Lecture #07

Hash Tables
Project #1 is due February 18th @ 11:59pm.
→ Special Office Hours: February 17th @ 3:00-5:00pm GHC 4405

Homework #2 is due February 16th @ 11:59pm.
We are now going to talk about how to support the DBMS’s execution engine to read/write data from pages.

Two types of data structures:
→ Hash Tables (Unordered)
→ Trees (Ordered)
DATA STRUCTURES

- Internal Meta-data
- Core Data Storage
- Temporary Data Structures
- Table Indexes
DESIGN DECISIONS

Data Organization
→ How we layout data structure in memory/pages and what information to store to support efficient access.

Concurrency
→ How to enable multiple threads to access the data structure at the same time without causing problems.
A **hash table** implements an unordered associative array that maps keys to values.

It uses a **hash function** to compute an offset into this array for a given key, from which the desired value can be found.

**Space Complexity:** \(O(n)\)

**Time Complexity:**

→ Average: \(O(1)\) \(\Rightarrow\) **Databases care about constants!**

→ Worst: \(O(n)\)
STATIC HASH TABLE

Allocate a giant array that has one slot for every element you need to store.

To find an entry, mod the key by the number of elements to find the offset in the array.
Allocate a giant array that has one slot for every element you need to store.

To find an entry, mod the key by the number of elements to find the offset in the array.

\[ \text{hash(key)} \% N \]
UNREALISTIC ASSUMPTIONS

**Assumption #1:** Number of elements is known ahead of time and fixed.

**Assumption #2:** Each key is unique.

**Assumption #3:** Perfect hash function guarantees no collisions.

→ If key1 ≠ key2, then

\[ \text{hash(key1)} \neq \text{hash(key2)} \]
Design Decision #1: Hash Function
→ How to map a large key space into a smaller domain.
→ Trade-off between being fast vs. collision rate.

Design Decision #2: Hashing Scheme
→ How to handle key collisions after hashing.
→ Trade-off between allocating a large hash table vs. additional instructions to get/put keys.
TODAY'S AGENDA

- Hash Functions
- Static Hashing Schemes
- Dynamic Hashing Schemes
HASH FUNCTIONS

For any input key, return an integer representation of that key.

We do not want to use a cryptographic hash function for DBMS hash tables (e.g., SHA-2).

We want something that is fast and has a low collision rate.
HASH FUNCTIONS

**CRC-64** *(1975)*
→ Used in networking for error detection.

**MurmurHash** *(2008)*
→ Designed as a fast, general-purpose hash function.

**Google CityHash** *(2011)*
→ Designed to be faster for short keys (<64 bytes).

**Facebook XXHash** *(2012)*
→ From the creator of zstd compression.

**Google FarmHash** *(2014)*
→ Newer version of CityHash with better collision rates.

← State-of-the-art
HASH FUNCTIONS

**SMhasher**

<table>
<thead>
<tr>
<th>Hash function</th>
<th>MiB/sec</th>
<th>cycl./hash</th>
<th>cycl./map</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>donothing32</td>
<td>11149-60.06</td>
<td>4.00</td>
<td>-</td>
<td>13</td>
</tr>
<tr>
<td>donothing64</td>
<td>11787-7676.42</td>
<td>4.00</td>
<td>-</td>
<td>13</td>
</tr>
<tr>
<td>donothing128</td>
<td>11745-060.76</td>
<td>4.06</td>
<td>-</td>
<td>13</td>
</tr>
<tr>
<td>NOP_OAAT_read64</td>
<td>11372-946.37</td>
<td>14.00</td>
<td>-</td>
<td>47</td>
</tr>
<tr>
<td>BadHash</td>
<td>769.94</td>
<td>73.97</td>
<td>-</td>
<td>47</td>
</tr>
<tr>
<td>sumhash</td>
<td>10699.57</td>
<td>29.53</td>
<td>-</td>
<td>363</td>
</tr>
<tr>
<td>sumhash32</td>
<td>42877.79</td>
<td>23.12</td>
<td>-</td>
<td>863</td>
</tr>
<tr>
<td>multiply_shift</td>
<td>8026.77</td>
<td>26.05</td>
<td>226.80 (8)</td>
<td>345</td>
</tr>
<tr>
<td>pair_multiply_shift</td>
<td>3716.95</td>
<td>40.22</td>
<td>166.34 (3)</td>
<td>609</td>
</tr>
<tr>
<td>crc32</td>
<td>383.12</td>
<td>134.21</td>
<td>257.50 (11)</td>
<td>422</td>
</tr>
<tr>
<td>mu32</td>
<td>350.53</td>
<td>644.31</td>
<td>894.12 (10)</td>
<td>4419</td>
</tr>
</tbody>
</table>

**Summary**

I added some SSE-assisted hashes and fast intel/arm CRC32-C, AES and SHA-HW variants. See also the old GitHub: https://github.com/aappleby/smhasher/wiki, the improved, but unmaintained fork https://github.com/demerphq/smhasher, and the new improved version SMHasher3 https://github.com/twojclik/smhasher3.

