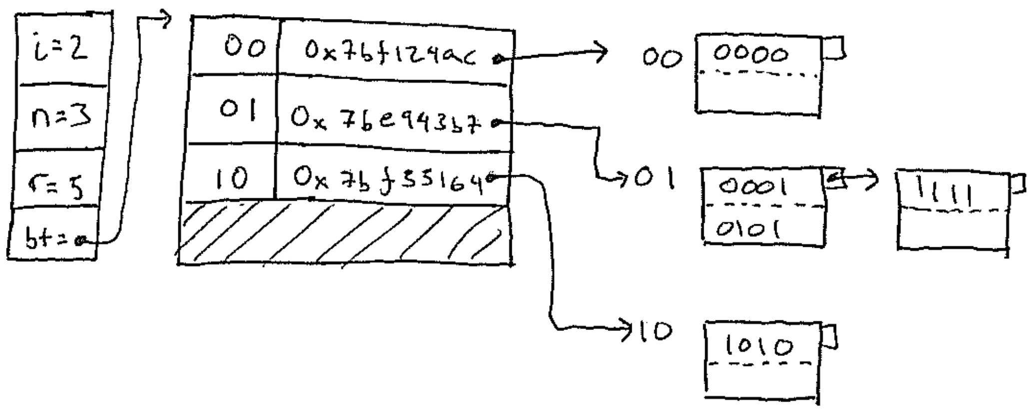
Linear Virtual Hashing

by. Tim Henderson (tadh@case.edu).
hackthology.com github.com/timtadh



Refrences

Database Systems - The Complete Book
by H. Garcia-Molina, J.D. Ullman, and J. Widom (2002)

Linear Hashing: a new tool for file and table addressing
by Litwin, witold
in Very Large Databases (VLDB)
Volume 6, Pages 212-223
1980

Classical Hashing - A review.

I expect all of you to be familiar with the classic hashtable. However, some of the finer points may be fuzzy. Let's review!

ADT:

- size(): int - how many entries?
- has (key: Hashable): boolean - is the key in the table?
- get (key: Hashable): Object throws KeyNotFound
- put (key: Hashable, value: Object)
- rm (key: Hashable) throws KeyNotFound.

Hashable

- hash(): int

insert

key → Hash() → <u>hash % table-size</u>

table

if spot not occupied put the value.
else ?

Search

key → hash() → <u>hash % table-size</u>

remove

Key → Hash() → <u>hash % table-size</u>

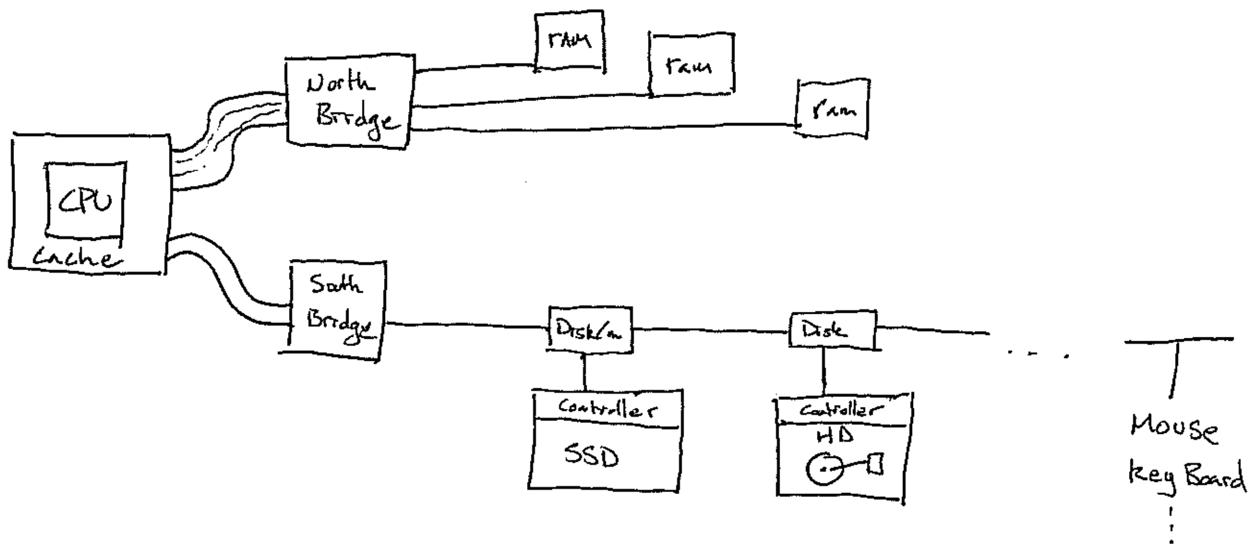
resize

1. Double Table size by allocating a new array.
2. Copy all entries from older array to the new one
    a. Scan old table to get each entry
    b. entries must be rehashed

