Hash tables

Chapter-contents:

- Hashing: general properties
- Families of hash functions
- Strategies to build hash tables
- Hashing extensions: Consistent hashing

Table of contents

Hash tables

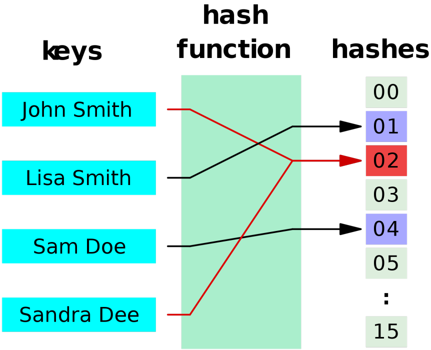
- Hashing: general properties
- Families of hash functions
- Strategies to build hash tables
- Hashing extensions: Consistent hashing

Hashing: definition

A hash table consists in :

- a hash map (function) $h : U \to [0..m-1]$.
- an array v of size m

Generally: h must be fast (eval. in $O(1)$). $U = [0..L]$ with $L \gg m$ possibly unbounded.
Similar techniques when U is tuples (fixed-length), or strings (variable length)...



Use cases for hashing

- Dictionaries and unordered sets in programs
- Indexing data (DBMS index, data mining)
- Managing a cache (ex: linux kernel, DBMS)
- Distributing data: partitioning, join evaluation (databases)
- Symbol tables in compilers

- Fingerprints (Git, identifying duplicates)
- Data mining: fingerprints, projections
- Cryptography: checking passwords, authentification, data integrity (checksum)

Strategies for hashing

Many possible strategies to deal with collisions:

- Static hashing: the number of buckets is fixed
    - Closed addressing collisions solved through chaining
    - Open addressing collisions solved by probing locations

      Linear probing, Quadratic probing, Double hashing

      Cuckoo hashing not strictly perfect, but O(1) access too
    - Perfect hashing builds collision-free hash functions

      FKS
- Dynamic hashing the number of buckets can vary

  Extendable hashing, Linear hashing

⚠ Some people use different denominations: *Open hashing* = separate chaining, *Closed hashing* = open addressing

Table of contents

Hash tables

- Hashing: general properties
- Families of hash functions
- Strategies to build hash tables
- Hashing extensions: Consistent hashing

Families H of hash functions: independence

$H = \{h : U \to [0..m-1]\}$

- *Universal family* :

$$\forall x, y \in U, x \neq y : \Pr_{h \leftarrow H}[h(x) = h(y)] \leq 1/m$$

- *k-independent = k-universal = strongly universal$_k$*:

$$\forall \text{ distinct } x_1, \ldots, x_k \in U, \forall y_1, \ldots, y_k \in [1..m] : \Pr_{h \leftarrow H}[h(x_1) = y_1 \wedge \cdots \wedge h(x_k) = y_k] = 1/m^k$$

Special case: k=2 ↔ pairwise independent.

In practice, often a bit too strong: either use (μ, k)-independent : $\Pr(...) \leq \mu/m^k$, or prove that the family is close to a k-independent one.

Families H of hash functions: independence (2)

- How to build efficiently universal or k-independent hash functions?
  - degree k polynomials with random coefficients, modulo some prime (Carter-Wegman)
  - combinations of multiplications, bit-shifting... (Dietzfelbinger)
  - tabulation hashing: 3-independent
  - Seigel

- How much independence is needed?
  - chaining: universal is enough
  - linear probing: 5-independence, or tabulation
  - cuckoo hashing: $\log n$-independence ($? > 6$), or tabulation if static.

Building hash functions in practice

Typically use bit operations: XOR, multiply, shift.

Metrics to evaluate "quality":

- chi-square test
- avalanche test
- collisions
- number of CPU cycles

[https://en.wikipedia.org/wiki/List_of_hash_functions]

```
unsigned int MurmurHash2 ( const void * key, int len, unsigned int seed )
{
  // 'm' and 'r' are mixing constants generated offline.
  // They're not really 'magic', they just happen to work well.

  const unsigned int m = 0x5bd1e995;
  const int r = 24;

  // Initialize the hash to a 'random' value

  unsigned int h = seed ^ len;

  // Mix 4 bytes at a time into the hash

  const unsigned char * data = (const unsigned char *)key;

  while(len >= 4)
  {
    unsigned int k = *(unsigned int *)data;

    k *= m;
    k ^= k >> r;
    k *= m;

    h *= m;
    h ^= k;

    data += 4;
    len -= 4;
  }
// [cut some code here]...
 return h;
}
```

Python built-in hash() function

- Hash is not very random, but meant to be good locality for usage in dict with real data.
- For (small) integers, identity mapping.
- Hash returns integers (generally 24bytes, depending on interpreter).
- mutable types such as lists are not hashable.
- custom-defined types are hashable. Hash is based on id, unless you redefine __hash__().

```
>>> map(hash, [0,1,2])
[0, 1, 2]
>>> hash(-.5)
-1152921504606846976
>>> hash(.5)
1152921504606846976
>>> map(hash, ("namea", "nameb", "namec"))
[-1658398457, -1658398460, -1658398459]
""" hashes of string/bytes objects
varies from one session to another,
unless you set PYTHONHASHSEED """
>>> hash([1,3,4])
TypeError: unhashable type: 'list'
```

