We now turn to a different family of index structures: hash indexes.

Hash indexes are unbeatable when it comes to equality selections, e.g.

\[
\begin{align*}
\text{SELECT} & \quad * \\
\text{FROM} & \quad R \\
\text{WHERE} & \quad [A = k].
\end{align*}
\]

If we carefully maintain the hash index while the underlying data file (for relation \( R \)) grows or shrinks, we can answer such an equality query using a single I/O operation.

Other query types, like joins, internally initiate a whole flood of such equality tests.

Hash indexes provide no support for range searches, however (hash indexes are also known as scatter storage).

In a typical DBMS, you will find support for \( B^+ \) trees as well as hash-based indexing structures.
In a B+ tree world, to locate a record with key $k$ means to compare $k$ with other keys organized in a (tree-shaped) search data structure.

Hash indexes use the bits of $k$ itself (independent of all other stored records and their keys) to find the location of the associated record.

We will now look into static hashing to illustrate the basic ideas behind hashing.

- Static hashing does not handle updates well (much like ISAM).
- Later, we will introduce extendible and linear hashing which refine the hashing principle and adapt well to record insertions and deletions.

4.1 Static Hashing

To build a static hash index for an attribute $A$ we need to

1. allocate an area of $N$ (successive) disk pages, the so-called primary buckets (or hash table),
2. in each bucket, install a pointer to a chain of overflow pages (initially, set this pointer to nil),
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 $A$ has the SQL type INTEGER).
The resulting setup looks like this:

- A primary bucket and its associated chain of overflow pages is referred to as a bucket (above).
- Each bucket contains data entries $k^*$
  (implemented using any of the variants a . . . c, see Chapter 2).

To perform $hsearch(k)$ (or $hinset(k)/hdelete(k)$) for a record with key $A = k$,

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. then search (insert/delete) the record on this page or, if necessary, access the overflow chain of bucket $h(k)$.
If we are lucky or (somehow) avoid chains of overflow pages altogether,

- \( hsearch(k) \) needs one I/O operation,
- \( hinsert(k) \) and \( hdelete(k) \) need two I/O operations.

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).
- However, we **do** want \( h \) to **scatter** the domain of the key attribute **evenly** across \([0 \ldots N - 1]\)
  (to avoid the development of extremely long overflow chains for few buckets).
- Such “good” hash functions are hard to discover, unfortunately
  (see next slide).
The birthday paradox
If you consider the people in a group as the domain and use their birthday as hash function $h$ (i.e., $h : \text{Person} \rightarrow [0 \ldots 364]$), chances are already $> 50\%$ that two people share the same birthday (collision) if the group has $\geq 23$ people.

Check yourself:

1. Compute the probability that $n$ people all have different birthdays:

   $\text{different\_birthday}(n)$:
   
   ```
   if $n = 1$ then
     return 1
   else
     return $\text{different\_birthday}(n - 1) \times \frac{365 - (n - 1)}{365}$
   ```
   
   probability that $n - 1$ persons have different birthdays

   probability that $n^{th}$ person has birthday different from first $n - 1$ persons

2. ... or try to find birthday mates at the next larger party.
If key values would be purely random we could arbitrarily extract a few bits and use these for the hash function. Real key distributions found in DBMS are far from random, though.

Fairly good hash functions may be found using the following two simple approaches:

1. **By division.** Simply define
   \[ h(k) = k \mod N \ . \]
   This guarantees the range of \( h(k) \) to be \([0 \ldots N - 1]\). **N.B.** If you choose \( N = 2^d \) for some \( d \) you effectively consider the least \( d \) bits of \( k \) only. **Prime numbers** were found to work best for \( N \).