So the fastest hash functions on x86_64 without quality problems are:

- xxh3low
- wyhash
- ahash64
- t1ha2_storice
- kpvnhesh
- FermiHash (not portable, too machine specific: 64 vs 32bit, old gcc, ...)
- halftime_hash128
- Spooky32
- pengyhash
- rtmhash32
- mx3
- MUM/mir (different results on 32/64-bit archs, lots of bad seeds to filter out)
- fasthash32

→ Newer version of CityHash with better collision rates.
STATIC HASHING SCHEMES

Approach #1: Linear Probe Hashing
Approach #2: Cuckoo Hashing

There are several other schemes (usually covered in the Advanced DB course):
→ Robin Hood Hashing
→ Hopscotch Hashing
→ Swiss Tables

Both are members of a broader class called “Open Addressing”.
LINEAR PROBE HASHING

Single giant table of slots.

Resolve collisions by linearly searching for the next free slot in the table.

→ To determine whether an element is present, hash to a location in the index and scan for it.

→ Must store the key in the index to know when to stop scanning.

→ Insertions and deletions are generalizations of lookups.

Example: Google’s `absl::flat_hash_map`
LINEAR PROBE HASHING

\[ \text{hash(key)} \% N \]

A
B
C
D
E
F

\[ A | \text{val} \]

\(<\text{key}> | <\text{value}>\)
LINEAR PROBE HASHING

\[ \text{hash(key)} \% N \]

A
B
C
D
E
F

B | val

A | val
LINEAR PROBE HASHING

hash(key) % N

A
B
C
D
E
F

B | val
A | val
C | val
LINEAR PROBE HASHING

\[ \text{hash(key)} \% N \]

A
B
C
D
E
F

<table>
<thead>
<tr>
<th>B</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>val</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CMU-DB
15-445/645 (Spring 2024)
LINEAR PROBE HASHING

hash(key) % N

A
B
C
D
E
F

B | val
A | val
C | val
D | val
**LINEAR PROBE HASHING**

\[
\text{hash(key) } \% N
\]

- A
- B
- C
- D
- E
- F

Hash function: \text{hash(key)} \% N

Example:

- \( B \) with value \( \text{val} \)
- \( A \) with value \( \text{val} \)
- \( C \) with value \( \text{val} \)
- \( D \) with value \( \text{val} \)
- \( E \) with value \( \text{val} \)
LINEAR PROBE HASHING

\[ \text{hash(key)} \% N \]

<table>
<thead>
<tr>
<th>A</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>val</td>
</tr>
<tr>
<td>C</td>
<td>val</td>
</tr>
<tr>
<td>D</td>
<td>val</td>
</tr>
<tr>
<td>E</td>
<td>val</td>
</tr>
<tr>
<td>F</td>
<td>val</td>
</tr>
</tbody>
</table>
LINEAR PROBE HASHING – DELETES

Approach #1: Movement → Rehash keys until you find the first empty slot. → Expensive! May need to reorganize the entire table. → No DBMS does this.

hash(key) % N

Delete

<table>
<thead>
<tr>
<th>A</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>val</td>
</tr>
<tr>
<td>D</td>
<td>val</td>
</tr>
<tr>
<td>E</td>
<td>val</td>
</tr>
<tr>
<td>F</td>
<td>val</td>
</tr>
</tbody>
</table>
LINEAR PROBE HASHING – DELETES

**.hash(key) % N**

- **A**
- **B**
- **C**
- **D**
- **E**
- **F**

Delete

![Diagram](attachment:linear_probe_hashing_deletes.png)
LINEAR PROBE HASHING – DELETES

\[ \text{hash(key)} \mod N \]

A
B
C
D
E
F

|   | val |
---|-----|
B  |     |
A  | val |
D  |     |
E  | val |
F  | val |
LINEAR PROBE HASHING – DELETES

Approach #1: Movement

→ Rehash keys until you find the first empty slot.
→ Expensive! May need to reorganize the entire table.
→ No DBMS does this.

```
hash(key) % N
```

```
A | val
B | val
C |
D |
E | val
F | val
```

Get D

E | val
D | val
C |
A | val
B | val
LINEAR PROBE HASHING – DELETES

Approach #1: Movement

→ Rehash keys until you find the first empty slot.
→ Expensive! May need to reorganize the entire table.
→ No DBMS does this.

\[
\text{hash(key)} \mod N
\]

| A | val |
| B |     |
| C |     |
| D | val |
| E | val |
| F | val |

Get
LINEAR PROBE HASHING – DELETES

Approach #1: Movement
→ Rehash keys until you find the first empty slot.