Hash function implementation in Python

```
tuplehash(PyTupleObject *v)
{
  register long x, y;
  register Py_ssize_t len = Py_SIZE(v);
  register PyObject **p;
  long mult = 1000003L;
  x = 0x345678L;
  p = v->ob_item;
  while (--len >= 0) {
    y = PyObject_Hash(*p++);
    if (y == -1)
      return -1;
    x = (x ^ y) * mult;
    /*cast might truncate len */
    mult += (long)(82520L + len + len);
  }
  x += 97531L;
  if (x == -1)
    x = -2;
  return x;
}
```

```
static long
string_hash(PyStringObject *a)
{
  register Py_ssize_t len;
  register unsigned char *p;
  register long x;

  if (a->ob_shash != -1)
    return a->ob_shash;
  len = Py_SIZE(a);
  p = (unsigned char *) a->ob_sval;
  x = *p << 7;
  while (--len >= 0)
    x = (1000003*x) ^ *p++;
  x ^= Py_SIZE(a);
  if (x == -1)
    x = -2;
  a->ob_shash = x;
  return x;
}
```

[https://svn.python.org/projects/python/trunk/Objects]

Pour des algorithmes de hachage sécurisés (famille SHA, etc): librairie `hashlib`

Hash function implementation in Java

```java
public int hashCode() {
    int h = hash;
    if (h == 0 && value.length > 0) {
        char val[] = value;

        for (int i = 0; i < value.length; i++) {
            h = 31 * h + val[i];
        }
        hash = h;
    }
    return h;
}
```

[http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/lang/String.java]

See also other mechanisms (secure hash functions and hashcodes):

[https://github.com/google/guava/tree/master/guava/src/com/google/common/hash]

Families H of hash functions: cryptographic hash functions

Just the function, not an array for storage.
Required properties:

- avalanche
- fast to evaluate
- one-way function:
  - resist preimage attack : from h and c, finding m such that $h(m) = c$ must be infeasible
  - and resist secondary preimage attack : from h, c, and m_1 s.t. $h(m_1) = c$, finding m_2 such that $h(m_2) = c$ must be infeasible.

(where *infeasible* means (provably) extremely hard... for current computers)

"Much more than encryption algorithms, one-way hash functions are the workhorses of modern cryptography" (Bruce Schneier)

Families H of hash functions: locality sensitive hashing

$H = \{h : \mathcal{M} \rightarrow \mathcal{S}\}$

$\mathcal{M} = (M, d)$ metric space, S set of buckets, $r_1 < r_2$ thresholds, $P_1 > P_2$ probabilities. H is (r_1, r_2, P_1, P_2)- sensitive if:

- $d(p, q) \leq r_1 \implies \mathrm{Pr}_{h \leftarrow H}[h(p) = h(q)] \geq P_1$.
- $d(p, q) \geq r_2 \implies \mathrm{Pr}_{h \leftarrow H}[h(p) = h(q)] \leq P_2$.

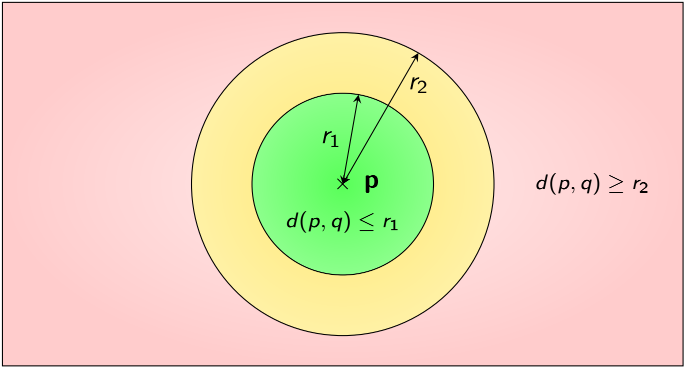


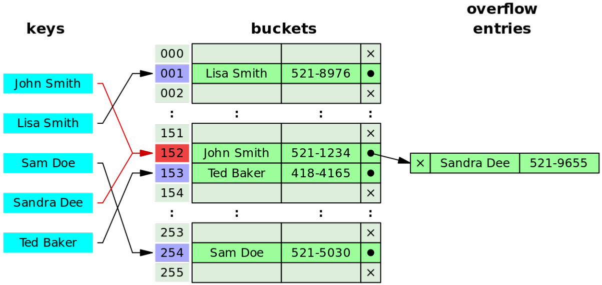
Table of contents

Hash tables

- Hashing: general properties
- Families of hash functions
- Strategies to build hash tables
- Hashing extensions: Consistent hashing

Hash tables

In DB, bucket = page containing multiple keys.
Typical default strategy: closed addressing.



But overflow chains can spoil performance.

Load factor: $$\frac{\#\text{keys}}{\#\text{buckets}}$$

Load factor around 80% generally considered best.