2. **By multiplication.** Extract the fractional part of \( Z \cdot k \) (for a specific \( Z \))\(^{10} \) and multiply by hash table size \( N \) (\( N \) is arbitrary here):
   \[ h(k) = \lfloor N \cdot (Z \cdot k - \lfloor Z \cdot k \rfloor) \rfloor . \]
   However, for \( Z = Z'/2^w \) and \( N = 2^d \) (\( w \): number of bits in a CPU word) we simply have
   \[ h(k) = msb_d(Z' \cdot k) \]
   where \( msb_d(x) \) denotes the \( d \) most significant (leading) bits of \( x \) (e.g., \( msb_3(42) = 5 \)).

**Non-numeric key domains?**

How would you hash a non-numeric key domain (e.g., hashing over a CHAR(·) attribute)?

\(^{10} Z = (\sqrt{5} - 1)/2 \approx 0.6180339887 \) is a good choice. See Don E. Knuth, “Sorting and Searching”.

Clearly, if the underlying data file grows, the development of overflow chains spoils the otherwise predictable hash I/O behaviour (1–2 I/Os).

Similarly, if the file shrinks significantly, the static hash table may be a waste of space (data entry slots in the primary buckets remain unallocated).

4.2 Extendible Hashing

Extendible hashing is prepared to adapt to growing (or shrinking) data files.

To keep track of the actual primary buckets which are part of our current hash table, we hash via a in-memory bucket directory (ignore the $2$ fields for now)\textsuperscript{11}:

\textsuperscript{11}N.B.: In this figure we have depicted the data entries as $h(k)\ast$ (not as $k\ast$).
To 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:

To find a record with key $k$ such that $h(k) = 5 = 101_2$, follow the second directory pointer to bucket B, then use entry 5* to access the wanted record.

To insert a record with key $k$:

1. apply $h$, i.e., compute $h(k)$,
2. use the last 2 bits of $h(k)$ to lookup the bucket pointer in the directory,
3. if the bucket still has capacity, store $h(k)*$ in it, otherwise . . . ?
Example (no bucket overflow):

- To insert a record with key $k$ such that $h(k) = 13 = 1101_2$, follow the second directory pointer to bucket B (which still has empty slots) and place $13^*$ there:

Example (continued, bucket overflow):

- Inserting a record with key $k$ such that $h(k) = 20 = 10100_2$ lets bucket A overflow.
- We thus initiate a bucket split for bucket A.
1. **Split** bucket A (creating a new bucket A2) and use bit number (2 + 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 A
  - 32 16
- bucket A2
  - 4 12 20

**N.B.:** we now need 3 bits to discriminate between the old bucket A and the new split bucket A2.

2. In this case we **double the directory** by simply copying its original pages (we now use 2 + 1 = 3 bits to lookup a bucket pointer).

3. Let bucket pointer for 100_2 point to A2 (the directory pointer for 000_2 still points to bucket A):

```plaintext
h
```

```
directory

bucket A

bucket B

bucket C

bucket D

bucket A2

h
```

```
000
001
010
011
100
101
110
111

3

32*16*

2

1* 5* 21* 13*

2

10*

15* 7* 19*

3

4* 12* 20*
```
If we split a bucket with local depth $d < n$ (global depth) directory doubling is not necessary:

- Consider the insertion of record with key $k$ and hash value $h(k) = 9 = 1001_2$.
- The associated bucket $B$ is split (creating a new bucket $B_2$) and entries are redistributed. The new local depth of bucket $B$ is $3$ (and thus does not exceed the global depth of $3$).
- Modifying the directory’s bucket pointer for $101_2$ suffices:
Algorithm: $hsearch \,(k)$
Input: search for hashed record with key value $k$
Output: pointer to hash bucket containing potential hit(s)

$n \leftarrow \left\lceil \frac{d}{2} \right\rceil$; 
// global depth of hash directory
$n \leftarrow \left\lceil \frac{d}{2} \right\rceil$; 
// global depth of hash directory
$b \leftarrow h(k) \& (2^n - 1)$;
return $bucket[b]$;

Algorithm: $hinsert \,(k\star)$
Input: entry $k\star$ to be inserted
Output: new global depth of extendible hash directory