Get → Rehash keys until you find the first empty slot.
LINEAR PROBE HASHING – DELETES

Approach #1: Movement

→ Rehash keys until you find the first empty slot.
**LINEAR PROBE HASHING – DELETES**

**hash(key) % N**

<table>
<thead>
<tr>
<th>A</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>val</td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>val</td>
</tr>
<tr>
<td>E</td>
<td>val</td>
</tr>
<tr>
<td>F</td>
<td>val</td>
</tr>
</tbody>
</table>

**Approach #1: Movement**

→ Rehash keys until you find the first empty slot.
**LINEAR PROBE HASHING – DELETES**

### hash(key) % N

<table>
<thead>
<tr>
<th>hash(key) % N</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
</tr>
</tbody>
</table>

#### Get

- **Approach #1: Movement**
  - Rehash keys until you find the first empty slot.
LINEAR PROBE HASHING – DELETES

Approach #1: Movement
→ Rehash keys until you find the first empty slot.

$\text{hash(key) } \% N$

<table>
<thead>
<tr>
<th>A</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>val</td>
</tr>
<tr>
<td>C</td>
<td>val</td>
</tr>
<tr>
<td>D</td>
<td>val</td>
</tr>
<tr>
<td>E</td>
<td>val</td>
</tr>
<tr>
<td>F</td>
<td>val</td>
</tr>
</tbody>
</table>

Get
LINEAR PROBE HASHING – DELETES

**Approach #1: Movement**

→ Rehash keys until you find the first empty slot.
→ Expensive! May need to reorganize the entire table.
→ No DBMS does this.
LINEAR PROBE HASHING – DELETES

![Diagram of Linear Probe Hashing]

**Approach #2: Tombstone**

→ Set a marker to indicate that the entry in the slot is logically deleted.
→ Reuse the slot for new keys.
→ May need periodic garbage collection.
Approach #2: Tombstone

→ Set a marker to indicate that the entry in the slot is logically deleted.
→ Reuse the slot for new keys.
→ May need periodic garbage collection.
**LINEAR PROBE HASHING – DELETES**

**Approach #2: Tombstone**

→ Set a marker to indicate that the entry in the slot is logically deleted.

→ Reuse the slot for new keys.

→ May need periodic garbage collection.

```plaintext
hash(key) \% N

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td>B</td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td>B</td>
<td></td>
<td>D</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td>B</td>
<td></td>
<td>D</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td>B</td>
<td></td>
<td>E</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td>B</td>
<td></td>
<td>E</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td>B</td>
<td></td>
<td>F</td>
</tr>
</tbody>
</table>
```

val

val

val

val

val

val
**LINEAR PROBE HASHING – DELETES**

**Approach #2: Tombstone**

→ Set a marker to indicate that the entry in the slot is logically deleted.
→ Reuse the slot for new keys.
→ May need periodic garbage collection.

\[
\text{hash(key)} \mod N
\]

<table>
<thead>
<tr>
<th>A</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>val</td>
</tr>
<tr>
<td>C</td>
<td>val</td>
</tr>
<tr>
<td>D</td>
<td>val</td>
</tr>
<tr>
<td>E</td>
<td>val</td>
</tr>
<tr>
<td>F</td>
<td>val</td>
</tr>
</tbody>
</table>

Get

\[
\text{hash(key)} \mod N
\]
LINEAR PROBE HASHING – DELETES

Approach #2: Tombstone

→ Set a marker to indicate that the entry in the slot is logically deleted.
→ Reuse the slot for new keys.
→ May need periodic garbage collection.
NON-UNIQUE KEYS

Choice #1: Separate Linked List
→ Store values in separate storage area for each key.
→ Value lists can overflow to multiple pages if the number of duplicates is large.

Choice #2: Redundant Keys
→ Store duplicate keys entries together in the hash table.
→ This is what most systems do.
OPTIMIZATIONS

Specialized hash table implementations based on key type(s) and sizes.
→ Example: Maintain multiple hash tables for different string sizes for a set of keys.

Store metadata separate in a separate array.
→ Packed bitmap tracks whether a slot is empty/tombstone.