Java 10 HashMap default load factor: 0.75

Java 8 HashMap: Closed addressing

- The hash function used is :

```
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

- Solve collisions through chaining
- The table is actually an array of bins. Each bins contains a collection of triples: `<hash_of_key, key, value>`
- Start with 16 bins. Resize when load factor exceeded. $(\#slots * = 2)$
- When a bin gets overpopulated, Java switches its content to a Tree to allow faster lookups
- Not thread-safe because of resizing.

[http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/util/HashMap.java]

Linear Probing

General idea: use a hash function h_0. Table h tries to insert w in $h_0(w)$, but if already occupied, inserts in next available bucket.

After x, u, s,
inserting w,
then z

h_0 :

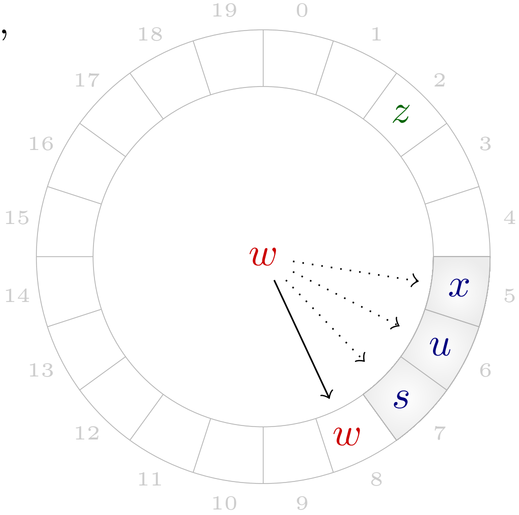
$x \mapsto 5$

$u \mapsto 6$

$s \mapsto 7$

$w \mapsto 5$

$z \mapsto 2$

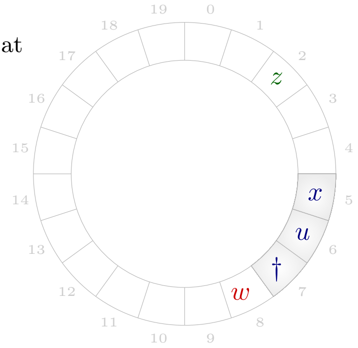


Linear Probing: deletions

Deletions: 2 possible strategies:

- lazy deletion with special flag
- or shift next cells in cascade if needed.

Deleting s at position 7



Quadratic probing: similar to Linear Probing but for i^{th} probe, check $h_0(x) + c_1 * i + c_2 * i^2 \mod m$ instead of $h_0(x) + i \mod m$

Linear Probing: guarantees

Guarantees:

Search	$O(1)$ expected
Delete	$O(1)$ expected
Insert	$O(1)$ expected

as long as load factor not too high, and provided we use 5-independent hash functions or tabulation hashing.

In practice, some popular functions achieve reasonably low collisions, even without 5-independence.
Typically among the fastest, because good locality.

Python 2.7 dictionaries: Probing, but not linearly

- The table is actually an array of triples: `<hash_of_key, key, value>`
- Start with 8 slots. Resized each time dictionary is 2/3 full.

  (#slots $* = 4$ for small dict, $* = 2$ for large. Dictionary may be rebuilt if dictionary is too sparse, which is checked only during insertions)

- initial index is basically "low order bits of key.hash()"

  (not random, but fast and good locality for real world input which are often sequences)

- Solve collision with a pseudo_random probing sequence that is guaranteed to visit all slots eventually.

```
""" To compute the index of the next slot when starting from slot j """
j = (5*j) + 1 + perturb;
perturb >>= PERTURB_SHIFT;     # Remark: eventually, perturb will be 0
# now use j % 2**i as the next table index;
```

[https://stackoverflow.com/questions/327311/how-are-pythons-built-in-dictionaries-implemented]

[https://github.com/wklken/Python-2.7.8/blob/master/Objects/dictobject.c]

Current Python dict=as above+2 optimizations: compact layout+dict-sharing .

Python 3.6+ dictionaries: adding 1 level of indirection

Python 2 dictionary layout is as follows:

```
entries = [['--', '--', '--'],
           [-8522787127447073495, 'barry', 'green'],
           ['--', '--', '--'],
           ['--', '--', '--'],
           ['--', '--', '--'],
           [-9092791511155847987, 'timmy', 'red'],
           ['--', '--', '--'],
           [-6480567542315338377, 'guido', 'blue']]
```

Python 3 instead inserts new keys sequentially in a dense table dk_entries. Another (sparse) table dk_indices allows to locate existing entries:

...

actually, -1 or -2 instead.

