Datenbanksysteme II:
Hashing

Ulf Leser
Content of this Lecture

- Hashing
- Extensible Hashing
- Linear Hashing
Sorting or Hashing

- Sorted or indexed files
  - Typically \( \log(n) \) IO for searching / deletions
  - Overhead for keeping order in file or in index
  - Low overhead (overflows) brings danger of degradation
  - Multiple orders require multiple indexes - multiple overhead
  - Good support for range queries

- Can we do better ... on average? ... under certain circumstances?

- Hash files
  - Can provide access in 1 IO
  - Support searching for multiple attributes (with some overhead)
  - Require notable overhead if table size changes considerably
  - Are bad at range queries
Hash Files

- Set of buckets ($\geq 1$ blocks) $B_0, \ldots, B_{m-1}$, $m > 1$
- Hash function $h(K) = \{0, \ldots, m-1\}$ on a set $K$ of values
- Hash table $H$ (bucket directory) of size $m$ with ptrs to $B_i$'s
- Hash files are structured according to one attribute only
Example

• Hash function on Name

\[ h(\text{Name}) = \begin{cases} 
0 & \text{if last character} \leq M \\
1 & \text{if last character} \geq N 
\end{cases} \]

Why last char?

Bucket 0

- Bond
- George
- Victoria

Bucket 1

- Adams
- Carter
- Truman
- Wilson
- Washington

Search “Adams”
1. \( h(\text{Adams}) = 1 \)
2. Bucket 1, Block 0?
   - Success

Search “Wilson”
1. \( h(\text{Wilson}) = 1 \)
2. Bucket 1, Block 0?
   - Bucket 1, Block 1?
   - Success

Search “Elisabeth”
1. \( h(\text{Elisabeth}) = 0 \)
2. Bucket 0, Block 0?
   - Failure
Efficiency of Hashing

• Given: \( n \) records, \( R \) records per block, \( m \) buckets
• Assume hash table is in main memory
• Average number of blocks per bucket: \( n / (m \times R) \)
  - Assuming a (perfect) uniformly distributing hash function
• Search
  - \( n / (m \times R) / 2 \) for successful search
  - \( n / (m \times R) \) for unsuccessful search
• Insert
  - \( n / (m \times R) \) if end of bucket cannot be accessed directly
  - \( n / (2m \times R) \) if free space in one of the bucket
• If \( |H| = m \) large enough and good hash function: 1 I/O
Problems with Hashing

- Hashing may degrade to sequential scans (heap file)
  - If number of buckets static and too small
  - If hash function produces large skew
- Extending hash range requires complete rehashing
  - But fine-grained hashing require many buckets from the start
- Hash Function
  - Examples: Modulo Function, Bit-Shifting
  - Desirable: uniform mapping of hash keys onto m
  - “Ideal” (i.e. uniform) mapping possible if data distribution and number of records are known in advance – which is unusual
  - Often, application-independent hash functions are used
    - Incorporating knowledge on expected distribution of keys
Content of this Lecture

• Hashing
• Extensible Hashing
• Linear Hashing
Extensible Hashing

• Hashing as described is a static index structure
  - Structure (buckets, hash function) is fixed once and never changed
• Better: Hash function should automatically adapt to size of data set and to distribution of values

• Principle idea
  - Hash function generates (long) bitstring
    • Should distribute values evenly on every position of bitstring
  - Use growing prefix of this bitstring as index in hash table
  - Size of prefix adapts to number of records
Hash functions

- \( h: K \rightarrow \{0,1\}^* \)
- Size of bitstring should be long enough for mapping into as many buckets as \textit{maximally desired}.
  - Though we do not use them all most of the time.
- Example: inverse person IDs
  - \( h(004) = 001000000... \quad (4=0..0100) \)
  - \( h(006) = 011000000... \quad (6=0..0110) \)
  - \( h(007) = 111000000... \quad (7 =0..0111) \)
  - \( h(013) = 101100000... \quad (13 =0..01101) \)
  - \( h(018) = 010010000... \quad (18 =0..010010) \)
  - \( h(032) = 000001000... \quad (32 =0..0100000) \)
  - \( H(048) = 000011000... \quad (48 =0..0110000) \)
Example

