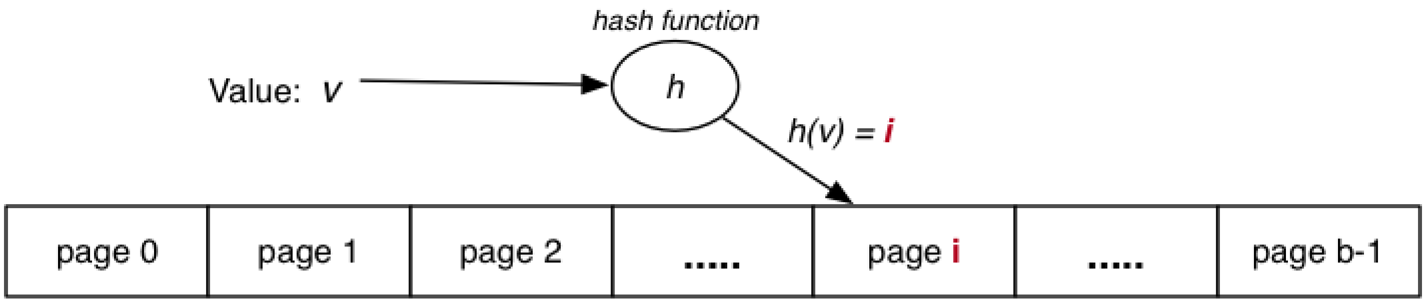
>>

Hashed Files

∧ >>

❖ Hashing

Basic idea: use key value to compute page address of tuple.



e.g. tuple with key = *v* is stored in page *i*

Requires: hash function *h(v)* that maps *KeyDomain* → *[0..b-1]*.

- hashing converts key value (any type) into integer value
- integer value is then mapped to page index
- note: can view integer value as a bit-string

<< ∧ >>

❖ Hashing (cont)

PostgreSQL hash function (simplified):

```
Datum hash_any(unsigned char *k, int keylen)
{
   uint32 a, b, c, len, *ka = (uint32 *)k;
   /* Set up the internal state */
   len = keylen;
   a = b = c = 0x9e3779b9+len+3923095;
   /* handle most of the key */
   while (len >= 12) {
      a += ka[0]; b += ka[1]; c += ka[2];
      mix(a, b, c);
      ka += 3;  len -= 12;
   }
   ... collect data from remaining bytes into a,b,c ...
   mix(a, b, c);
   return UInt32GetDatum(c);
}
```

See **`backend/access/hash/hashfunc.c`** for details (incl **`mix()`**)

COMP9315 21T1 ♢ Hashed Files ♢ [2/15]

❖ Hashing (cont)

`hash_any()` gives hash value as 32-bit quantity (**`uint32`**).

Two ways to map raw hash value into a page address:

- if $b = 2^k$, bitwise AND with *k* low-order bits set to one

```
uint32 hashToPageNum(uint32 hval) {
    uint32 mask = 0xFFFFFFFF;
    return (hval & (mask >> (32-k)));
}
```

- otherwise, use *mod* to produce value in range *0..b-1*

```
uint32 hashToPageNum(uint32 hval) {
    return (hval % b);
}
```

❖ Hashing Performance

Aims:

- distribute tuples evenly amongst buckets
- have most buckets nearly full (attempt to minimise wasted space)

Note: if data distribution not uniform, address distribution can't be uniform.

Best case: every bucket contains same number of tuples.

Worst case: every tuple hashes to same bucket.

Average case: some buckets have more tuples than others.

Use overflow pages to handle "overfull" buckets (cf. sorted files)

All tuples in each bucket must have same hash value.

<< ^ >>

❖ Hashing Performance (cont)

Two important measures for hash files:

- load factor: $L = r / bc$
- average overflow chain length: $Ov = b_{ov} / b$

Three cases for distribution of tuples in a hashed file:

Case	L	Ov
Best	$\cong 1$	0
Worst	$>> 1$	**
Average	< 1	$0<Ov<1$

(** performance is same as Heap File)

To achieve average case, aim for $0.75 \leq L \leq 0.9$.

COMP9315 21T1 ♢ Hashed Files ♢ [5/15]

<< ^ >>

❖ Selection with Hashing

Select via hashing on unique key *k* (*one*)

```
// select * from R where k = val
getPageViaHash(R, val, P)
for each tuple t in page P {
    if (t.k == val) return t
}
for each overflow page Q of P {
    for each tuple t in page Q {
        if (t.k == val) return t
}   }
```

$Cost_{one}$: Best = 1, Avg = *1+Ov/2* Worst = *1+max(OvLen)*

<< ∧ >>

❖ Selection with Hashing (cont)

Working out which page, given a key ...

```
getPageViaHash(Reln R, Value key, Page p)
{
   uint32 h = hash_any(key, len(key));
   PageID pid = h % nPages(R);
   get_page(R, pid, buf);
}
```

<< ^ >>

❖ Selection with Hashing (cont)

Select via hashing on non-unique hash key *nk* (*pmr*)

```
// select * from R where nk = val
getPageViaHash(R, val, P)
for each tuple t in page P {
    if (t.nk == val) add t to results
}
for each overflow page Q of P {
    for each tuple t in page Q {
        if (t.nk == val) add t to results
}   }
return results
```

$Cost_{pmr} = 1 + Ov$

If *Ov* is small (e.g. 0 or 1), very good retrieval cost

<< ∧ >>

❖ Selection with Hashing (cont)

Hashing does not help with *range* queries** ...

$Cost_{range} = b + b_{ov}$

Selection on attribute *j* which is not hash key ...