```
indices = [None, 1, None, None, None, 0, None, 2]
entries = [[-9092791511155847987, 'timmy', 'red'],
           [-8522787127447073495, 'barry', 'green'],
           [-6480567542315338377, 'guido', 'blue']]
```

insertion order

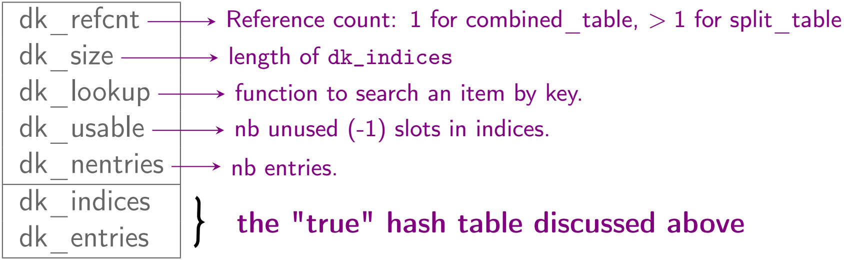
- ✔ Compression ($\geq$ 30% less)
- ✔ Faster resize
- ✔ Faster iterations

Python dictionaries: optimizations

User-defined dictionaries (explicit `dict()`) are as above: `combined-table dict`. However, Python3.4+ stores objects attributes in : `split-table dict`.

In split-table dictionaries, the tables split values from the keys: values are kept in a separate array for each object so that multiple dict can share same key data structure.

Python 3 dictionary layout (supports both split and combined dict):



dk_refcnt → Reference count: 1 for combined_table, > 1 for split_table

dk_size → length of `dk_indices`

dk_lookup → function to search an item by key.

dk_usable → nb unused (-1) slots in indices.

dk_nentries → nb entries.

dk_indices
dk_entries } **the "true" hash table discussed above**

[https://github.com/python/cpython/blob/master/Objects/dictobject.c]

More about hashing in Python...

- Python set() (at least up to Python3.11 in 2023) adopts layout ±similar to (but distinct from) Python2.7's dict.

  Why not use the compact layout? (i) Typical use pattern differs, (ii) it would not save as much space, and (iii) it's potentially incompatible with current implementation of probing in sets (hybrid b/w strictly linear and randomized sequence).

  Consequence: set iteration not based on insertion order!