Use table + slot versioning metadata to quickly invalidate all entries in the hash table.
→ Example: If table version does not match slot version, then treat the slot as empty.

Source: Maksim Kita
CUCKOO HASHING

Use multiple hash functions to find multiple locations in the hash table to insert records.
→ On insert, check multiple locations and pick the one that is empty.
→ If no location is available, evict the element from one of them and then re-hash it find a new location.

Look-ups and deletions are always $O(1)$ because only one location per hash table is checked.

Best open-source implementation is from CMU.
CUCKOO HASHING

Put A: $\text{hash}_1(A)$
$\text{hash}_2(A)$
CUCKOO HASHING

Put A: $\text{hash}_1(A) \quad \text{hash}_2(A)$
CUCKOO HASHING

Put A: $hash_1(A)$
$hash_2(A)$

Put B: $hash_1(B)$
$hash_2(B)$
CUCKOO HASHING

Put A: \( \text{hash}_1(A) \)
\( \text{hash}_2(A) \)

Put B: \( \text{hash}_1(B) \)
\( \text{hash}_2(B) \)
CUCKOO HASHING

Put A: $\text{hash}_1(A)$
$\text{hash}_2(A)$

Put B: $\text{hash}_1(B)$
$\text{hash}_2(B)$

Put C: $\text{hash}_1(C)$
$\text{hash}_2(C)$
CUCKOO HASHING

Put A: \( \text{hash}_1(A) \)
\( \text{hash}_2(A) \)

Put B: \( \text{hash}_1(B) \)
\( \text{hash}_2(B) \)

Put C: \( \text{hash}_1(C) \)
\( \text{hash}_2(C) \)
Put A: \( \text{hash}_1(A) \)
\( \text{hash}_2(A) \)

Put B: \( \text{hash}_1(B) \)
\( \text{hash}_2(B) \)

Put C: \( \text{hash}_1(C) \)
\( \text{hash}_2(C) \)
CUCKOO HASHING

Put A: \( \text{hash}_1(A) \)
\( \text{hash}_2(A) \)

Put B: \( \text{hash}_1(B) \)
\( \text{hash}_2(B) \)

Put C: \( \text{hash}_1(C) \)
\( \text{hash}_2(C) \)
\( \text{hash}_1(B) \)
CUCKOO HASHING

Put A: $\text{hash}_1(A)$  
$\text{hash}_2(A)$

Put B: $\text{hash}_1(B)$  
$\text{hash}_2(B)$

Put C: $\text{hash}_1(C)$  
$\text{hash}_2(C)$  
$\text{hash}_1(B)$
CUCKOO HASHING

Put A: $hash_1(A)$
$hash_2(A)$

Put B: $hash_1(B)$
$hash_2(B)$

Put C: $hash_1(C)$
$hash_2(C)$
$hash_1(B)$
$hash_2(A)$
CUCKOO HASHING

Put A: \( \text{hash}_1(A) \)
\( \text{hash}_2(A) \)

Put B: \( \text{hash}_1(B) \)
\( \text{hash}_2(B) \)

Put C: \( \text{hash}_1(C) \)
\( \text{hash}_2(C) \)
\( \text{hash}_1(B) \)
\( \text{hash}_2(A) \)

Get B: \( \text{hash}_1(B) \)
\( \text{hash}_2(B) \)
OBSERVATION

The previous hash tables require the DBMS to know the number of elements it wants to store.
→ Otherwise, it must rebuild the table if it needs to grow/shrink in size.

Dynamic hash tables incrementally resize themselves as needed.
→ Chained Hashing
→ Extendible Hashing
→ Linear Hashing
CHAINED HASHING

Maintain a linked list of buckets for each slot in the hash table.

Resolve collisions by placing all elements with the same hash key into the same bucket.
→ To determine whether an element is present, hash to its bucket and scan for it.
→ Insertions and deletions are generalizations of lookups.
CHAINED HASHING

hash(key) % N

Bucket Pointers

Buckets
CHAINED HASHING

\[ \text{hash(key)} \mod N \]

Put A

Bucket Pointers

Buckets

A | val
CHAINED HASHING

hash(key) % N

Bucket Pointers

Put A
Put B

B | val

A | val

Buckets
CHAINED HASHING

\(\text{hash(key)} \% N\)