- **Parameters**
  - d: global „depth“ of hash table, *size of longest prefix currently used*
  - t: local „depth“ of each bucket, *size of prefix used in this bucket*

- **Example**
  - Let a bucket store two records
  - Start with two buckets and 1 bit for identification (d=t₁=t₂=1)

<table>
<thead>
<tr>
<th>Keys</th>
<th>as bitstring</th>
<th>inverse</th>
<th>( h_{d=1}(k) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2125</td>
<td>100001001101</td>
<td>101100100001</td>
<td>1</td>
</tr>
<tr>
<td>2126</td>
<td>100001001110</td>
<td>011100100001</td>
<td>0</td>
</tr>
<tr>
<td>2127</td>
<td>100001001111</td>
<td>111100100001</td>
<td>1</td>
</tr>
</tbody>
</table>

0

1

\( (2126, \text{’Russel’}, \text{’C4’}, 232) \)

\( (2125, \text{’Sokrates’}, \text{’C4’}, 226) \)

\( (2127, \text{’Kopernikus’}, \text{’C3’}, 310) \)
Example cont’d

- New record with $x=2129$
- Bucket for "1" full
- Needs to be split
- Double hash table, $d++$
- Pointers to non-splitting blocks remain unchanged
- Block is split and records are distributed according to bits until new $d$

<table>
<thead>
<tr>
<th>$k$</th>
<th>as bitstring</th>
<th>inverse</th>
<th>$h_{d=1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2125</td>
<td>100001001101</td>
<td>101100100001</td>
<td>1</td>
</tr>
<tr>
<td>2126</td>
<td>100001001110</td>
<td>011100100001</td>
<td>0</td>
</tr>
<tr>
<td>2127</td>
<td>100001001111</td>
<td>111100100001</td>
<td>1</td>
</tr>
<tr>
<td>2129</td>
<td>100001010001</td>
<td>100010100001</td>
<td>1</td>
</tr>
</tbody>
</table>
More Complex Example

- Assume reversed bit hash function on integers
- Currently four buckets in use
- Global depth \( d = 3 \)
- Local depth \( t \) between 1 and 3
- Size of global directory: \( 2^d = 8 \)
Example: Hash Table
Inserting Values

Current content
40 = 101000
32 = 100000
18 = 010010
13 = 001101
12 = 001100
7 = 000111
6 = 000110
4 = 000100

000: 32, 40; t=3
001: 4, 12; t=3
01: 6, 18; t=2

INSERT(28)
- 28 = 011100
- h(28) = 001110

Overflow

7, 13; t=1

d = t;
Splitting Deep Buckets

- $h(12) = 001100$
- $h(4) = 001000$
- $h(28) = 001110$

- $12, 28; t=4$
- $32, 40; t=3$
- $4; t=4$
- $6, 18; t=2$
- $7, 13; t=1$
Next Insert

INSERT( 5)
- $5 = 000101$
- $h(5) = 101000$

Overflow but no dir duplication

$d \neq t$: Overflow but no dir duplication
Splitting Shallow Buckets

• Assume we split overflowed bucket B
• All records $r \in B$, $h(r)$ has the same length-$t$ prefix
• If we split at next position ($t++$)
  - Generate new bucket and rehash records
  - This might generate empty buckets
  - Other bucket might still be overflowed – repeat split
    • In the example, we rehash $5=101000$, $7=111000$, $13=101100$
    • Hence, one split suffices (with block prefixes 101 and 111)
    • But, if we had $5=10100$, $13=101100$, $21=101010$?
• Might even force a deep split with increase in $d$
• Suboptimal space usage
Summary

• Advantages
  - Partly adapts to growing or shrinking number of records
  - No rehashing of the entire table
  - Essentially no limit in number of records
  - Very fast if directory can be cached and h is well chosen