$n \leftarrow \left\lceil \frac{d}{2} \right\rceil$; 
// global depth of hash directory
$b \leftarrow hsearch(k)$;
if $b$ has capacity then
    place $h(k)\star$ in bucket $b$;
    return $n$;
else
    // bucket $b$ overflows, we need to split
    $d \leftarrow \left\lceil \frac{d}{2} \right\rceil$; 
    // local depth of bucket $b$
    create a new empty bucket $b_2$;
    // redistribute entries of bucket $b$ including $h(k)\star$
    for each $h(k')\star$ in bucket $b$ do
        if $h(k') \& 2^d \neq 0$ then
            move $h(k')\star$ to bucket $b_2$;
    $d \leftarrow d + 1$; 
    // new local depth of buckets $b$ and $b_2$
    if $n < d + 1$ then
        // we need to double the directory
        allocate $2^n$ directory entries $bucket[2^n \ldots 2^{n+1} - 1]$;
        copy $bucket[0 \ldots 2^n - 1]$ into $bucket[2^n \ldots 2^{n+1} - 1]$;
        $n \leftarrow n + 1$;
        $n \leftarrow n$;
        $bucket[(h(k) \& (2^n - 1)) \mid 2^{n-1}] \leftarrow addr(b_2)$;
    return $n$;

Remarks:

- & and | denote bit-wise and and bit-wise or (just like in C, C++)
- The directory entries are accessed via the array $bucket[0 \ldots 2^n - 1]$ whose entries point to the hash buckets.
Extendible hashing uses overflow chains hanging off a bucket only as a last resort. Under which circumstances will extendible hashing create an overflow chain?

- Deleting an entry $h(k)\ast$ from a bucket with local depth $d$ may leave this bucket completely empty.
  - Extendible hashing merges the empty bucket and its associated bucket $2^{n-d}$ partner buckets.
  - You should work out the details on your own.

4.3 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,
  - . . . but linear hashing may perform bad if the key distribution in the data file is skewed.

- We will now investigate linear hashing in detail and come back to the two 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).

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 \( \text{level} = 0, 1, 2, \ldots \)).

This relationship is depicted below. Here, \( h_{\text{level}} \) has range \([0 \ldots N - 1]\) (i.e., a range of size \( N \)):

\[
\begin{align*}
\text{level} + 1 & \\
\text{level} & \\
\text{level} + 2 & \\
\end{align*}
\]

Given an initial hash function \( h \) and an initial hash table size \( N \), one approach to define such a family of hash functions \( h_0, h_1, h_2, \ldots \) would be

\[
h_{\text{level}}(k) = h(k) \mod (2^{\text{level}} \cdot N)
\]

(for \( \text{level} = 0, 1, 2, \ldots \)).
The basic **linear hashing** scheme then goes like this:

- **Start** with $level = 0$, $next = 0$.

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

- **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 $l$ pages,
  - or . . .

  we **split the bucket** as hash table position $next$ (**not the bucket which triggered the 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 remain in bucket $next$, some go to bucket $2^{level} \cdot N + next$ [Which?]),
  3. **increment** $next$ by 1.
As we proceed, `next` walks down the table. Hashing via $h_{level}$ has to take care of `next`'s position:

$$h_{level}(k) \begin{cases} < next : \text{we hit an already split bucket, rehash: find record in bucket } h_{level+1}(k) \\ \geq next : \text{we hit a yet unsplit bucket, bucket found} \end{cases}$$

![Diagram showing the range of $h_{level}$ and $h_{level+1}$, with buckets and images of already split buckets.]
If \( \text{next} \) is incremented beyond the 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 if \( \text{next} \) is incremented beyond the last current hash table position, \( i.e., \)

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

Answer:

- N.B.: If \( \text{next} > 2^{\text{level}} \cdot N - 1 \), all buckets in the current hash table are hashed via \( h_{\text{level+1}} \) (see previous slide).
- Linear hashing thus proceeds in a round-robin fashion:
  
  If \( \text{next} > 2^{\text{level}} \cdot N - 1 \), then
  
  1. increment \( \text{level} \) by 1,
  2. reset \( \text{next} \) to 0 (start splitting from the hash table top again).