Put A
Put B
Put C

Bucket Pointers

B | val

A | val

C | val

Buckets
CHAINED HASHING

.hash(key) % N

Put A
Put B
Put C
Put D

Bucket
Pointers

val

val

val

val

B

A

C

D

60
CHAINED HASHING

$\text{hash(key)} \% N$

<table>
<thead>
<tr>
<th>Put A</th>
<th>Put B</th>
<th>Put C</th>
<th>Put D</th>
<th>Put E</th>
</tr>
</thead>
</table>

Bucket Pointers

<table>
<thead>
<tr>
<th>B</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>val</td>
</tr>
<tr>
<td>C</td>
<td>val</td>
</tr>
</tbody>
</table>

D | val
CHAINED HASHING

\[ \text{hash(key)} \mod N \]

Put A
Put B
Put C
Put D
Put E

Bucket Pointers

<table>
<thead>
<tr>
<th>Bucket</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bucket</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bucket</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bucket</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
A & | val \\
B & | val \\
C & | val \\
D & | val \\
E & | val \\
\end{align*}
\]
CHAINED HASHING

$\text{hash(key)} \% N$

Put A
Put B
Put C
Put D
Put E
Put F

Bucket Pointers

$A | val$
$B | val$
$C | val$
$D | val$
$E | val$
$F | val$
CHAINED HASHING

\[ \text{hash(key)} \mod N \]

Bucket Pointers

- A | val
- B | val
- C | val
- D | val
- E | val
- F | val

Bloom Filter

Bloom Filter

Bloom Filter
### CHAINED HASHING

**hash(key) % N**

**Bucket Pointers**

- **Bloom Filter**
- **Bloom Filter**
- **Bloom Filter**

**Get G**

Does key 'G' exist?

- **B | val**
- **A | val**
- **C | val**
- **F | val**
- **D | val**
- **E | val**
CHAINED HASHING

\[ \text{hash(key)} \% N \]

Bucket Pointers

Get G

Does key 'G' exist?

\[ \begin{align*}
A & | \text{val} \\
C & | \text{val} \\
D & | \text{val} \\
F & | \text{val}
\end{align*} \]
BLOOM FILTERS

Probabilistic data structure (bitmap) that answers set membership queries.

→ False negatives will never occur.
→ False positives can sometimes occur.
→ See Bloom Filter Calculator.

**Insert(x):**

→ Use $k$ hash functions to set bits in the filter to 1.

**Lookup(x):**

→ Check whether the bits are 1 for each hash function.
# Bloom Filters

**Bloom Filter**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
**BLOOM FILTERS**

**Bloom Filter**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Insert('RZA')

\[\text{hash}_1('RZA') = 2222 \mod 8 = 6\]

\[\text{hash}_2('RZA') = 4444 \mod 8 = 4\]
**BLOOM FILTERS**

**Bloom Filter**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Insert('RZA')

Insert('GZA')

\[
\text{hash}_1('GZA') = 5555 \% 8 = 3 \\
\text{hash}_2('GZA') = 7777 \% 8 = 1
\]
BLOOM FILTERS

Bloom Filter

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- Insert('RZA')
- Insert('GZA')
- Lookup('RZA') → TRUE

hash₁('RZA') = 2222 % 8 = 6
hash₂('RZA') = 4444 % 8 = 4
**Bloom Filters**

**Bloom Filter**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- `hash_1('Raekwon') = 3333 % 8 = 5`
- `hash_2('Raekwon') = 8899 % 8 = 3`

- **Insert('RZA')**
- **Insert('GZA')**
- **Lookup('RZA') → TRUE**
- **Lookup('Raekwon') → FALSE**
**BLOOM FILTERS**

**Bloom Filter**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Insert('RZA')

Insert('GZA')

Lookup('RZA') → **TRUE**

Lookup('Raekwon') → **FALSE**

Lookup('ODB') → **TRUE** (false positive)

\[ \text{hash}_1('ODB') = 6699 \% 8 = 3 \]

\[ \text{hash}_2('ODB') = 9966 \% 8 = 6 \]
EXTENDIBLE HASHING

Chained-hashing approach that splits buckets incrementally instead of letting the linked list grow forever.

Multiple slot locations can point to the same bucket chain.

Reshuffle bucket entries on split and increase the number of bits to examine.

→ Data movement is localized to just the split chain.
EXTENDIBLE HASHING

global 2

[Diagram showing extendible hashing with global and local levels, numbers 00010…, 01110…, 10101…, and 11010…]
EXTENDIBLE HASHING

Hash Bits

- **global**
  - 2

- **Hash Bits**
  - 00
  - 01
  - 10
  - 11

- **local**
  - 00010...
  - 01110...

  - 10101...
  - 10011...

  - 11010...
EXTENDIBLE HASHING

Hash Bits

00
01
10
11

global

2

local

00010...
0110...