• Disadvantages
  - Directory needs to be maintained - requires LOG/REDO information, locking in split operations, storage, etc.
  - No adaptation of bucket fill degree - many buckets might be almost empty, few almost full
  - Directory doubling is a “unforeseen” costly operation
    • Everything smooth for a long time, suddenly one operation takes ages
  - Values are not sorted, no range queries
Content of this Lecture

- Hashing
- Extensible Hashing
- Linear Hashing
Linear Hashing

• Similar scheme as in extensible hashing, but
  - Don’t double directory on overflow, but increase one-by-one
  - Guaranteed lower bound on bucket fill-degree
  - Tolerate some overflow blocks in buckets
    • Hopefully few on average, if hash function spreads evenly
Overview

- $h$ generates bitstring of length $x$, read right to left
- Parameters
  - $i$: Current number of bits from $x$ used
    - As $i$ grows, more bits are considered
    - If $h$ generates $x$ bits, we use $a_1a_2...a_i$ for the last $i$ bits of $h(k)$
  - $n$: Total number of buckets currently used
    - Only the first $n$ values of bitstrings of length $i$ have their own buckets
    - Sometimes, $i$ must be increased - later
  - $r$: Total number of records
- Fix threshold $t$ - linear hashing guarantees that $r/n < t$
  - As $r$ increases, we sometimes increase $n$ such that always $r/n < t$
  - Linear hashing guarantees average fill-degree
    - But does not prevent chaining in case of “bad” hash function

<table>
<thead>
<tr>
<th>011101010110</th>
</tr>
</thead>
<tbody>
<tr>
<td>grows</td>
</tr>
<tr>
<td>$i$</td>
</tr>
<tr>
<td>$x$</td>
</tr>
<tr>
<td>22</td>
</tr>
</tbody>
</table>
Insert(k): First Action

- **Insert k**
  - Let m by the integer value encoded by the i last bits of h(k)
  - If m < n
    - Store k in bucket m, potentially using overflow blocks
    - Obviously, we store k in a bucket that does exist
  - If m ≥ n
    - Bucket m does not exist
      - There exist buckets 0 … n-1
    - We redirect k in a bucket that does exist
    - Flip i-th bit (from the right) of m to 0 and store k in this bucket
      - Algorithm ensures that here the i’th (highest) bit must be 1
    - This flipping also needs to be done when searching keys
Insert( k): Second Action

• **Check threshold**: if \( r/n \geq t \), then
  - If \( n = 2^i \)
    • No more room to add another bucket
    • Set \( i++ \)
    • This is only a *conceptual increase* - no physical action
    • Proceed (now \( n < 2^i \))
  - If \( n < 2^i \)
    • There is still/now *room on our address space*
    • We add \((n+1)\)th bucket and set \( n++ \)
    • We need to choose *which bucket to split*
      - We do not split the bucket where we just inserted (why should we?)
      - We do not search for overflowed buckets (too costly)
      - Instead, we follow a cyclic scheme
Which Bucket to Split

• We split **buckets in fixed, cyclic order**
• Split bucket with number $n-2^{i-1}$
  - As $n$ increases, this **pointer cycles through all buckets**
  - Let $n=1a_2a_3...a_i$; then we split block $a_2a_3...a_i$ into $0a_2a_3...a_i$ and $1a_2a_3...a_i$
    • Requires redistribution of bucket with key $a_2a_3...a_i$
    • This is one of the buckets where we had put redirected records with $h(k)>n$
    • This is **not necessarily an overflowed bucket**
    • Only total fill degree is guaranteed
### Buckets are Split in Fixed Order

Assume we would split after every insert

<table>
<thead>
<tr>
<th>i</th>
<th>n</th>
<th>Existing buckets</th>
<th>Bucket to split</th>
<th>Generates</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2=10</td>
<td>0,1</td>
<td>0</td>
<td>00, 10</td>
</tr>
<tr>
<td>2</td>
<td>3=11</td>
<td>00,10</td>
<td>1</td>
<td>01, 11</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>00,10 01,11</td>
<td>00</td>
<td>000, 100</td>
</tr>
<tr>
<td>3</td>
<td>5=101</td>
<td>000,100 10, 01,11</td>
<td>01</td>
<td>001, 101</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>000,100 001,101 10,11</td>
<td>10</td>
<td>010, 110</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>000,100,001,101, 010,110, 11</td>
<td>11</td>
<td>011, 111</td>
</tr>
</tbody>
</table>
Example