  ```python
  list(set(['aa','c','ab'])) # needs not be ['aa','c','ab']
  list(dict.fromkeys(['c','ab','d','c','e'])) == ['c','ab','d','e']
  ```

- How can we fix hash seeds if needed?
  - we can set PYTHONHASHSEED env. var. before running Python; ex:

    ```
    export PYTHONHASHSEED=0
    ```

  - or we can hash objects with a custom hash (or use hashlib).

Cuckoo Hashing

General idea:

- use two hash function, each on a distinct array of size $m/2$ (or distinct part of array)
- when inserting, if one of the 2 possible location is empty, insert there
- otherwise, displace (remove and re-insert) one of the items

Cuckoo chick ejects egg

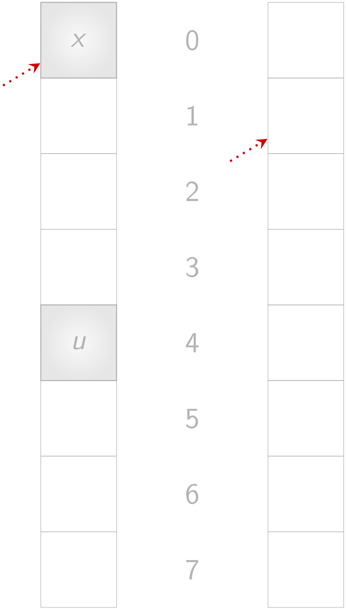
Reed warbler feeds cuckoo

https://www.cs.tau.ac.il/~shanir/advanced-seminar-data-structures-2009/bib/pagh01cuckoo.pdf

Cuckoo Hashing

After x, u, inserting s

		h_0	h_1
x	$\mapsto$	0	5
u	$\mapsto$	4	2
s	$\mapsto$	0	1
w	$\mapsto$	4	1
z	$\mapsto$	4	1
y	$\mapsto$	0	2

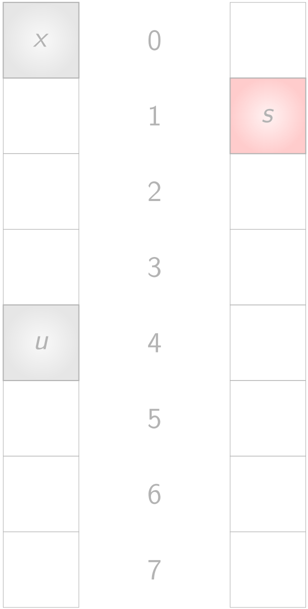


One of the two positions is free, so we insert s in it.

Cuckoo Hashing

After x, u, inserting s

		h_0	h_1
x	$\mapsto$	0	5
u	$\mapsto$	4	2
s	$\mapsto$	0	1
w	$\mapsto$	4	1
z	$\mapsto$	4	1
y	$\mapsto$	0	2

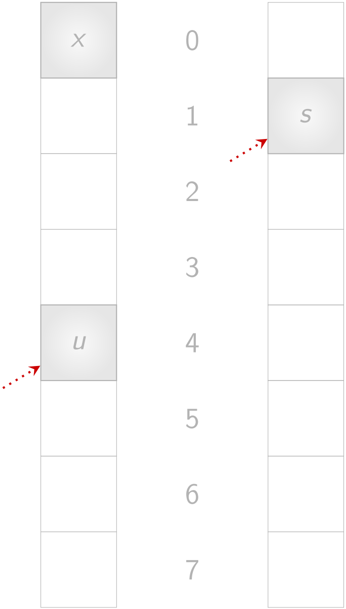


One of the two positions is free, so we insert s in it.

Cuckoo Hashing

After $x, u, s,$ inserting w

	h_0	h_1
$x \mapsto$	0	5
$u \mapsto$	4	2
$s \mapsto$	0	1
$w \mapsto$	4	1
$z \mapsto$	4	1
$y \mapsto$	0	2

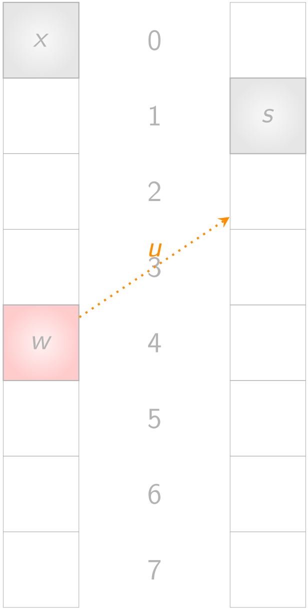


None of the two positions is free, so we insert w arbitrarily in one of them.

Cuckoo Hashing

After x, u, s, inserting w

		h_0	h_1
x	$\mapsto$	0	5
u	$\mapsto$	4	2
s	$\mapsto$	0	1
w	$\mapsto$	4	1
z	$\mapsto$	4	1
y	$\mapsto$	0	2

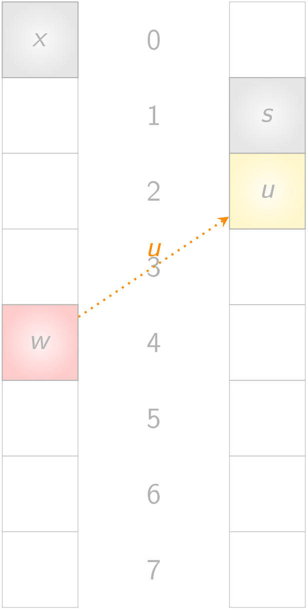


u is pushed out of its current location by w. u is pushed into its alternative location.

Cuckoo Hashing

After $x, u, s,$ inserting w

		h_0	h_1
x	$\mapsto$	0	5
u	$\mapsto$	4	2
s	$\mapsto$	0	1
w	$\mapsto$	4	1
z	$\mapsto$	4	1
y	$\mapsto$	0	2

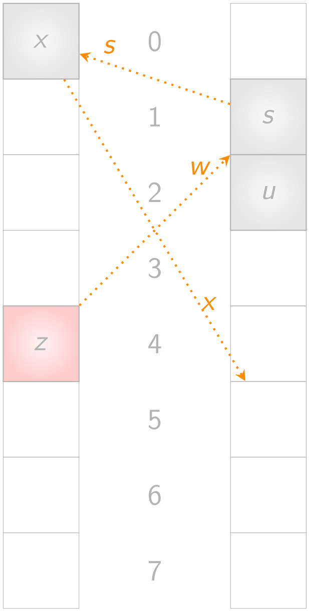


u is pushed out of its current location by w. u is pushed into its alternative location.

Cuckoo Hashing

After
$x, u, s, w,$
inserting z

		h_0	h_1
x	$\mapsto$	0	5
u	$\mapsto$	4	2
s	$\mapsto$	0	1
w	$\mapsto$	4	1
z	$\mapsto$	4	1
y	$\mapsto$	0	2

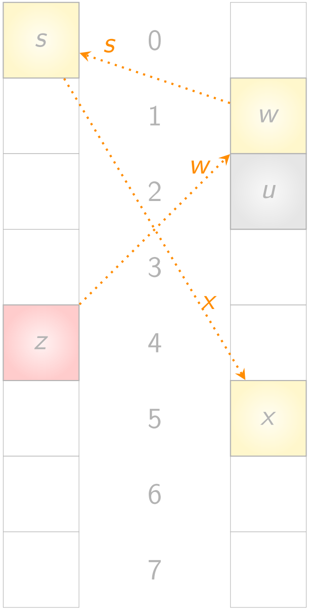


w is pushed out of its current location by z. w is pushed into its alternative location, which triggers other relocations in cascade.

Cuckoo Hashing

After
$x, u, s, w,$
inserting z

		h_0	h_1
x	$\mapsto$	0	5
u	$\mapsto$	4	2
s	$\mapsto$	0	1
w	$\mapsto$	4	1
z	$\mapsto$	4	1
y	$\mapsto$	0	2

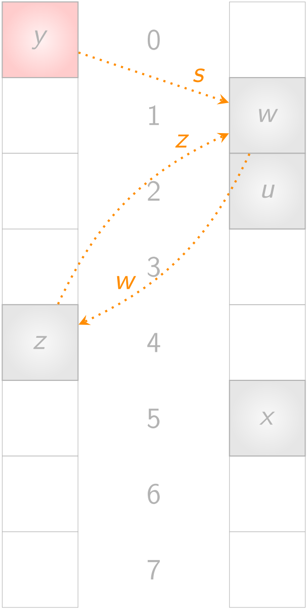


w is pushed out of its current location by z. w is pushed into its alternative location, which triggers other relocations in cascade.

Cuckoo Hashing

After x, u, s, w, z, inserting y

		h_0	h_1
x	$\mapsto$	0	5
u	$\mapsto$	4	2
s	$\mapsto$	0	1
w	$\mapsto$	4	1
z	$\mapsto$	4	1
y	$\mapsto$	0	2



s is pushed out of its current location by y. But the relocations create a cycle. So the hash table is rebuilt with new hash functions.

Cuckoo Hashing : guarantees

Guarantees:

Search	$O(1)$
Delete	$O(1)$
Insert	$O(1)$ expected (amortized)

Typically 20-30% slower than linear probing.

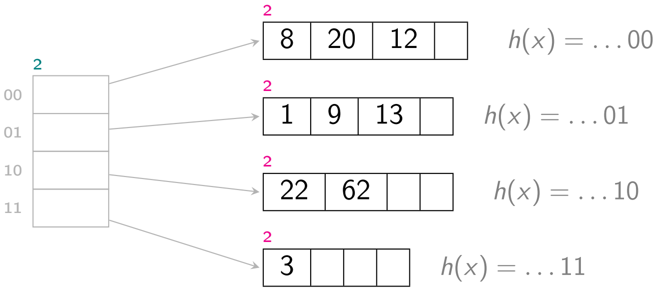
Extendible Hashing

General idea: organize hashes into a *trie*=radix search tree.

A simple extendible hashing scheme can maintain:

- a dynamic set of buckets. Each bucket b has a depth d_b.
- a bucket directory. With depth d. $\forall b, d \geq d_b$

$\Longrightarrow$ hashes viewed as bit strings (LSB order), trie collapsed as an array directory.



Directory

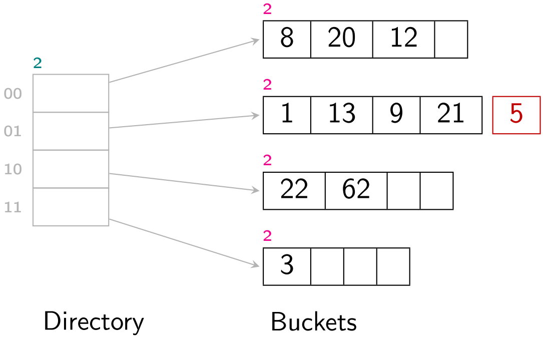
Buckets, where we represent $h(k)$ rather than k

http://cgi.di.uoa.gr/~ad/M149/p315-fagin.pdf

Extendible Hashing: bucket overflow (1)

