x \neq y \in S \) then \( \Pr_{h \leftarrow H}[h(x) = h(y)] \) is “small”.
  - \( h \leftarrow H \) means the random choice over a family \( H \) of hash functions.

• Small range: we want \( M \) to be small. At odds with first desired property
  - ideally \( M = O(n) \) but it takes too much space.

• Small number of bits to store a hash function \( h \). This is at least \( \Omega(\log_2 |H|) \).

• \( h \) should be easy to compute

• Given this, the time to lookup an item \( x \) is \( O(\text{length of list } A[h(x)]) \)
Bad News

• One way to spread elements out nicely is to spread them randomly. Unfortunately, we can’t just use a random number generator to decide where the next element goes because then we would never be able to find it again. So, we want $h$ to be something “pseudorandom” in some formal sense.

• (Bad news) For any deterministic hash function $h$ (i.e., $|H|=1$), if $|U| \geq (n - 1)M + 1$, there exists a set $S$ of $n$ elements that all hash to the same location.
  • simple pigeon hole argument.
Randomness to Rescue

• Introduce a family of hash functions, \( H \) with \( |H| > 1 \), that \( h \) will be randomly chosen from for each key (but use the same choice for the same key).

• **Universal Hashing**: if \( x \neq y \in S \) then \( \Pr_{h \in H} [h(x) = h(y)] \leq 1/M. \)

• If \( H \) is universal, then for any set \( S \subseteq U \) of size \( n \), for any \( x \in U \) (e.g., that we might want to lookup, \( x \) may not come from \( S \)), if we choose \( h \) at random in a universal hash family \( H \), the expected number of collisions between \( x \) and other elements in \( S \) is at most \( n/M. \)
Property of Universal Hashing

• Proof:
  • Each \( y \in S \) (\( y \neq x \)) has at most a \( 1/M \) chance of colliding with \( x \) by the definition of “universal”. So
  • Let \( C_{xy} = 1 \) if \( x \) and \( y \) collide and 0 otherwise.
  • Let \( C_x \) denote the total number of collisions for \( x \). So, \( C_x = \sum_{y \in S \land y \neq x} C_{xy} \).
  • We know \( E[C_{xy}] = \Pr(x \text{ and } y \text{ collide}) \leq 1/M \).
  • So, by linearity of expectation, \( E[C_x] = \sum_{y \in S \land y \neq x} E[C_{xy}] \leq n/M \).
How to Construct Universal Hashing?

• Consider the case where $|U| = 2^u$ and $M = 2^m$

• Take an $u \times m$ matrix $A$ and fill it with random bits. For $x \in U$, view $x$ as a $u$-bit vector in $\{0, 1\}^u$, and define $h(x) := Ax$ where the calculations are done modulo 2.

• There are $2^{um}$ hash functions in this family $H$

Note that $h(\overrightarrow{0}) = 0$, so picking a random function from $H$ does not map each key to a random place.
Why it is a universal hash family?

- Proof:
  - Let \( A = (\vec{c}_1, \vec{c}_2, \ldots, \vec{c}_m) \), where \( \vec{r}_i \) is the \( i^{th} \) row of the matrix \( A \).
  - Let’s view \( x \in U \) as a vector of \( \{0,1\} \), e.g., \( x = (0, 0, 1, 0, \ldots, 1) \).
  - Then \( h(x) = x_1 \vec{c}_1 + x_2 \vec{c}_2 + \cdots + x_m \vec{c}_m \).
  - Suppose we have \( x^{(1)}, x^{(2)} \in U \), s.t., \( x^{(1)} \neq x^{(2)} \). They will differ in at least one bit. WLOG, say it’s bit 1 and \( x^{(1)}_1 = 0, x^{(2)}_1 = 1 \).
  - For any \( \vec{c}_2, \ldots, \vec{c}_m \in \{0,1\}^u \), let’s fix those vectors (except \( \vec{c}_1 \)).
    - No matter how \( \vec{c}_1 \) changes, \( h(x^{(1)}) = x_2 \vec{c}_2 + \cdots + x_m \vec{c}_m \) remain the same.
    - On the other hand, \( h(x^{(2)}) = x_1 \vec{c}_1 + h(x^{(1)}) \) are all different
      - because each \( \vec{c}_1 \) will be different from any other vectors by at least one bit and the corresponding bit in the hash value is flipped.
      - Thus we have only 1 out of \( 2^m \) different \( \vec{c}_1 \) so that \( h(x^{(1)}) = h(x^{(2)}) \).
  - \( \Pr(h(x^{(1)}) = h(x^{(2)})) = \sum_{\vec{c}_2, \ldots, \vec{c}_m} \Pr(h(x^{(1)}) = h(x^{(2)})|\vec{c}_2, \ldots, \vec{c}_m) \Pr(\vec{c}_2, \ldots, \vec{c}_m) \)

This is not the only way to construct universal hash family though.
Perfect Hashing (for static case)

• We say a hash function is perfect for $S$ if all lookups involve $O(1)$ work.
• Naïve method: an $O(n^2)$ space solution
• Let $H$ be universal and $M = n^2$. Then just pick a random $h$ from $H$ and try it out!

• Claim: If $H$ is universal and $M = n^2$, then $Pr_{h \sim H}(no \ collisions \ in \ S) \geq 1/2$
Naïve method: $O(n^2)$ space

• Proof:
  • How many pairs $(x,y)$ in $S$ are there? Answer:
    • For each pair, the chance they collide is $\leq 1/M$ by definition of “universal”
    • So, $\Pr(\text{exists a collision}) \leq n(n-1)/2M = n(n-1)/2n^2 < 1/2$. (by union bound)
An O(n) space solution (for static S)

- first hash into a table of size n using universal hashing. This will produce some collisions (unless we are extraordinarily lucky)
- then rehash each bin using Method 1, squaring the size of the bin to get zero collisions

Formally:
- a first-level hash function $h$ and first-level table $A$,
- n second-level hash functions $h_1, \ldots, h_n$ and n second-level tables $A_1, \ldots, A_n$
- To lookup an element $x$, we first compute $i = h(x)$ and then find the element in $A_i[h_i(x)]$.
- We omit the analysis of this method.
Dynamic S?

• Cuckoo hashing
  • Linear space
  • Constant lookup time

• Pagh, Rasmus; Rodler, Flemming Friche (2001). "Cuckoo Hashing". *Algorithms — ESA 2001*
Summary

• Today’s lecture
  • Multi-field index key in hash index
  • How to construct a good hash function