$Cost_{one}$, $Cost_{range}$, $Cost_{pmr}$ = $b + b_{ov}$

** unless the hash function is order-preserving (and most aren't)

<< ∧ >>

❖ Insertion with Hashing

Insertion uses similar process to *one* queries.

```
// insert tuple t with key=val into rel R
getPageViaHash(R, val, P)
if room in page P {
    insert t into P; return
}
for each overflow page Q of P {
    if room in page Q {
        insert t into Q; return
}   }
add new overflow page Q
link Q to previous page
insert t into Q
```

$Cost_{insert}$: Best: $1_r + 1_w$ Worst: $1{+}max(OvLen))_r + 2_w$

❖ Deletion with Hashing

Similar performance to select on non-unique key:

```
// delete from R where k = val
// f = data file ... ovf = ovflow file
getPageViaHash(R, val, P)
ndel = delTuples(P,k,val)
if (ndel > 0) put_page(dataFile(R),P.pid,P)
for each overflow page Q of P {
    ndel = delTuples(Q,k,val)
    if (ndel > 0) put_page(ovFile(R),Q.pid,Q)
}
```

Extra cost over select is cost of writing back modified pages.

Method works for both unique and non-unique hash keys.

<< ∧ >>

❖ Problem with Hashing...

So far, discussion of hashing has assumed a fixed file size (*b*).

What size file to use?

- the size we need right now (performance degrades as file overflows)
- the maximum size we might ever need (signifcant waste of space)

Problem: change file size ⇒ change hash function ⇒ rebuild file

Methods for hashing with files whose size changes:

- extendible hashing, dynamic hashing (need a directory, no overflows)
- linear hashing (expands file "sytematically", no directory, has overflows)

<< ∧ >>

❖ Flexible Hashing

All flexible hashing methods ...

- treat hash as 32-bit bit-string
- adjust hashing by using more/less bits

Start with hash function to convert value to bit-string:

```
uint32 hash(unsigned char *val)
```

Require a function to extract *d* bits from bit-string:

```
unit32 bits(int d, uint32 val)
```

Use result of **`bits()`** as page address.

<< ^ >>

❖ Flexible Hashing (cont)

Important concept for flexible hashing: splitting

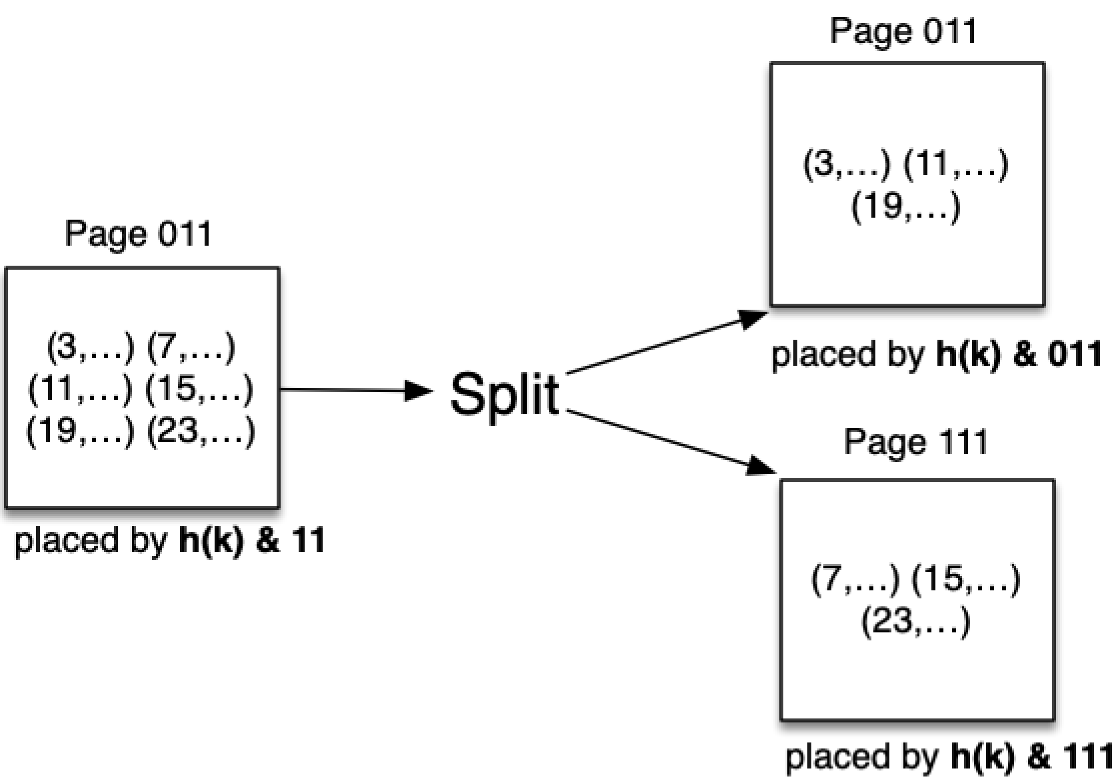
- consider one page (all tuples have same hash value)
- recompute page numbers by considering one extra bit
- if current page is **101**, new pages have hashes **0101** and **1101**
- some tuples stay in page **0101** (was **101**)
- some tuples move to page **1101** (new page)
- also, rehash any tuples in overflow pages of page **101**

Result: expandable data file, never requiring a complete file rebuild

<< ^

❖ Flexible Hashing (cont)

Example of splitting:



Tuples only show key value; assume *h(val) = val*

Produced: 7 Mar 2021