What happens when bucket capacity is exceeded?

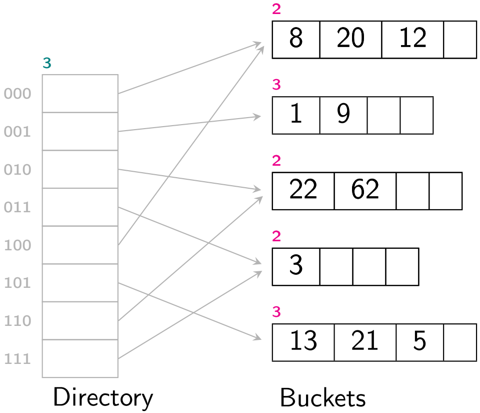
- When bucket depth $d_b = d$, split overflowing bucket, relocate corresponding entries and double directory.
- When bucket depth $d_b < d$, split overflowing bucket and relocate corresponding entries.



Extendible Hashing: bucket overflow (2)

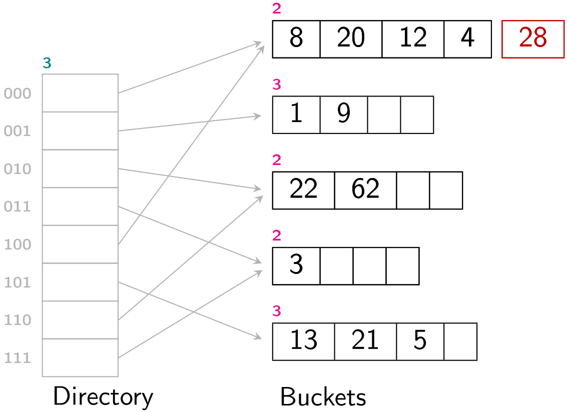
What happens when bucket capacity is exceeded?

- When bucket depth $d_b = d$, split overflowing bucket, relocate corresponding entries and double directory.
- When bucket depth $d_b < d$, split overflowing bucket and relocate corresponding entries.



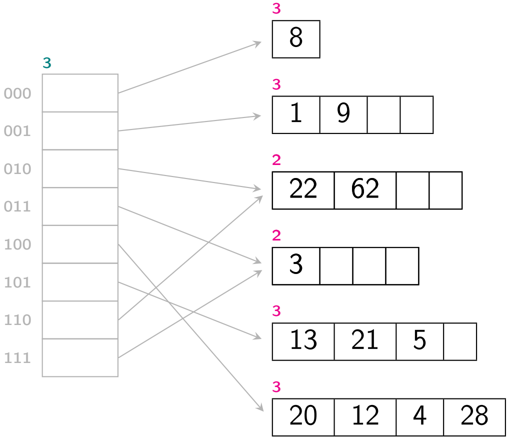
Extendible Hashing: bucket overflow (3)

What happens when bucket capacity is exceeded?



Extendible Hashing: bucket overflow (4)

What happens when bucket capacity is exceeded?



Directory

Buckets

Extendible Hashing

For deletions: if empties bucket, try to merge with its "twin" bucket, and adapts depth.

- ✔ At most 2 blocks to access data
- ✔ grows and shrinks gracefully as data evolves

- ✘ When data is skewed and considering bit $d + 1$ does not prevent overflow, extendible hashing resorts to *overflow chains*.
- ✘ needs separate directory datastructure

Used in file systems such as GFS2.

Linear Hashing

General idea:

- global variables: $N = \#$buckets (initially), $i =$current level, $b' =$current bucket for round robin
- use family of hash functions $h_0, \ldots, h_k$ such that h_{i+1} splits each bucket of h_i in two: b and $b + N * 2^i$.
- Initially, use h_0. Whenever a bucket b "overflows" :
  - use overflow chain for that bucket b
  - split bucket b' (not necessarily b) in round-robin order, using the next-level function to rehash b'. Increment b'.
- Eventually, all buckets (esp. overflow buckets with their chain) will be split, and we will move to next level.
- When searching for item, we evaluate first hash function. Then, depending on position of bucket w.r.t. b', may need to rehash with next function (2 possible locations then).

Used in Hash index (PostgreSQL).

http://infolab.cs.unipi.gr/pre-eclass/courses/db/db-post/readings/Litwin-VLDB80.pdf

Bibliography. . .

Nice lectures, with theoretical analysis of performance (probing, cuckoo):

https://web.stanford.edu/class/cs166/

Dynamic hashing : extendible and linear hashing.

https://db.inf.uni-tuebingen.de/staticfiles/teaching/ss13/db2/db2-06-1up.pdf

Some surveys:

http://www.cse.fau.edu/~xqzhu/papers/ACS.Chi.2017.Hashing.pdf

http://romania.amazon.com/techon/presentations/PracticalSurveyHashTables_AurelianTutuianu.pdf

Table of contents

Hash tables

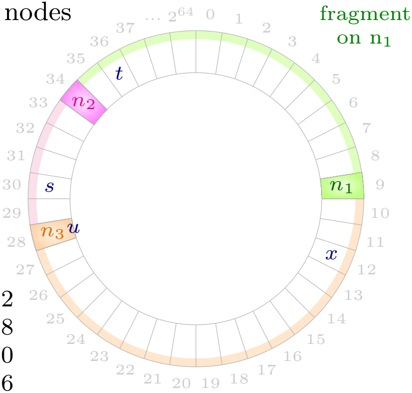
- Hashing: general properties
- Families of hash functions
- Strategies to build hash tables
- Hashing extensions: Consistent hashing
    - A key-value store : Redis

Distributed hash tables: consistent hashing

≃Rendezvous hashing ≃ Highest Random Weight ≃ keyring

To avoid redistributing too many keys as a new server is added/removed.
Assign keys to servers based on random partitioning of hash space.
Figure assumes hash space is 0..2^{64} (2^{64} -1 would make more sense, actually)

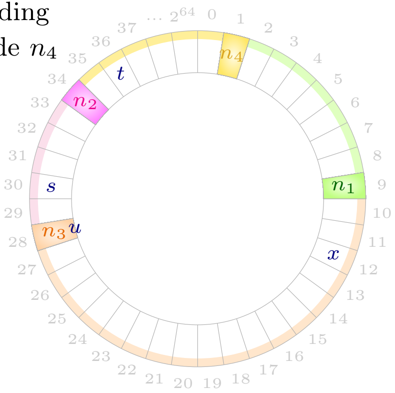
$N = 3$ nodes



h_0 :
$x \mapsto 12$
$u \mapsto 28$
$s \mapsto 30$
$t \mapsto 36$

Key t is stored on n_1, x and u on n_3...

Adding node n_4



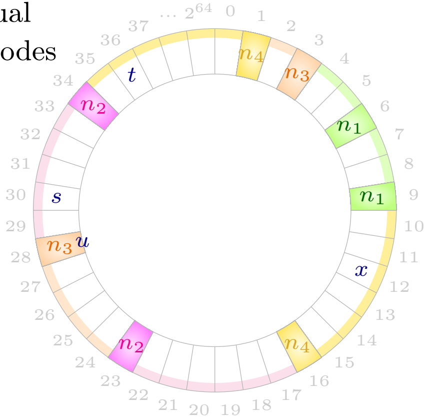
Key t is moved to the new node n_4.

Expected: $O(k/N)$ keys moved, with $k = \#$keys, $N = \#$servers.

Consistent hashing: improving repartition

Each server takes charge of multiple fragments on the ring.

Using virtual copies of nodes



Key t is moved to the new node n_4.

Reduces variance. Taking $k \simeq \log N$ copies "guarantees" reasonably balanced loads.

Consistent hashing: usage

- Teradata (1986)
- Akamai (1998)
- Chord (2001)
- Amazon's Dynamo (2007)
