Week 06 Lecture

Assignment 1 Review

Add a new base data type to PostgreSQL

Email addresses: *local* **@** *domain*

Variable lengths, up to 128 chars, case-insensitive

Operators: **=** same, **>** greater (dom,loc), **~** same domain, etc.

Support btree index and hashed files

```
Local       ::= NamePart NameParts
Domain      ::= NamePart '.' NamePart NameParts
NamePart    ::= Letter | Letter NameChars (Letter|Digit)
NameParts   ::= Empty | '.' NamePart NameParts
NameChars   ::= Empty | (Letter|Digit|'-') NameChars
```

Need: storage structure, in/out/operator functions, operator classes

---

... Assignment 1 Review

Decisions for stored representation:

- split into local+domain or keep as one string
- canonicalize before storing or when using operators
- fixed-length structure or variable length structure

Typical solution:

```
struct Email { char local[128]; char domain[128]; }
```

Problems: wastes space, buffers too short (129 for `'\0'`)

Better solution:

```
struct Email { int32 len; int32 dom0; char addr[1]; }
```

Assumes: copy whole string, convert to lower-case, replace `'@'` by `'\0'`

... Assignment 1 Review

Storing in canonical form (e.g. all lower-case), and pre-split

- simplifies query-time operations like `email_cmp()`

Having a generic `email_cmp()` function

- simplifies rest of code, especially operator functions

Accesing data in var-length pre-split struct:

```
struct Email *ep;
ep = (struct Email *)PG_GETARG_POINTER(0);
char *local = &(ep->addr[0]);
char *domain = &(ep->addr[ep->dom0]);
```

... Assignment 1 Review

Common errors ...

- `struct Email { char *local; char *domain; }`
    - tuple data must be stored within the struct
- buffers of size 128 (should be 129, unless storing length)
- `sscanf(str, "[^@]@[^@]", locBuf, domBuf)`
- or even a regex like `"[A-Za-z0-9.-]+@[A-Za-z0-9.-]+"`
- `internallength = ?` in `create type EmailAddress`
    - needs to match `sizeof struct Email` (unless varlen)
- memory leaks (e.g. not freeing regex buffers)
- thinking that 20 tuples is going to use indexing

Debugging server errors can be tedious (`fprintf` to log file)

Recap on Implementing Selection

Selection = `select * from` *R* `where` *C*

- yields a subset of *R* tuples satisfying condition *C*
- a very important (frequent) operation in relational databases

Types of selection determined by type of condition

- *one*: `select * from` *R* `where id =` *k*
- *pmr*: `select * from` *R* `where age=65` (1-d)

  `select * from` *R* `where age=65 and gender='m'` (n-d)
- *rng*: `select * from` *R* `where age≥18 and age≤21` (1-d)

  `select * from` *R* `where age between 18 and 21` (n-d)
  `and height between 160 and 190`

... Recap on Implementing Selection

Strategies for implementing selection efficiently

- arrangement of tuples in file (e.g. sorting, hashing)
- auxiliary data structures (e.g. indexes, signatures)

Interested in cost for `select`, `delete`, `update`, and `insert`

- for `select`, simply count number of pages read n_r
- for others, use n_r and n_w to distinguish reads/writes

Typical file structure has

- *b* main data pages, b_{Ov} overflow pages, *c* tuples per page
- auxiliary files with e.g. oversized values, index entries

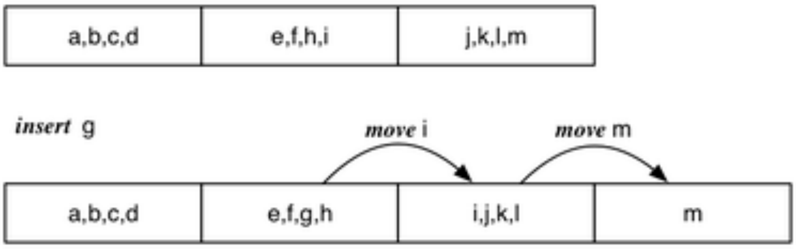
Sorted Files

Sorted Files

Records stored in file in order of some field *k* (the sort key).

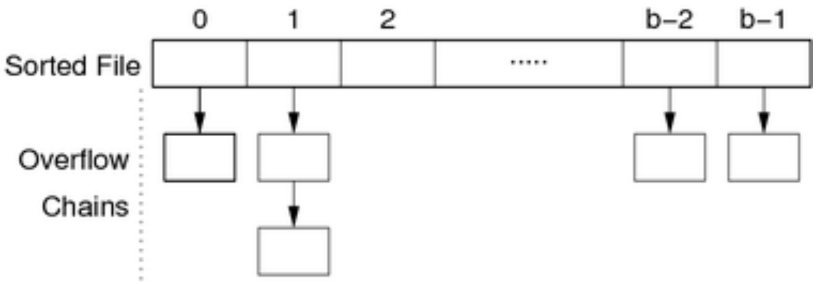
Makes searching more efficient; makes insertion less efficient

E.g. assume *c = 4*



... Sorted Files

In order to mitigate insertion costs, use overflow blocks.



Total number of overflow blocks = b_{ov}.

Average overflow chain length = $Ov = b_{ov} / b$.

