Chapter 6
Hash-Based Indexing
Efficient Support for Equality Search

Architecture and Implementation of Database Systems
Summer 2014

Torsten Grust
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen
Hash-Based Indexing

- We now turn to a different family of index structures: hash indexes.
- Hash indexes are “unbeatable” when it comes to support for equality selections:

```sql
1 SELECT *
2 FROM R
3 WHERE A = k
```

- Further, other query operations internally generate a flood of equality tests (e.g., nested-loop join). (Non-)presence of hash index support can make a real difference in such scenarios.
Hashing vs. $B^+$-trees

- Hash indexes provide **no support for range queries**, however (hash indexes are also known as scatter storage).
- In a $B^+$-tree-world, to locate a record with key $k$ means to **compare** $k$ with other keys $k'$ organized in a (tree-shaped) search data structure.
- Hash indexes **use the bits of $k$ itself** (independent of all other stored records) to find the location of the associated record.
- We will now briefly look into **static hashing** to illustrate the basics.
  - Static hashing does **not** handle updates well (much like ISAM).
  - Later, we introduce **extendible hashing** and **linear hashing** which refine the hashing principle and adapt well to record insertions and deletions.
Static Hashing

- To build a **static hash index** on attribute A:

  **Build static hash index on column A**

1. Allocate a fixed area of $N$ (successive) disk pages, the so-called **primary buckets**.
2. In each bucket, install a pointer to a chain of **overflow pages** (initially set the pointer to **null**).
3. Define a **hash function** $h$ with range $[0, \ldots, N - 1]$. The **domain** of $h$ is the type of A, e.g.

   $$ h : \text{INTEGER} \rightarrow [0, \ldots, N - 1] $$

   If A is of SQL type INTEGER.
Static Hashing

• A primary bucket and its associated chain of overflow pages is referred to as a bucket (above).

• Each bucket contains index entries $k*$ (implemented using any of the variants A, B, C, see slide 2.22.)
Static Hashing

- To perform $h\text{search}(k)$ (or $\text{hinsert}(k)/\text{hdelete}(k)$) for a record with key $A = k$:

**Static hashing scheme**

1. **Apply hash function** $h$ to the key value, *i.e.*, compute $h(k)$.
2. **Access the primary bucket page** with number $h(k)$.
3. Search (insert/delete) subject record on this page or, if required, **access the overflow chain** of bucket $h(k)$.

- If the hashing scheme works well and overflow chain access is avoidable,
  - $h\text{search}(k)$ requires a **single I/O operation**,
  - $\text{hinsert}(k)/\text{hdelete}(k)$ require **two I/O operations**.
Static Hashing: Collisions and Overflow Chains

- At least for static hashing, **overflow chain management** is important.
- Generally, we do not want hash function \( h \) to avoid **collisions**, \( i.e., \)

\[
h(k) = h(k') \quad \text{even if} \quad k \neq k'
\]

(otherwise we would need as many primary bucket pages as different key values in the data file).

- At the same time, we want \( h \) to **scatter** the key attribute domain **evenly** across \([0, \ldots, N - 1]\) to avoid the development of long overflow chains for few buckets. This makes the hash tables’ I/O behavior non-uniform and unpredictable for a query optimizer.

- Such “good” hash functions are hard to discover, unfortunately.
The Birthday Paradox (Need for Overflow Chain Management)

Example (The birthday paradox)

Consider the people in a group as the domain and use their birthday as hash function \( h \) (\( h \) : Person \( \rightarrow \) \([0, \ldots, 364]\)).

If the group has 23 or more members, chances are > 50 % that two people share the same birthday (collision).