- Cassandra
- Discord...

To combine partitioning and replication, DynamoDB copies keys from a fragment in the next x fragments on the ring.
They also designed heuristics on how to build the fragments efficiently.

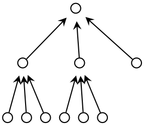
[https://www.microsoft.com/en-us/research/wp-content/uploads/2017/02/HRW98.pdf]

[https://www.akamai.com/us/en/multimedia/documents/technical-publication/
consistent-hashing-and-random-trees-distributed-caching-protocols-for-relieving-hot-spots-on-the-world-wide-w
pdf]

[https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf]

Consistent hashing for CDN

P: set of pages
C: set of available servers (n servers)



For each page:

- 1 balanced d-ary tree with n nodes
- a mapping $h : P \times \{1, \ldots, n\} \to C$ (consistent hashing).
- the root of each tree is mapped by h to *the* server that stores the page

For each page request:

1. *Browser:* picks a random leaf-to-root path, maps this path to a sequence of machines using h. Sends a request to the leaf that contains: the sequence of machines, the page request, and the browser id.
2. *Cache machine:* if it is caching the page, returns it. Otherwise increments a counter for the page, then asks the next machine and returns the page to browser when he gets it. When the counter for a page exceeds some threshold, caches a copy of the page.

[https://www.akamai.com/us/en/multimedia/documents/technical-publication/consistent-hashing-and-random-trees-distributed-caching-protocols-for-relieving-hot-spots-on-the-world-wide-web-technical-publication.pdf]

Key-value store

- Essentially a simple dictionnary (key-value pairs, access via the key). Core operations (other depend on store):
  - put(k,v) (*set* in Redis)
  - get(k)
  - delete(k).

  Very fast for lookup : data indexed by key.
- And, like most NoSQL systems :
  - Can generally be distributed.
  - (Very) scalable.
  - Interaction through API rather than query language.
  - Schema-on-read (values need not follow same schema)

Examples : Oracle BerkeleyDB, Riak, Amazon DynamoDB, LevelDB, Memcached, Redis.

Redis

(Remote Dictionary Server)

- Key-value store ("in-memory data structure store")
- Open-source, written in C, 1st version in 2009 by Salvatore Sanfilippo [http://antirez.com/latest/0]. Now open source project backed by *Redis Ltd.*.
- Applications : cache, real-time storage. Possibly also : streaming engine, message broker, database
- In-memory (though it can persist writes in log or periodic dump).
- Used in : stackoverflow GitHub ...
- Interaction with a redis server via a command-line interface : redis-cli. Or through clients which have been developped for most common languages. Ex: redis-py for Python.

Redis (open source) features

- Client-Server architecture, TCP connection (implemented its own event library)
- Mostly single-thread execution (multithread network from Redis 6.0)
- Features :
  - Some transactions (can group operations)
  - Replication
  - Sharding
  - Server-side Lua scripts (like stored procedure)
  - PUB/SUB
  - "Tracking" (Client-side caching, with server sending invalidation messages)

Redis : data model

Stores a collection of key-value pairs.
The key is any binary sequence (String). Typically :

- not too long (it's more efficient to use a hash for large keys)
- informative, and following a schema. Ex : `"user:1000"` or `"comment:4321:reply-to"`

The value is a Core redis data type, or an extension provided by a module or UDF (server-side function). Core data types:

- String (an array of byte/char), by default max size is 512MB.
- Containers of strings:
  - Lists (doubly linked, by insertion order)
  - Sets
  - Sorted sets (strings with score)
  - Hashes (a hash table)
- Others : streams, geospatial, HLL, bitmaps, bitfield.

Redis : String operations

Most operations are O(1):

