Hash Tables
UPCOMING DATABASE EVENTS

MapD Talk
→ Thursday Sept 20th @ 12:00pm
→ CIC 4th Floor
ADMINISTRIVIA

Project #1 is due Wednesday Sept 26th @ 11:59pm

Homework #2 is due Friday Sept 28th @ 11:59pm
REMINDER

If you have a question during the lecture, raise your hand and stop me. Do not come up to the front after the lecture.

There are no stupid questions(*).
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
→ Trees
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 associative array abstract data type that maps keys to values.

It uses a hash function to compute an offset into the array, from which the desired value can be found.
STATIC HASH TABLE

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

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

```
hash(key)
0  abc
1  Ø
2  def
...  
n  xyz
```
Allocate a giant array that has one slot for every element that you need to record.

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

You know the number of elements ahead of time.

Each key is unique.

Perfect hash function.

→ If key1 ≠ key2, then

\[
\text{hash(key1)} \neq \text{hash(key2)}
\]
HASH TABLE

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 find/insert keys.
TODAY'S AGENDA

Hash Functions
Static Hashing Schemes
Dynamic Hashing Schemes
HASH FUNCTIONS

We don’t want to use a cryptographic hash function for our join algorithm.

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

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

**Google CityHash** (2011)
→ Based on ideas from MurmurHash2
→ Designed to be faster for short keys (<64 bytes).

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

**CLHash** (2016)
→ Fast hashing function based on **carry-less multiplication**.
# HASH FUNCTION BENCHMARKS

*Intel Core i7-8700K @ 3.70GHz*

![Graph showing throughput vs. key size for different hash functions (std::hash, MurmurHash3, CityHash, FarmHash, CLHash).](image)

- **Throughput (MB/sec)**
- **Key Size (bytes)**

**Sources:**
- Fredrik Widlund

---

*Source: Fredrik Widlund*
HASH FUNCTION BENCHMARKS

Intel Core i7-8700K @ 3.70GHz

Throughput (MB/sec) vs. Key Size (bytes)

- std::hash
- MurmurHash3
- CityHash
- FarmHash
- CLHash

Source: Fredrik Widlund
STATIC HASHING SCHEMES

Approach #1: Linear Probe Hashing

Approach #2: Robin Hood Hashing

Approach #3: Cuckoo Hashing
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.
→ Have to store the key in the index to know when to stop scanning.
→ Insertions and deletions are generalizations of lookups.
LINEAR PROBE HASHING

hash(key)

A
B
C
D
E
F

A | val

<key> | <value>
LINEAR PROBE HASHING

hash(key)

A
B
C
D
E
F
LINEAR PROBE HASHING

hash(key)

A
B
C
D
E
F

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

hash(key)
LINEAR PROBE HASHING

hash(key)

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

hash(key)

A | val
B | val
C | val
D | val
E | val
F | val
NON-UNIQUE KEYS

Choice #1: Separate Linked List
→ Store values in separate storage area for each key.
NON-UNIQUE KEYS

Choice #1: Separate Linked List
→ Store values in separate storage area for each key.

Choice #2: Redundant Keys
→ Store duplicate keys entries together in the hash table.
OBSERVATION

To reduce the # of wasteful comparisons, it is important to avoid collisions of hashed keys.

This requires a hash table with ~2x the number of slots as the number of elements.
ROBIN HOOD HASHING

Variant of linear probe hashing that steals slots from "rich" keys and give them to "poor" keys.