Check: Compute the probability that \( n \) people all have different birthdays:

Function: different_birthday (\( n \))

1. if \( n = 1 \) then
2. return 1;
3. else
4. return \( \text{different}_\text{birthday}(n - 1) \times \frac{365 - (n - 1)}{365} \);  
5. \( \text{probability that } n - 1 \text{ persons have different birthdays} \times \frac{365}{365} \);  
6. \( \text{probability that } n \text{th person has birthday different from first } n - 1 \text{ persons} \);
Hash Functions

- It is impossible to generate truly random hash values from the non-random key values found in actual tables. Can we define hash functions that scatter even better than a random function?

**Hash function**

1. **By division.** Simply define

   \[ h(k) = k \mod N \]

   This guarantees the range of \( h(k) \) to be \([0, \ldots, N - 1]\).

   **Note:** Choosing \( N = 2^d \) for some \( d \) effectively considers the least \( d \) bits of \( k \) only. **Prime numbers** work best for \( N \).

2. **By multiplication.** Extract the fractional part of \( Z \cdot k \) (for a specific \( Z^1 \)) and multiply by arbitrary hash table size \( N \):

   \[ h(k) = \lfloor N \cdot (Z \cdot k - \lfloor Z \cdot k \rfloor) \rfloor \]

   \( ^1 \)The (inverse) **golden ratio** \( Z = \frac{2}{\sqrt{5}+1} \approx 0.6180339887 \) is a good choice. See D.E.Knuth, “*Sorting and Searching.*”
Static Hashing and Dynamic Files

- For a static hashing scheme:
  - If the underlying **data file grows**, the development of overflow chains spoils the otherwise predictable behavior hash I/O behavior (1–2 I/O operations).
  - If the underlying **data file shrinks**, a significant fraction of primary hash buckets may be (almost) empty—a waste of page space.
  - As in the ISAM case, however, static hashing has advantages when it comes to concurrent access.
  - We may periodically **rehash** the data file to restore the ideal situation (20 % free space, no overflow chains).

⇒ Expensive and the index cannot be used while rehashing is in progress.
Extendible Hashing

- **Extendible Hashing** can adapt to growing (or shrinking) data files.
- To keep track of the actual primary buckets that are part of the current hash table, we hash via an in-memory bucket directory:

---

**Example (Extendible hash table setup; ignore the 2 fields for now)**

![Diagram of Extendible Hashing]

Note: This figure depicts the entries as $h(k)\ast$, not $k\ast$. 

\[2\]
Extendible Hashing: Search

Search for a record with key $k$

1. Apply $h$, i.e., compute $h(k)$.
2. Consider the last 2 bits of $h(k)$ and follow the corresponding directory pointer to find the bucket.

Example (Search for a record)

To find a record with key $k$ such that $h(k) = 5 = 101_2$, follow the second directory pointer ($101_2 \land 11_2 = 01_2$) to bucket B, then use entry 5* to access the wanted record.
Global and local depth annotations

- **Global depth** ($n$ at hash directory): *Use the last $n$ bits of $h(k)$ to lookup a bucket pointer in the directory* (the directory size is $2^n$).
Extendible Hashing: Global and Local Depth

Global and local depth annotations

- **Global depth** (at hash directory): 
  *Use the last $n$ bits of $h(k)$ to lookup a bucket pointer in the directory* (the directory size is $2^n$).

- **Local depth** (at individual buckets): 
  *The hash values $h(k)$ of all entries in this bucket agree on their last $d$ bits.*
Extendible Hashing: Insert

Insert record with key $k$

1. Apply $h$, i.e., compute $h(k)$.
2. Use the last $n$ bits of $h(k)$ to lookup the bucket pointer in the directory.
3. If the primary bucket still has capacity, store $h(k)^*$ in it. (Otherwise ... ?)

Example (Insert record with $h(k) = 13 = 1101_2$)

```
            2
           4* 12 32 16  bucket A
          00
          01
          10
          11
         2
        directory
           h

            2
           1* 5* 21 13*  bucket B
          1 0* 15* 7* 19*
        10* bucket C

            2
           10*  bucket D
          2
```

hash table
Extendible Hashing: Insert, Bucket Split

**Example (Insert record with** \( h(k) = 20 = 10100_2 \))

Insertion of a record with \( h(k) = 20 = 10100_2 \) leads to **overflow in primary bucket** A. Initiate a **bucket split** for A.