10101...
10011...

11010...

local
EXTENDIBLE HASHING

Get A
\[
\text{hash}(A) = 01110\ldots
\]
EXTENDIBLE HASHING

Get \(A\) with \(hash(A) = 01110\ldots\)
EXTENDIBLE HASHING

Get A
\[\text{hash}(A) = 01110...\]

Put B
\[\text{hash}(B) = 10111...\]
EXTENDIBLE HASHING

Get A  
\[hash(A) = 01110...\]

Put B  
\[hash(B) = 10111...\]

Put C  
\[hash(C) = 10100...\]
EXTENDIBLE HASHING

Get A
\[ \text{hash}(A) = 01110... \]

Put B
\[ \text{hash}(B) = 10111... \]

Put C
\[ \text{hash}(C) = 10100... \]
EXTENDIBLE HASHING

Get A
\[ \text{hash}(A) = 01110... \]

Put B
\[ \text{hash}(B) = 10111... \]

Put C
\[ \text{hash}(C) = 10100... \]

\[ \begin{array}{c}
\text{global} \\
00 \\
01 \\
10 \\
11 \\
\end{array} \begin{array}{c}
\\text{local} \\
\text{local} \\
\text{local} \\
\text{local} \\
\end{array} \]
## EXTENDIBLE HASHING

<table>
<thead>
<tr>
<th>global</th>
<th>1 local</th>
<th>2 local</th>
<th>2 local</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>00010...</td>
<td>00010...</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>01101...</td>
<td>01101...</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>10011...</td>
<td>10011...</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>10111...</td>
<td>10111...</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>11010...</td>
<td>11010...</td>
<td></td>
</tr>
<tr>
<td>011</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Get A**
  \[ \text{hash}(A) = 01110... \]

- **Put B**
  \[ \text{hash}(B) = 10111... \]

- **Put C**
  \[ \text{hash}(C) = 10100... \]
EXTENDIBLE HASHING

Get A
\[\text{hash}(A) = 01110\ldots\]

Put B
\[\text{hash}(B) = 10111\ldots\]

Put C
\[\text{hash}(C) = 10100\ldots\]
EXTENDIBLE HASHING

Get A
\[ \text{hash}(A) = 01110... \]

Put B
\[ \text{hash}(B) = 10111... \]

Put C
\[ \text{hash}(C) = 10100... \]
EXTENDIBLE HASHING

Get A
hash(A) = 01110...

Put B
hash(B) = 10111...

Put C
hash(C) = 10100...

Global slots: 3
- 000
- 010
- 100
- 110
- 001
- 011
- 101
- 111

Hash values:
- 00010...
- 01110...
- 10011...
- 10101...
- 10111...
- 11010...

**Extendible Hashing**

- Get A
  \[\text{hash}(A) = 01110...\]

- Put B
  \[\text{hash}(B) = 10111...\]

- Put C
  \[\text{hash}(C) = 10100...\]
EXTENDIBLE HASHING

Get $A$
\[\text{hash}(A) = 01110\ldots\]

Put $B$
\[\text{hash}(B) = 10111\ldots\]

Put $C$
\[\text{hash}(C) = 10100\ldots\]
EXTENDIBLE HASHING

Get A
\( \text{hash}(A) = 01110... \)

Put B
\( \text{hash}(B) = 10111... \)

Put C
\( \text{hash}(C) = 10100... \)
LINEAR HASHING

The hash table maintains a pointer that tracks the next bucket to split.
→ When any bucket overflows, split the bucket at the pointer location.

Use multiple hashes to find the right bucket for a given key.

Can use different overflow criterion:
→ Space Utilization
→ Average Length of Overflow Chains
LINEAR HASHING
LINEAR HASHING

Split Pointer

$hash_0(key) = key \% n$
LINEAR HASHING

Get 6

$$hash_0(6) = 6 \mod 4 = 2$$

$$hash_0(key) = key \mod n$$
**LINEAR HASHING**

\[ \text{hash}_0(key) = key \mod n \]

**Get 6**
\[ \text{hash}_0(6) = 6 \mod 4 = 2 \]

**Put 17**
\[ \text{hash}_0(17) = 17 \mod 4 = 1 \]
**LINEAR HASHING**

Get 6
\[ \text{hash}_0(6) = 6 \mod 4 = 2 \]

Put 17
\[ \text{hash}_0(17) = 17 \mod 4 = 1 \]

\[ \text{hash}_0(\text{key}) = \text{key} \mod n \]
LINEAR HASHING

\[
\text{hash}_0(key) = key \mod n
\]