→ Each key tracks the number of positions they are from where its optimal position in the table.
→ On insert, a key takes the slot of another key if the first key is farther away from its optimal position than the second key.
ROBIN HOOD HASHING

hash(key)

A | val [θ]

# of "Jumps" From First Position
ROBIN HOOD HASHING

\text{hash(key)}

A
B
C
D
E
F

\begin{array}{c|c}
B & \text{val} [\emptyset] \\
A & \text{val} [\emptyset] \\
\end{array}
ROBIN HOOD HASHING

hash(key)

A
B
C
D
E
F

B\mid val[0]

A\mid val[0]

C\mid val[1]

A[0] == C[0]
ROBIN HOOD HASHING

hash(key)

A
B
C
D
E
F

B | val [0]
A | val [0]
C | val [1]
D | val [1]

C[1] > D[0]
ROBIN HOOD HASHING

hash(key)

<table>
<thead>
<tr>
<th>A</th>
<th>val [0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>val [1]</td>
</tr>
<tr>
<td>D</td>
<td>val [1]</td>
</tr>
<tr>
<td>E</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>

A[0] == E[0]
ROBIN HOOD HASHING

hash(key)

A
B
C
D
E
F

B | val [0]
A | val [0]
C | val [1]
D | val [1]

A[0] == E[0]
C[1] == E[1]
ROBIN HOOD HASHING

hash(key)

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

B | val | [0] |
A | val | [0] |
C | val | [1] |
D | val | [1] |

A[0] == E[0]
C[1] == E[1]
ROBIN HOOD HASHING

hash(key)

A B C D E F

B | val [0]
A | val [0]
C | val [1]
E | val [2]
D | val [2]

A[0] == E[0]
C[1] == E[1]
ROBIN HOOD HASHING

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

\[
\begin{array}{c|c}
A & \text{val} [0] \\
B & \text{val} [0] \\
C & \text{val} [1] \\
D & \text{val} [2] \\
E & \text{val} [2] \\
F & \text{val} [1] \\
\end{array}
\]

\[ D[2] > F[0] \]
Cuckoo Hashing

Use multiple hash tables with different hash functions.
→ On insert, check every table and pick anyone that has a free slot.
→ If no table has a free slot, evict the element from one of them and then re-hash it to find a new location.

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

Hash Table #1

Insert A

hash₁(A)  hash₂(A)

Hash Table #2
Cuckoo Hashing

Hash Table #1

A | val

Hash Table #2

Insert A

hash_1(A)  hash_2(A)
CUCKOO HASHING

Hash Table #1

\[ A | val \]

Hash Table #2

\[ \text{Insert A} \quad \text{hash}_1(A) \quad \text{hash}_2(A) \]

\[ \text{Insert B} \quad \text{hash}_1(B) \quad \text{hash}_2(B) \]
**Cuckoo Hashing**

Hash Table #1

- Insert A
  - hash\(_1\)(A)
  - hash\(_2\)(A)

- A | val

Hash Table #2

- Insert B
  - hash\(_1\)(B)
  - hash\(_2\)(B)

- B | val
Cuckoo Hashing

Insert A
\[ \text{hash}_1(A) \quad \text{hash}_2(A) \]

Insert B
\[ \text{hash}_1(B) \quad \text{hash}_2(B) \]

Insert C
\[ \text{hash}_1(C) \quad \text{hash}_2(C) \]

Hash Table #1

Hash Table #2

A | val

B | val
Cuckoo Hashing

Hash Table #1

Insert A
hash₁(A) hash₂(A)

Insert B
hash₁(B) hash₂(B)

Insert C
hash₁(C) hash₂(C)

Hash Table #2

A | val

C | val
CUCKOO HASHING

Hash Table #1

Insert A
\( \text{hash}_1(A) \) \( \text{hash}_2(A) \)

\( A | \text{val} \)

Insert B
\( \text{hash}_1(B) \) \( \text{hash}_2(B) \)

Insert C
\( \text{hash}_1(C) \) \( \text{hash}_2(C) \)
\( \text{hash}_1(B) \)

Hash Table #2

\( C | \text{val} \)