1. **Split** bucket A (creating a new bucket A2) and use bit position \( d + 1 \) to redistribute the entries:

   \[
   \begin{align*}
   4 &= 100_2 \\
   12 &= 1100_2 \\
   32 &= 100000_2 \\
   16 &= 10000_2 \\
   20 &= 10100_2 \\
   \end{align*}
   \]

   ![Bucket Diagram]

   **Note:** We now need 3 bits to discriminate between the old bucket A and the new split bucket A2.
Extendible Hashing: Insert, Directory Doubling

Example (Insert record with \( h(k) = 20 = 10100_2 \))

1. *In the present case,* we need to **double the directory** by simply copying its original pages (we now use \( 2 + 1 = 3 \) bits to lookup a bucket pointer).
2. Let bucket pointer for \( _{100}_2 \) point to A2 (the directory pointer for \( _{000}_2 \) still points to bucket A):

```markdown
20\*  
19\*  
15\*  
13\*  
10\*  
5\*   
32\*  
16\*
```

```plaintext
3

bucket A

3

bucket A2

2

bucket B

2

bucket C

2

bucket D

000

001

010

011

100

101

110

111

directory

h
```
Extendible Hashing: Insert

If we split a bucket with local depth \( d < n \) (global depth), directory doubling is \textit{not} necessary:

**Example (Insert record with} \( h(k) = 9 = 1001_2 \))

- Insert record with key \( k \) such that \( h(k) = 9 = 1001_2 \).

- The associated bucket \( B \) is split, creating a new bucket \( B2 \). Entries are redistributed. New local depth of \( B \) and \( B2 \) is \( 3 \) and thus does \textit{not} exceed the global depth of \( 3 \).

\( \Rightarrow \) Modifying the directory’s bucket pointer for \( 101_2 \) is sufficient (see following slide).
Extendible Hashing: Insert

Example (After insertion of record with $h(k) = 9 = 1001_2$)
Extendible Hashing: Search Procedure

- The following \texttt{hsearch(\cdot)} and \texttt{hinsert(\cdot)} procedures operate over an in-memory array representation of the bucket directory $\text{bucket}[0, \ldots, 2^n - 1]$.

\begin{verbatim}
Extendible Hashing: Search

Function: \texttt{hsearch(k)}

1 $n \leftarrow \lfloor n \rfloor$;          /* global depth */
2 $b \leftarrow h(k) \& (2^n - 1)$;    /* mask all but the low $n$ bits */
3 \textbf{return} \text{bucket}[b];
\end{verbatim}
Extendible Hashing: Insert Procedure

Function: hinsert(k*)

1. \( n \leftarrow n; \) /* global depth */
2. \( b \leftarrow hsearch(k); \)
3. if \( b \) has capacity then
   4. Place \( k* \) in bucket \( b; \)
   5. return;
4. /* overflow in bucket \( b, \) need to split */
5. \( d \leftarrow d_b; \) /* local depth of hash bucket \( b \) */
6. Create a new empty bucket \( b2; \)
7. /* redistribute entries of \( b \) including \( k* \) */
8. :
Extendible Hashing: Insert Procedure (continued)

Extendible Hashing: Insertion (cont’d)

/* redistribute entries of b including k* */
foreach k* in bucket b do
    if h(k') & 2^d ≠ 0 then
        Move k' to bucket b2 ;
/* new local depths for buckets b and b2 */
d_b ← d + 1 ;
d_{b2} ← d + 1 ;
if n < d + 1 then
    /* we need to double the directory */
    Allocate 2^n new directory entries bucket[2^n, ..., 2^{n+1} − 1] ;
    Copy bucket[0, ..., 2^n − 1] into bucket[2^n, ..., 2^{n+1} − 1] ;
    n ← n + 1 ;
/* update the bucket directory to point to b2 */
    bucket[(h(k) & (2^n − 1)) | 2^n] ← addr(b2)
**Extendible Hashing: Overflow Chains? / Delete**