Get 6
\[
\text{hash}_0(6) = 6 \mod 4 = 2
\]

Put 17
\[
\text{hash}_0(17) = 17 \mod 4 = 1
\]

Overflow!
LINEAR HASHING

Get 6
\[ \text{hash}_0(6) = 6 \% 4 = 2 \]

Put 17
\[ \text{hash}_0(17) = 17 \% 4 = 1 \]

\[ \text{hash}_0(\text{key}) = \text{key} \% n \]
\[ \text{hash}_1(\text{key}) = \text{key} \% 2n \]
LINEAR HASHING

Get 6
\[ \text{hash}_0(6) = 6 \mod 4 = 2 \]

Put 17
\[ \text{hash}_0(17) = 17 \mod 4 = 1 \]

\[ \text{hash}_0(\text{key}) = \text{key} \mod n \]
\[ \text{hash}_1(\text{key}) = \text{key} \mod 2n \]

Split Pointer

Overflow!
**LINEAR HASHING**

**Split Pointer**

\[ \text{hash}_0(key) = key \mod n \]

\[ \text{hash}_1(key) = key \mod 2n \]

- Get 6
  \[ \text{hash}_0(6) = 6 \mod 4 = 2 \]

- Put 17
  \[ \text{hash}_0(17) = 17 \mod 4 = 1 \]
  \[ \text{hash}_1(8) = 8 \mod 8 = 0 \]
LINEAR HASHING

Get 6
\[\text{hash}_0(6) = 6 \mod 4 = 2\]

Put 17
\[\text{hash}_0(17) = 17 \mod 4 = 1, \quad \text{hash}_1(8) = 8 \mod 8 = 0, \quad \text{hash}_1(20) = 20 \mod 8 = 4\]

\[\text{hash}_0(\text{key}) = \text{key} \mod n\]
\[\text{hash}_1(\text{key}) = \text{key} \mod 2n\]
LINEAR HASHING

Get 6
\[ \text{hash}_0(6) = 6 \mod 4 = 2 \]

Put 17
\begin{align*}
\text{hash}_0(17) &= 17 \mod 4 = 1 \\
\text{hash}_1(8) &= 8 \mod 8 = 0 \\
\text{hash}_1(20) &= 20 \mod 8 = 4
\end{align*}

\[ \text{hash}_0(\text{key}) = \text{key} \mod n \]
\[ \text{hash}_1(\text{key}) = \text{key} \mod 2n \]
LINEAR HASHING

\[
\begin{align*}
\text{Split Pointer} & \\
0 & \rightarrow \{8\} \\
1 & \rightarrow \{5, 9, 13\} \rightarrow \{17\} \\
2 & \rightarrow \{\} \\
3 & \rightarrow \{6\} \\
4 & \rightarrow \{7, 11\} \rightarrow \{20\}
\end{align*}
\]

Get 6
\[
\text{hash}_0(6) = 6 \mod 4 = 2
\]

Put 17
\[
\text{hash}_0(17) = 17 \mod 4 = 1 \\
\text{hash}_1(8) = 8 \mod 8 = 0 \\
\text{hash}_1(20) = 20 \mod 8 = 4
\]

\[
\begin{align*}
\text{hash}_0(\text{key}) & = \text{key} \mod n \\
\text{hash}_1(\text{key}) & = \text{key} \mod 2n
\end{align*}
\]
**LINEAR HASHING**

**Split Pointer**

- **hash\_0(key) = key % n**
- **hash\_1(key) = key % 2n**

Get 6
\[ hash\_0(6) = 6 \mod 4 = 2 \]

Put 17
\[ hash\_0(17) = 17 \mod 4 = 1 \]
\[ hash\_1(8) = 8 \mod 8 = 0 \]
\[ hash\_1(20) = 20 \mod 8 = 4 \]

Get 20
\[ hash\_0(20) = 20 \mod 4 = 0 \]
### LINEAR HASHING

**Split Pointer**

- \( hash_0(key) = key \mod n \)
- \( hash_1(key) = key \mod 2n \)

**Example:**

- **Get 6**
  - \( hash_0(6) = 6 \mod 4 = 2 \)

- **Put 17**
  - \( hash_0(17) = 17 \mod 4 = 1 \)
  - \( hash_1(8) = 8 \mod 8 = 0 \)
  - \( hash_1(20) = 20 \mod 8 = 4 \)

- **Get 20**
  - \( hash_0(20) = 20 \mod 4 = 0 \)
LINEAR HASHING

Split Pointer

Get 6
\[ \text{hash}_0(6) = 6 \% 4 = 2 \]