```
> SET user:1 salvatore
OK
> GET user:1
"salvatore"


> SET ticket:27 "\"{'username': 'priya', 'ticket_id': 321}\"" EX 100

> INCRBY views:page:2 10
(integer) 10
```

[https://redis.io/docs/data-types/strings/]

Redis : Hash operations

Most operations are O(1):

```
> HSET user:123 username martina firstName Martina lastName Elisa country GB
(integer) 4
> HGET user:123 username
"martina"
> HGETALL user:123
1) "username"
2) "martina"
...
8) "GB"
```

[https://redis.io/docs/data-types/hashes/]

Redis Cluster : sharding

Design : high perf. (scales to 1000 nodes) > best-effort write safety > availability

- Client-Server, TCP connection (Redis implements its own event library)
- Cluster nodes connected to each other (full mesh) through a TCP cluster bus (gossip protocol/heartbeats to discover new nodes/detect failures).
- Redis assigns keys to nodes based hash slot (fixed hash function. Slot may contain several k-v pairs).
- Each master (and its replicas) is responsible for a range of "hash slots".
- No proxys : clients can contact any server (included replicas) and the reply indicates where to redirect query if needed
- Asynchronous replication (roughly at the same time as acknowledgement of a write). Rule for merging conflicts is : last failover wins.
- Replicas are not attached to a single master. If master fails, replica replaces master. When a master ends up without replica, a replica migrates from a master with multiple replicas (the master that has most replicas) to an orphaned master.
- After failure/split the part of cluster that has a minority of masters eventually won't accept writes, but may cause lost writes in the meantime.

Redis : beyond Redis open source

Modules many dynamic libraries (*modules*), mostly proprietary with source available (Redis Source Available License) :

- Data models (Json datatypes, graph database, time series...)
- Indexing (secondary indexes, full text search, bloom filters, cuckoo filters, spatial indexes)
- AI (NN...)

Redis Stack extends Redis with other data models and processing tools (json, graph, time series). Bundles the corresponding modules, Redis Insight (GUI for visualization).

Redis Enterprise commercial, available on premises or deployed on AWS, GCP, Azure (possibly multicloud). Improves scalability, availability, security features, support...

Mostly developed by *Redis*.

References on Redis

- `https://redis.io/docs/`
- `http://b3d.bdpedia.fr/redis.html`
- `https://sebastien.combefis.be/files/ecam/nosql/ECAM-NoSQL4MIN-Cours2-Slides.pdf`
- `https://cs.brown.edu/courses/cs227/archives/2011/slides/mar07-redis.pdf` (outdated)