...
Cuckoo Hashing

Hash Table #1

- Insert A
  - $\text{hash}_1(A)$
  - $\text{hash}_2(A)$

- Insert B
  - $\text{hash}_1(B)$
  - $\text{hash}_2(B)$

- Insert C
  - $\text{hash}_1(C)$
  - $\text{hash}_2(C)$
  - $\text{hash}_1(B)$

Hash Table #2

- Insert C
  - $\text{hash}_1(C)$
  - $\text{hash}_2(C)$

- Insert A
  - $\text{hash}_1(A)$
  - $\text{hash}_2(A)$

- Insert B
  - $\text{hash}_1(B)$
  - $\text{hash}_2(B)$

- Insert C
  - $\text{hash}_1(C)$
  - $\text{hash}_2(C)$
  - $\text{hash}_1(B)$
Cuckoo Hashing

Hash Table #1

Insert A
hash\(_{1}(A)\) hash\(_{2}(A)\)

Insert B
hash\(_{1}(B)\) hash\(_{2}(B)\)

Insert C
hash\(_{1}(C)\) hash\(_{2}(C)\)
hash\(_{1}(B)\) hash\(_{2}(A)\)

Hash Table #2

C | val

A | val
CUCKOO HASHING

Make sure that we don’t get stuck in an infinite loop when moving keys.

If we find a cycle, then we can rebuild the entire hash tables with new hash functions.
→ With two hash functions, we (probably) won’t need to rebuild the table until it is at about 50% full.
→ With three hash functions, we (probably) won’t need to rebuild the table until it is at about 90% full.
The previous hash tables require knowing the number of elements you want to store ahead of time.

→ Otherwise you have rebuild the entire table if you need to grow/shrink.
The previous hash tables require knowing the number of elements you want to store ahead of time.
→ Otherwise you have rebuild the entire table if you need to grow/shrink.

Dynamic hash tables are able to grow/shrink on demand.
→ 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)

...
CHAINED HASHING

$hash(key)$

Buckets
The hash table can grow infinitely because you just keep adding new buckets to the linked list.

You only need to take a latch on the bucket to store a new entry or extend the linked list.
EXTENDIBLE HASHING

Chained-hashing approach where we split buckets instead of letting the linked list grow forever. This requires reshuffling entries on split, but the change is localized.
EXTENDIBLE HASHING

- **Global**
- **Local 1**
  - 00...
  - 01...
  - 10...
  - 11...

- **Local 2**
  - 10101...
  - 10011...
  - 11010...
EXTENDIBLE HASHING

Find A
hash(A) = 01110...

hash(A) = 10011...
hash(A) = 10101...
hash(A) = 00010...

global

local

local

local

local
EXTENDIBLE HASHING

Find A
hash(A) = 01110...
EXTENDIBLE HASHING

Find A
hash(A) = 01110...

Insert B
hash(B) = 10111...
EXTENDIBLE HASHING

Find A
dash hash(A) = 01110...

Insert B
dash hash(B) = 10111...

CMU 15-445/645 (Fall 2018)
EXTENDIBLE HASHING

Find A
hash(A) = 01110...

Insert B
hash(B) = 10111...

Insert C
hash(C) = 10100...
**Extentible Hashing**

- **Global**
  - 00…
  - 01…
  - 10…
  - 11…

- **Local**
  - 00010…
  - 01110…
  - 10101…
  - 10011…
  - 10111…
  - 11010…

**Operations**

- **Find A**
  - hash(A) = 01110…

- **Insert B**
  - hash(B) = 10111…

- **Insert C**
  - hash(C) = 10100…
EXTENDIBLE HASHING

Find A
hash(A) = 01110...

Insert B
hash(B) = 10111...

Insert C
hash(C) = 10100...
**EXTENDIBLE HASHING**

<table>
<thead>
<tr>
<th>global</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>000...</td>
<td></td>
</tr>
<tr>
<td>010...</td>
<td></td>
</tr>
<tr>
<td>100...</td>
<td></td>
</tr>
<tr>
<td>110...</td>
<td></td>
</tr>
<tr>
<td>001...</td>
<td></td>
</tr>
<tr>
<td>011...</td>
<td></td>
</tr>
<tr>
<td>101...</td>
<td></td>
</tr>
<tr>
<td>111...</td>
<td></td>
</tr>
</tbody>
</table>

- **Find A**
  - $\text{hash}(A) = 01110...$

- **Insert B**
  - $\text{hash}(B) = 10111...$

- **Insert C**
  - $\text{hash}(C) = 10100...$
LINEAR HASHING

Maintain a pointer that tracks the next bucket to split.