Properties

1. Ammortized (on average) lookup, insert, remove Costs of O(1).
2. lots of wasted space. Experience suggests expansion at .6 utilization.
3. All entries must be copied on expansion
4. great for in memory lookup tables
5. Terrible for secondary memory tables. Why?

Secondary Storage, The Rules Change



1. Secondary Storage is slower as a medium
2. The Bus is slower
3. Many Peripherals hang of the South Bridge
4. Depending on interface, they may be daisy chained
   ↳ bus contention.

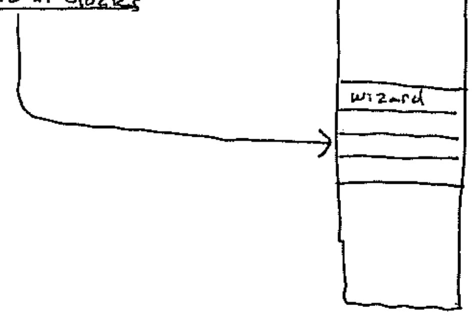
To deal with all these factors

1. We read and write pages ~~that~~ which are Blocks of size 4096 bytes.
2. In a good world we would read contiguous runs and write back individual Blocks.
3. Writes are batched.
4. You don't "read a byte", you ~~ge~~ read several Blocks and get just the byte you want.
5. You employ caching heavily.
6. Data-Structures are now measured in terms of # of Disk Accesses. (Block reads/writes).

Let's go Back To Hashing -

First Adjustment : Let's hash into Blocks

Key = "wizard"
↓
Hash()
↓
hash = 0x1817b1ae → hash % # Blocks



Blocks are sorted Arrays of keys/Values. So it takes. lg(BlockSize/RecordSize) to get an item.

But ... how do we expand?

1. The classic "Doubling" Scheme no longer looks so good.
   a. Allocates a huge amount of space
   b. probably writes to all of it, assuming a good hash function,
2. We can't just "add" blocks
3. We need to be able to still find old records.

Solution - Linear Hashing.

note. There are other systems but LH is the best.

Linear Hashing - Properties.

1. Mean Accesses per Lookup

	Mean Accesses per Lookup	unsuccessful ~~insert~~ lookup	insert
Utilization	.6 ~ 1.03	.75 ~ 1.27	.75 ~ 2.62
	.9 ~ 1.35	.9 ~ 2.37	.9 ~ 3.73

2. Grows at a linear Rate
   at most 2 new blocks created on insert.

3. Requires Little Dynamic re-arrangement.
   only on block split and only 1 blocks worth of records.

4. Dynamically shrinks and grows

5. Does not need address translation (necessarily)
   when translation is used it becomes "virtual"

6. In Comparison a b+ Tree of reasonable size may need at least 4 disk accesses per random lookup. (of course a b+ tree will perform must better for a range scan).

7. Simple Algorithm. Esp. compared to B+ Trees.

Linear Hashing.

1. <u>Key</u> Insight : Incrementally use more bits of the hash function $H(\cdot)$

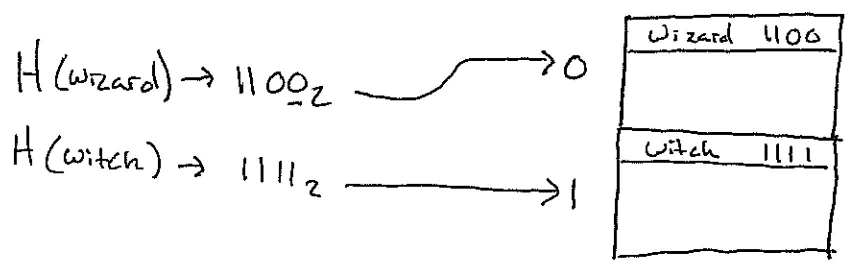
   This allows us to grow the table by almost "just adding a block"

   ex.

