Linear Virtual Hashing

by Tim Henderson (tadh@case.edu)

hackthology.com  github.com/timtadh

References

Database Systems - The Complete Book

Linear Hashing: a new tool for file and table addressing
by Litwin, w.told
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 hash table. 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 on the table?

**get (key: Hashable)**: Object throws KeyNotFoundException

**put (key: Hashable, value: Object)**

**rm (key: Hashable)** throws KeyNotFoundException.

**Hashable**

**hash()**: int

**insert**

```
Key \rightarrow Hash() \rightarrow \text{hash} \% \text{table.size} \rightarrow \text{table}
```

**Search**

```
key \rightarrow \text{hash()} \rightarrow \text{hash} \% \text{table.size}
```
Remove

Key → Hash(·) → hash % table-size

Resize

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

Properties

1. Amortized (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?
1. Secondary Storage is slower as a medium.
2. The Bus is slower.
3. Many Peripherals hang off the South Bridge.
4. Depending on interface, they may be daisy chained to bus controller.

To deal with all these factors:

1. We read and write pages 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 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"

\[ \text{Hash} = \text{Hash}(\text{Key}) \]

\[ \text{hash} = 0x1817612e \rightarrow \text{hash mod # Blocks} \]

Blocks are sorted arrays of Keys/Values. So it takes \( \log (\text{BlockSize/RecordSize}) \) to get an item.

But... how do we expand?

1. The classic "Doubling" scheme no longer works 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 LHT is the best.
Linear Hashing - Properties.

1. Mean Accesses per Lookup  
   Unsuccessful insert (lookup)  
   Utilization  \( n \approx 1.03 \)  \( n \approx 1.27 \)  \( n \approx 2.62 \)  \( n \approx 3.73 \)

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 block's 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. E.g. compared to B+ Trees.
Linear Hashing.

1. **Key Insight:** Incrementally use more bits of the hash function $H(.)$.
   This allows us to grow the table by almost "just adding a block." 
   
   \[ H(\text{wizard}) \rightarrow 1102 \]
   
   We will use $\log_2(n)$ bits of $H(.)$ where $n$ is the # of blocks in the file.

   \[
   \text{File } n = 2 \quad i = 1 \quad r = 2
   \]

   \[
   H(\text{wizard}) \rightarrow 1102 \quad \rightarrow 0
   \]

   \[
   H(\text{witch}) \rightarrow 1112 \quad \rightarrow 1
   \]

2. **When a bucket is full, chain another block.**

   \[
   H(\text{pony}) \rightarrow 1110 \quad \rightarrow 6
   \]

   \[
   \text{File } n = 2 \quad i = 1 \quad r = 5
   \]

   \[
   \text{Table: cabbage 0100, apple 1100, wizard 1100, witch 1111}
   \]

3. **Split when record $\geq$ UTILIZATION * # Buckets * # Records for Block.**

   \[ 5 \geq 0.8 \times 2 \times 3 \quad \text{so we need to split} \]

   \[ 5 \geq 4.8 \quad \text{but how do we do that?} \]
Splitting

First we add a bucket.

A bit of the hash function is now used.

The bucket we added is

\[ 1a_2 = 10 = a_1, a_2 \]

The bucket we need to split is

\[ 0a_2 \]

Why?

In general if we add \( 1a_2 \ldots a_i \) we split \( 0a_2 \ldots a_i \).

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

File \( n = 3 \ i = 2 \ r = 5 \)

\[
\begin{array}{c}
00 \\
1100 \\
1100 \\
1100 \\
01 \\
1111 \\
1111 \\
1111 \\
10 \\
1110 \end{array}
\]
Linear Hashing Algorithm Summary.

**Insert(key)**

\[ \text{hash} = \text{Hash(key)} \]
\[ \text{bkt-idx} = \text{let} \]
\[ \text{hash} = x \times x \times a_1 a_2 \ldots a_i \]
\[ m = a_1 \ldots a_i \]
\[ \text{if } m < n \text{ then } \]
\[ \text{else} \]
\[ m - 2^i = 0 a_2 \ldots a_i \]
\[ \text{bkt} = \text{get-bucket(bkt-idx)} \]
\[ \text{bkt}.\text{Put(key, value)} \]
\[ r++ \]
\[ \text{if } r > \text{UTILIZATION} \times n \times \text{(records per block)} \text{ then } \]
\[ \text{split}() \]

**Get(key)**

\[ \text{hash} = \text{Hash(key)} \]
\[ \text{bkt-idx} = \text{bucket-idx(hash)} \]
\[ \text{bkt} = \text{get-bucket(bkt-idx)} \]
\[ \text{return } \text{bkt}.\text{Get(key)} \]

**Split()**

\[ \text{bkt-idx} = n \times (1 << (i - 1)) \]
\[ \text{bkt-a} = \text{get-bucket(bkt-idx)} \]
\[ \text{bkt-b} = \text{malloc}() \]
\[ n++ \]
\[ \text{if } n > (1 << i) \text{ then } \]
\[ i++ \]
\[ \text{for each entry in bkt-a} \]
\[ \text{put entry in bkt-b if } a_i = 1 \text{ in key} \]

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