**Overflow chains?**

Extendible hashing uses overflow chains hanging off a bucket only as a resort. Under which circumstances will extendible hashing create an overflow chain?
Extendible Hashing: Overflow Chains? / Delete

**Overflow chains?**

Extendible hashing uses overflow chains hanging off a bucket only as a resort. Under which circumstances will extendible hashing create an overflow chain?

If considering $d + 1$ bits does *not* lead to satisfying record redistribution in procedure $hinsert(k)$ (skewed data, hash collisions).

- Deleting an entry $k*$ from a bucket may leave its bucket completely (or almost) empty.
- Extendible hashing then tries to **merge** the empty bucket and its associated partner bucket.

**Extendible hashing: deletion**

When is local depth decreased? When is global depth decreased? (Try to work out the details on your own.)
Linear Hashing

- **Linear hashing** can, just like extendible hashing, adapt its underlying data structure to record insertions and deletions:
  - Linear hashing **does not need a hash directory** in addition to the actual hash table buckets.
  - Linear hashing can define **flexible criteria that determine when a bucket is to be split**.
  - Linear hashing, however, may perform badly if the key distribution in the data file is **skewed**.

- We will now investigate linear hashing in detail and come back to the points above as we go along.

- The core idea behind linear hashing is to use an **ordered family of hash functions**, \( h_0, h_1, h_2, \ldots \) (traditionally the subscript is called the hash function’s **level**).
Linear Hashing: Hash Function Family

- We design the family so that the range of $h_{\text{level}+1}$ is twice as large as the range of $h_{\text{level}}$ (for level = 0, 1, 2, ...).

Example ($h_{\text{level}}$ with range $[0, \ldots, N - 1]$)
Linear Hashing: Hash Function Family

- Given an initial hash function $h$ and an initial hash table size $N$, one approach to define such a family of hash functions $h_0$, $h_1$, $h_2$, ... would be:

\[
    h_{level}(k) = h(k) \mod (2^{level} \cdot N) \quad (\text{level} = 0, 1, 2, \ldots)
\]

Hash function family
Linear Hashing: Basic Scheme

Basic linear hashing scheme

1. Initialize: $level \leftarrow 0$, $next \leftarrow 0$.

2. The current hash function in use for searches (insertions/deletions) is $h_{level}$, active hash table buckets are those in $h_{level}$’s range: $[0, \ldots, 2^{level} \cdot N - 1]$.

3. Whenever we realize that the current hash table overflows, e.g.,
   - insertions filled a primary bucket beyond $c$ % capacity,
   - or the overflow chain of a bucket grew longer than $p$ pages,
   - or ⟨insert your criterion here⟩

we split the bucket at hash table position $next$
(in general, this is not the bucket which triggered the split!)
Linear Hashing: Bucket Split

Linear hashing: bucket split

1. **Allocate a new bucket, append** it to the hash table (its position will be $2^{level} \cdot N + next$).

2. **Redistribute** the entries in bucket $next$ by **rehashing** them via $h_{level+1}$ (some entries will remain in bucket $next$, some go to bucket $2^{level} \cdot N + next$). For $next = 0$:

   ```
   h_{level+1}:
   .
   0 ← next
   2^{level} \cdot N - 1
   2^{level} \cdot N + next
   ```

3. **Increment** $next$ by 1.