Put 17
\[ \text{hash}_0(17) = 17 \% 4 = 1 \]
\[ \text{hash}_1(8) = 8 \% 8 = 0 \]
\[ \text{hash}_1(20) = 20 \% 8 = 4 \]

Get 20
\[ \text{hash}_0(20) = 20 \% 4 = 0 \]
\[ \text{hash}_1(20) = 20 \% 8 = 4 \]

\[ \text{hash}_0(key) = key \% n \]
\[ \text{hash}_1(key) = key \% 2n \]
Linear Hashing

\[ \text{hash}_0(key) = key \mod n \]
\[ \text{hash}_1(key) = key \mod 2n \]

Get 6
\[ \text{hash}_0(6) = 6 \mod 4 = 2 \]

Put 17
\[ \text{hash}_0(17) = 17 \mod 4 = 1 \]
\[ \text{hash}_1(8) = 8 \mod 8 = 0 \]
\[ \text{hash}_1(20) = 20 \mod 8 = 4 \]

Get 20
\[ \text{hash}_0(20) = 20 \mod 4 = 0 \]
\[ \text{hash}_1(20) = 20 \mod 8 = 4 \]

Get 9
\[ \text{hash}_0(9) = 9 \mod 4 = 1 \]
LINEAR HASHING

Split Pointer

\[ hash_0(key) = key \% n \]
\[ hash_1(key) = key \% 2n \]

Get 6
\[ hash_0(6) = 6 \% 4 = 2 \]

Put 17
\[ hash_0(17) = 17 \% 4 = 1 \]
\[ hash_1(8) = 8 \% 8 = 0 \]
\[ hash_1(20) = 20 \% 8 = 4 \]

Get 20
\[ hash_0(20) = 20 \% 4 = 0 \]
\[ hash_1(20) = 20 \% 8 = 4 \]

Get 9
\[ hash_0(9) = 9 \% 4 = 1 \]
LINEAR HASHING – RESIZING

Splitting buckets based on the split pointer will eventually get to all overflowed buckets.
→ When the pointer reaches the last slot, remove the first hash function and move pointer back to beginning.

If the “highest” bucket below the split pointer is empty, the hash table could remove it and move the splinter pointer in reverse direction.
LINEAR HASHING – DELETES

Split Pointer

\[ \text{hash}_0(key) = key \mod n \]
\[ \text{hash}_1(key) = key \mod 2n \]

Delete 20
\[ \text{hash}_0(20) = 20 \mod 4 = 0 \]
LINEAR HASHING – DELETES

Split Pointer

hash_0(key) = key % n
hash_1(key) = key % 2n

Delete 20
hash_0(20) = 20 % 4 = 0
hash_1(20) = 20 % 8 = 4
LINEAR HASHING – DELETES

`hash_0(key) = key % n`

`hash_1(key) = key % 2n`

Delete 20

`hash_0(20) = 20 % 4 = 0`

`hash_1(20) = 20 % 8 = 4`
**LINEAR HASHING – DELETES**

**Hash Functions**

- $\text{hash}_0(key) = key \mod n$
- $\text{hash}_1(key) = key \mod 2n$

**Example:** Delete 20

- $\text{hash}_0(20) = 20 \mod 4 = 0$
- $\text{hash}_1(20) = 20 \mod 8 = 4$

**Split Pointer Diagram**

- Split Pointer:
  - 0
  - 1
  - 2
  - 3
  - 4

- Hashing:
  - Hashing functions show how keys are mapped to slots.
  - Key 20 is deleted from the hash table.
LINEAR HASHING – DELETES

**Split Pointer**

$hash_0(key) = key \% n$

Delete 20

$hash_0(20) = 20 \% 4 = 0$

$hash_1(20) = 20 \% 8 = 4$
LINEAR HASHING – DELETES

Split Pointer

Split Pointer

hash₀(key) = key % n

Delete 20
hash₀(20) = 20 % 4 = 0
hash₁(20) = 20 % 8 = 4

Put 21
hash₀(21) = 21 % 4 = 1

Overflow!
CONCLUSION

Fast data structures that support $O(1)$ look-ups that are used all throughout DBMS internals.

→ Trade-off between speed and flexibility.

Hashing schemes get used for both in-memory and on-disk.

→ Linear Probing, Cuckoo Hashing, Chained Hashing: Generally used for in-memory hash tables; e.g., in hash join and aggregate operators.

→ Extendible Hashing, Linear Hashing: for disk-based hash indexing.
NEXT CLASS

B+ Trees

→ aka “The Greatest Data Structure of All Time”