   $H(\text{wizard}) \rightarrow 1100_2$

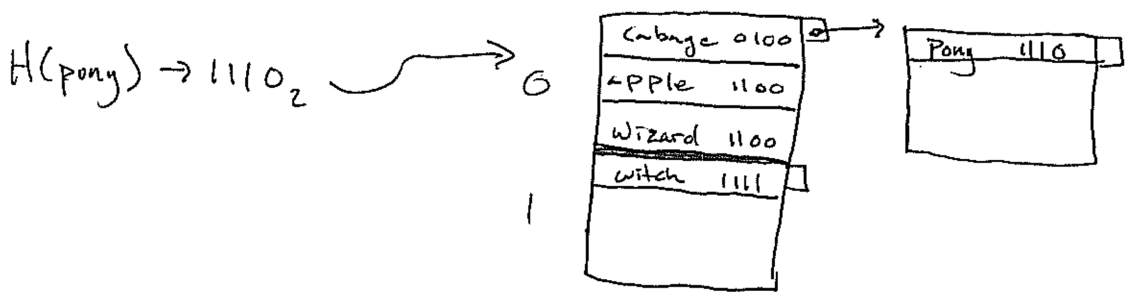
   We will use $\lg(n)$ bits of $H(\cdot)$ where n is the # of Blocks in the file.

File $n=2$ $i=1$ $r=2$



2. When a ~~bucket~~ Block is full chain another Block

File $n=2$ $i=1$ $r=5$



3. Split when record > UTILIZATION * # Buckets * # Records Per Block

   $5 > .8 \cdot 2 \cdot 3$

   $5 > 4.8$

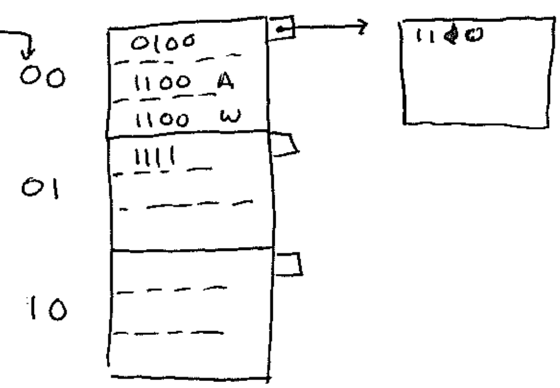
   So we need to split but how do we do that?

Splitting

First we Add a Bucket

File $n=3$ $i=2$ $r=5$

A bit more of the hash function is now used.



The bucket we Added is

$1a_i = 10 = a_1 a_2$

The bucke we need to split is

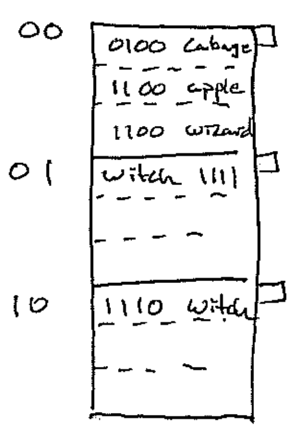
$0a_i$

Why?

In general if we add $1a_2 \dots a_i$ we split $0a_2 \dots a_i$

The split bucket is thus 00. After the split we get...

File $n=3$ $i=2$ $r=5$



Linear Hashing Algorithm Summary.

Insert(key)

```
hash = Hash(key)
bkt_idx = let
      hash = xxxx a_1 a_2 ... a_i
      m = a_1 ... a_i 2
   in
      if m < n then
         m
      else
         m - 2^(i-1)   = 0 a_2 ... a_i
bkt = get_bucket (bkt_idx)
bkt.Put (key, value)
r++
if r > UTILIZATION • n • (records per Block) then
      split().
```

We use this algorithm everytime we need to compute the bucket index let's call it bucket_idx().

Get (key)

```
hash = Hash (key)
bkt_idx = bucket_idx(hash)
bkt = get_bucket(bkt_idx)
return bkt.Get(key).
```

Remove (key)

```
hash = Hash(key)
bkt_idx = bucket_idx (hash)
bkt = get_bucket (bkt_idx)
~~Return~~ bkt.Remove(key)
r--
```

Split()

```
bkt_idx = n % (1 << (i-1))
bkt_a = get_bucket( bkt_idx)
bkt_b = alloc()
n++
if n > (1 << i) then
      i++
for each entry in bkt_a
      Put entry in bkt_b if a_1 = 1 in key.
```