- Assume 2 records in one block, \( x=4 \), \( t=1.49 \), \( i=1 \)

Start

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>1010</td>
</tr>
<tr>
<td>1</td>
<td>1111</td>
<td></td>
</tr>
</tbody>
</table>

1a) Insert 0101
\( m=1 < n=10_b \)
Insert into bucket 1
But now \( r/n > t \)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>1010</td>
</tr>
<tr>
<td>1</td>
<td>1111</td>
<td>0101</td>
</tr>
</tbody>
</table>

1b) Since \( n=2^i=2=10_b \)
We need more address space
Increase \( i \) (virtually)
Add bucket number \( 2=10_b \)
n=10_b=1a_1: Split bucket 0 into 10 and 00

Split
Yet unsplit
Split

\( n+1 \)
Example 2

2) Insert 0001
   m=1, bucket exists
   Insert into m
   Requires **overflow block**
   (We skip the necessary split)

```
00 0000
01 1111 0001
10 1010
```

3a) Insert 0111
   m=3=n=11b
   Bucket doesn’t exist
   Flip and redirect to 01

```
00 0000
01 1111 0001
01 1111 1010 0111
10 1010
```

3b) r/n=6/3>t – We split
   n<4, so no need to increase i
   Add bucket number 3=11b
   Since n=3=11b, with split 01
   Delete overflow block

```
00 0000
01 1111 0001
01 0001 0101
10 1010
11 1111 0111
```
Example 3

4a) Insert 0011
   \( m = 3 = 11_b < n = 4 = 100_b \)
   Insert into \( 11_b \)

\[
\begin{array}{|c|c|}
\hline
00 & 0000 \\
\hline
01 & 0001 \\
& 0101 \\
\hline
10 & 1010 \\
\hline
11 & 1111 \\
& 0111 \\
\hline
\end{array}
\]

4b) We must split again
   Since \( n = 2^i \), increase \( i \)
   Nothing to do physically
   ("Think" a leading 0)
Example 4

4c) Split
Add block number $4 = 100_b$
Split $000_b$ into $000_b$ and $100_b$

<table>
<thead>
<tr>
<th>000</th>
<th>0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>0001</td>
</tr>
<tr>
<td></td>
<td>0101</td>
</tr>
<tr>
<td>010</td>
<td>1010</td>
</tr>
<tr>
<td>011</td>
<td>1111</td>
</tr>
<tr>
<td></td>
<td>0111</td>
</tr>
<tr>
<td>100</td>
<td>-</td>
</tr>
</tbody>
</table>

We keep the average bucket filling
But we have unevenly filled buckets – some empty, some overflow
Observations

• Due to the extension mechanism: $2^{i-1} \leq n < 2^i$
  - Whenever $n$ reaches $2^i$, $i$ is increased $\Rightarrow$ $2^i$ doubles and $n=2^i/2$ (for the new $i$)
  - $n$ as binary number always has the form $1b_1b_2...b_{i-1}$

• As defined: $m<2^i$
  - But possibly: $m>n$
    • If we drop the leading 1 in $m$, we get $m_{\text{new}}<m/2$
    • Since $n>2^{i-1}$, $m_{\text{new}} \leq n$
    • Thus, the chosen bucket must already exist
    • We don’t know when it will be split
Summary

- **Advantages**
  - Adapts to varying number of records
  - *Slower growth* and on average better space usage compared to extensible hashing
  - If buckets are sequential on disk, we **don’t need a directory**
    - Compute m: look in m’th block (possible after flipping)

- **Disadvantages**
  - Can degrade, as buckets are split in fixed order
  - No adaptation to skewed value distribution