Remark:

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

- Linear hash table: primary bucket capacity of 4, initial hash table size $N = 4$, level = 0, next = 0:

  ![Diagram of linear hash table with initial state]

- Insert record with key $k$ such that $h_0(k) = 43$:

  ![Diagram of linear hash table after insert]

- Insert record with key \( k \) such that \( h_0(k) = 37 \):

- Insert record with key \( k \) such that \( h_0(k) = 29 \):
Insert 3 records with keys $k$ such that $h_0(k) = 22$ (66, 34):

Insert record with key $k$ such that $h_0(k) = 50$:

N.B.

Rehashing a bucket means to **rehash its overflow chain**, too.
Algorithm: \textit{hsearch} \((k)\)
Input: search for hashed record with key value \(k\)
Output: pointer to hash bucket containing potential hit(s)

\[
b \leftarrow h_{\text{level}}(k);
\]
\[
\text{if } b < \text{next} \text{ then}
\]
\[
\quad // \text{bucket } b \text{ has already been split,}
\quad // \text{the record for key } k \text{ may be in bucket } b \text{ or bucket}\ 2^{\text{level} \cdot N + b},
\quad // \text{rehash:}
\quad b \leftarrow h_{\text{level+1}}(k);
\]
\[
\text{return } \text{bucket}[b];
\]

Algorithm: \textit{hinsert} \((k\ast)\)
Input: entry \(k\ast\) to be inserted
Output: none

\[
b \leftarrow h_{\text{level}}(k);
\]
\[
\text{if } b < \text{next} \text{ then}
\]
\[
\quad // \text{bucket } b \text{ has already been split, rehash:}
\quad b \leftarrow h_{\text{level+1}}(k);
\quad \text{place } h(k)\ast \text{ in } \text{bucket}[b];
\quad \text{if } \text{full(bucket[b])} \text{ then}
\]
\[
\quad // \text{the last insertion triggered a split of bucket } \text{next}
\quad \text{allocate a new bucket } b';
\quad \text{bucket}[2^{\text{level} \cdot N + \text{next}]} \leftarrow b';
\quad // \text{rehash the entries of bucket } \text{next}
\quad \text{for each entry with key } k' \text{ in } \text{bucket[next]} \text{ do}
\quad \quad \text{place entry in } \text{bucket}[h_{\text{level+1}}(k')];
\quad \text{next} \leftarrow \text{next} + 1;
\quad // \text{did we split every bucket in the original hash table?}
\quad \text{if } \text{next} > 2^{\text{level} \cdot N - 1} \text{ then}
\quad \quad // \text{hash table size has doubled, start a new round now}
\quad \quad \text{level} \leftarrow \text{level} + 1;
\quad \quad \text{next} \leftarrow 0;
\]
\[
\text{return ;}
\]

Remarks:

- \(\text{bucket}[b]\) denotes the \(b\)th bucket in the hash table.
- Function \(\text{full(\cdot)}\) is a tunable parameter: whenever \(\text{full(bucket[b])}\) evaluates to \text{true} we trigger a split.
For a linear hash table, $hdelete(k)$ can essentially be implemented as the inverse of $hinsert(k)$:

**Algorithm:**

$hdelete (k)$

**Input:**

key $k$ of entry to be deleted

**Output:**

none

```plaintext
if empty(bucket[2^{level}\cdot N + next]) then
  // the last bucket in the hash table is empty, remove it
  remove bucket[2^{level}\cdot N + next] from hash table;
  next ← next - 1;
  if next < 0 then
    // round-robin scheme for deletion
    level ← level - 1;
    next ← 2^{level}\cdot N - 1;
```

...
Chapter Contents

4.1 Static Hashing ......................................................... 92
4.2 Extendible Hashing .................................................. 97
4.3 Linear Hashing ....................................................... 103