   ⇒ **All buckets with positions < next** have been rehashed.
Linear Hashing: Rehashing

Searches need to take current next position into account

\[ h_{\text{level}}(k) \begin{cases} < \text{next} : \text{we hit an already split bucket, rehash} \\ \geq \text{next} : \text{we hit a yet unsplit bucket, bucket found} \end{cases} \]

Example (Current state of linear hashing scheme)

range of \( h_{\text{level}} \)

range of \( h_{\text{level}+1} \)

\( 2^{\text{level}} \cdot N - 1 \) hash buckets

buckets already split (\( h_{\text{level}+1} \))

next bucket to be split

unsplit buckets (\( h_{\text{level}} \))

images of already split buckets (\( h_{\text{level}+1} \))
Linear Hashing: Split Rounds

💡 When \( \text{next} \) is incremented beyond hash table size...?

A bucket split increments \( \text{next} \) by 1 to mark the next bucket to be split. How would you propose to handle the situation when \( \text{next} \) is incremented \textit{beyond} the last current hash table position, \textit{i.e.} \( \text{next} > 2^{\text{level}} \cdot N - 1? \)
Linear Hashing: Split Rounds

When next is incremented beyond hash table size...?

A bucket split increments next by 1 to mark the next bucket to be split. How would you propose to handle the situation when next is incremented beyond the last current hash table position, i.e.

\[ next > 2^{level} \cdot N - 1? \]

Answer:

• If \( next > 2^{level} \cdot N - 1 \), all buckets in the current hash table are hashed via function \( h_{level+1} \).

⇒ Proceed in a round-robin fashion:

If \( next > 2^{level} \cdot N - 1 \), then

1. increment level by 1,
2. \( next \leftarrow 0 \) (start splitting from hash table top again).

• In general, an overflowing bucket is not split immediately, but—due to round-robin splitting—no later than in the following round.
Linear Hashing: Running Example

Linear hash table setup:
- Bucket capacity of 4 entries, initial hash table size $N = 4$.
- Split criterion: allocation of a page in an overflow chain.

Example (Linear hash table, $h_{level}(k)$* shown)

<table>
<thead>
<tr>
<th>level = 0</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$h_1$</td>
<td>$h_0$</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>00</td>
<td>32* 44* 36*</td>
</tr>
<tr>
<td>001</td>
<td>01</td>
<td>9* 25* 5*</td>
</tr>
<tr>
<td>010</td>
<td>10</td>
<td>14* 18* 10* 30*</td>
</tr>
<tr>
<td>011</td>
<td>11</td>
<td>31* 35* 7* 11*</td>
</tr>
</tbody>
</table>

hash buckets overflow pages
Example (Insert record with key $k$ such that $h_0(k) = 43 = 101011_2$)

level = 0

<table>
<thead>
<tr>
<th>$h_1$</th>
<th>$h_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>00</td>
</tr>
<tr>
<td>001</td>
<td>01</td>
</tr>
<tr>
<td>010</td>
<td>10</td>
</tr>
<tr>
<td>011</td>
<td>11</td>
</tr>
<tr>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>

hash buckets overflow pages

next
Linear Hashing: Running Example

Example (Insert record with key $k$ such that $h_0(k) = 37 = 100101_2$)

<table>
<thead>
<tr>
<th>level = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h_1$</td>
</tr>
<tr>
<td>000 00</td>
</tr>
<tr>
<td>001 01</td>
</tr>
<tr>
<td>010 10</td>
</tr>
<tr>
<td>011 11</td>
</tr>
<tr>
<td>100</td>
</tr>
</tbody>
</table>

hash buckets overflow pages
Linear Hashing: Running Example

Example (Insert record with key $k$ such that $h_0(k) = 29 = 11101_2$)

level = 0

<table>
<thead>
<tr>
<th>$h_1$</th>
<th>$h_0$</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>00</td>
<td>32*</td>
</tr>
<tr>
<td>001</td>
<td>01</td>
<td>9*</td>
</tr>
<tr>
<td>010</td>
<td>10</td>
<td>14*</td>
</tr>
<tr>
<td>011</td>
<td>11</td>
<td>31*</td>
</tr>
<tr>
<td>100</td>
<td></td>
<td>44*</td>
</tr>
<tr>
<td>101</td>
<td></td>
<td>5*</td>
</tr>
</tbody>
</table>
Linear Hashing: Running Example