When any bucket overflows, split the bucket at the pointer location.

Overflow criterion is left up to the implementation.

→ Space Utilization
→ Average Length of Overflow Chains
LINEAR HASHING

Split Pointer

0
1
2
3

hash_1(key) = key % n

8
20
5
9
6
7
11
LINEAR HASHING

- **Split Pointer**
- **Hash Function**: \( \text{hash}_1(key) = key \mod n \)
- **Find 6**
  \[ \text{hash}_1(6) = 6 \mod 4 = 2 \]
- **Example**: 6 is hashed to slot 2.
LINEAR HASHING

Split Pointer

Find 6
\[ \text{hash}_1(6) = 6 \mod 4 = 2 \]

Insert 17
\[ \text{hash}_1(17) = 17 \mod 4 = 1 \]

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

Split Pointer

Find 6
$\text{hash}_1(6) = 6 \% 4 = 2$

Insert 17
$\text{hash}_1(17) = 17 \% 4 = 1$

Overflow!

$\text{hash}_1(\text{key}) = \text{key} \% n$
LINEAR HASHING

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

\[
\text{hash}_2(key) = key \mod 2n
\]

Find 6
\[
\text{hash}_1(6) = 6 \mod 4 = 2
\]

Insert 17
\[
\text{hash}_1(17) = 17 \mod 4 = 1
\]

Overflow!
LINEAR HASHING

\[ \text{hash}_1(\text{key}) = \text{key} \mod n \]
\[ \text{hash}_2(\text{key}) = \text{key} \mod 2n \]

Find 6
\[ \text{hash}_1(6) = 6 \mod 4 = 2 \]

Insert 17
\[ \text{hash}_1(17) = 17 \mod 4 = 1 \]
LINEAR HASHING

- Split Pointer

$\text{hash}_1(key) = key \% n$
$\text{hash}_2(key) = key \% 2n$

Find 6
$\text{hash}_1(6) = 6 \% 4 = 2$

Insert 17
$\text{hash}_1(17) = 17 \% 4 = 1$
LINEAR HASHING

Split Pointer

\[ \text{hash}_1(key) = key \mod n \]
\[ \text{hash}_2(key) = key \mod 2n \]

Find 6
\[ \text{hash}_1(6) = 6 \mod 4 = 2 \]

Insert 17
\[ \text{hash}_1(17) = 17 \mod 4 = 1 \]

Find 20
\[ \text{hash}_1(20) = 20 \mod 4 = 0 \]
**Linear Hashing**

- **Hash Function 1**
  \[ \text{hash}_1(key) = key \% n \]
  - Example: \[ \text{hash}_1(6) = 6 \% 4 = 2 \]
- **Hash Function 2**
  \[ \text{hash}_2(key) = key \% 2n \]
  - Example: \[ \text{hash}_2(20) = 20 \% 8 = 4 \]

**Find 6**
\[ \text{hash}_1(6) = 6 \% 4 = 2 \]

**Insert 17**
\[ \text{hash}_1(17) = 17 \% 4 = 1 \]

**Find 20**
- **Hash Function 1**
  \[ \text{hash}_1(20) = 20 \% 4 = 0 \]
- **Hash Function 2**
  \[ \text{hash}_2(20) = 20 \% 8 = 4 \]
LINEAR HASHING

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

The pointer can also move backwards when buckets are empty.
CONCLUSION

Fast data structures that support $O(1)$ look-ups that are used all throughout the DBMS internals. → Trade-off between speed and flexibility.

Hash tables are usually **not** what you want to use for a table index...

**Postgres Demo**
NEXT CLASS

B+Trees
Skip Lists
Radix Trees