Example (Insert three records with key $k$ such that $h_0(k) = 22 = 10110_2 / 66 = 100010_2 / 34 = 10010_2$)

<table>
<thead>
<tr>
<th>h₀</th>
<th>h₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>00</td>
</tr>
<tr>
<td>001</td>
<td>01</td>
</tr>
<tr>
<td>010</td>
<td>10</td>
</tr>
<tr>
<td>011</td>
<td>11</td>
</tr>
<tr>
<td>100</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td></td>
</tr>
</tbody>
</table>

level = 0

next
Linear Hashing: Running Example

Example (Insert record with key $k$ such that $h_0(k) = 50 = 110010_2$)

level = 1

$h_1$

000

001

010

011

100

101

110

111

32

9

66

43

44

5

14

31

25

18

10

34

35

11

36

37

29

30

22

7

50

next

Rehashing a bucket requires rehashing its overflow chain, too.
Linear Hashing: Search Procedure

- Procedures operate over hash table bucket (page) address array `bucket[0, \ldots, 2^{\text{level}} \cdot N - 1]`.
- Variables `level`, `next` are hash-table globals, `N` is constant.

**Linear hashing: search**

```plaintext
Function: hsearch(k)

1. \( b \leftarrow h_{\text{level}}(k) \);
2. \text{if } b < \text{next then}
3.     /* \( b \) has already been split, record for key \( k \) */
4.     /* may be in bucket \( b \) or bucket \( 2^{\text{level}} \cdot N + b \) */
5.     /* ⇒ rehash */
6.     \( b \leftarrow h_{\text{level}+1}(k) \);
7. /* return address of bucket at position \( b \) */
8. return \( \text{bucket}[b] \);
```

Linear Hashing: Insert Procedure

```
Linear hashing: insert

Function: hinsert(k*)

b ← h_{level}(k);
if b < next then
    /* rehash */
    b ← h_{level+1}(k);
Place k* in bucket[b];
if overflow(bucket[b]) then
    Allocate new page b' ;
    /* Grow hash table by one page */
    bucket[2^{level} \cdot N + next] ← addr(b') ;
```

• Predicate overflow(·) is a tunable parameter: whenever overflow(bucket[b]) returns true, trigger a split.
Linear Hashing: Insert Procedure (continued)

Linear hashing: insert (cont’d)

::

```c
if overflow(···) then
    ::
        foreach entry k' in bucket[next] do
            /* redistribute */
            Place k' in bucket[h_{level+1}(k')];

        next ← next + 1;
        /* did we split every bucket in the hash?    */
        if next > 2^{level} \cdot N − 1 then
            /* hash table size doubled, split from top */
            level ← level + 1;
            next ← 0;

        return;
```

Linear Hashing: Delete Procedure (Sketch)

- Linear hashing deletion essentially behaves as the “inverse” of $h_{\text{insert}}(\cdot)$:

\begin{verbatim}
Function: \text{hdelete}(k)

\begin{algorithmic}
  \State $b \leftarrow h_{\text{level}}(k)$;
  \State ... Remove $k*$ from $\text{bucket}[b]$;
  \If {$\text{empty}(\text{bucket}[b])$}
    \State Move entries from page $\text{bucket}[2^{\text{level}} \cdot N + \text{next} - 1]$ to page $\text{bucket}[\text{next} - 1]$;
    \State next $\leftarrow$ next $- 1$;
    \If {next $<$ 0}
      \Statex /* round-robin scheme for deletion */
      \State level $\leftarrow$ level $- 1$;
      \State next $\leftarrow 2^{\text{level}} \cdot N - 1$;
    \EndIf
  \EndIf

  \EndIf

\end{algorithmic}
\end{verbatim}

- Possible: replace \text{empty}(\cdot) by suitable \text{underflow}(\cdot) predicate.