Bucket = data page + its overflow page(s)

Selection in Sorted Files

For *one* queries on sort key, use binary search.

```
// select * from R where k = val  (sorted on R.k)
lo = 0; hi = b-1
while (lo <= hi) {
    mid = (lo+hi) div 2;
    (tup,loVal,hiVal) = searchBucket(f,mid,k,val);
    if (tup != null) return tup;
    else if (val < loVal) hi = mid - 1;
    else if (val > hiVal) lo = mid + 1;
    else return NOT_FOUND;
}
return NOT_FOUND;
```

where `f` is file for relation, `mid,lo,hi` are page indexes,
`k` is a field/attr, `val,loVal,hiVal` are values for `k`

... Selection in Sorted Files

Search a page and its overflow chain for a key value

```
searchBucket(f,p,k,val)
{
    buf = getPage(f,p);
    (tup,min,max) = searchPage(buf,k,val,+INF,-INF)
    if (tup != NULL) return(tup,min,max);
```

```
    ovf = openOvFile(f);
    ovp = ovflow(buf);
    while (tup == NULL && ovp != NO_PAGE) {
        buf = getPage(ovf,ovp);
        (tup,min,max) = searchPage(buf,k,val,min,max)
        ovp = ovflow(buf);
    }
    return (tup,min,max);
}
```

Assumes each page contains index of next page in Ov chain

... Selection in Sorted Files

Search within a page for key; also find min/max key values

```
searchPage(buf,k,val,min,max)
{
    res = NULL;
    for (i = 0; i < nTuples(buf); i++) {
        tup = getTuple(buf,i);
        if (tup.k == val) res = tup;
        if (tup.k < min) min = tup.k;
        if (tup.k > max) max = tup.k;
    }
    return (res,min,max);
}
```

... Selection in Sorted Files

The above method treats each bucket like a single large page.

Cases:

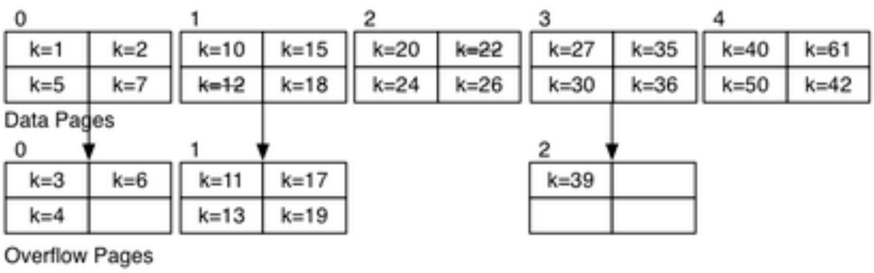
- best: find tuple in first data page we read
- worst: full binary search, and not found
    - examine $log_2 b$ data pages
    - plus examine all of their overflow pages
- average: examine some data pages + their overflow pages

$Cost_{one}$:   Best = *1*   Worst = $log_2\, b + b_{ov}$

Average case cost analysis relies on assumptions (e.g. data distribution)

Exercise 1: Searching in Sorted File

Consider this sorted file with overflows ($b=5$, $c=4$):



Compute the cost for answering each of the following:

- `select * from R where k = 24`

- `select * from R where k = 3`
- `select * from R where k = 14`
- `select max(k) from R`

Exercise 2: Optimising Sorted-file Search

15/54

The `searchBucket(f,p,k,val)` function requires:

- read the p^{th} page from data file
- scan it to find a match and min/max k values in page
- while no match, repeat the above for each overflow page
- if we find a match in any page, return it
- otherwise, remember min/max over all pages in bucket

Suggest an optimisation that would improve `searchBucket()` performance for most buckets.

... Selection in Sorted Files

16/54

For *pmr* query, on non-unique attribute *k*

- assume file is sorted on *k*
- tuples containing *k* may appear in several pages

Begin by locating a page *p* containing `k=`*val* (as for *one* query).

Scan backwards and forwards from *p* to find matches.

Thus, $Cost_{pmr} = Cost_{one} + (b_q\text{-}1).(1+Ov)$

... Selection in Sorted Files

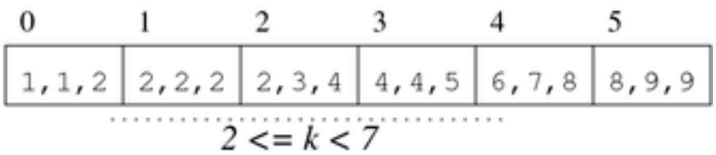
17/54

For *range* queries on unique sort key (e.g. primary key):

- use binary search to find lower bound
- read sequentially until reach upper bound

$Cost_{range} = Cost_{one} + (b_q\text{-}1).(1+Ov)$

If secondary key, similar method to *pmr*.



... Selection in Sorted Files

18/54

So far, have assumed query condition involves sort key *k*.

If condition contains attribute *j*, not the sort key

- file is unlikely to be sorted by *j* as well
- sortedness gives no searching benefits

$Cost_{one}$, $Cost_{range}$, $Cost_{pmr}$ as for heap files

Updates to Sorted Files

Insertion approach:

- find appropriate page for tuple (via binary search)
- if page not full, insert into page
- otherwise, insert into next overflow block with space

Thus, $Cost_{insert} = Cost_{one} + \delta_w$ (where δ_w = 1 or 2)

Deletion strategy:

- find matching tuple(s)
- mark them as deleted

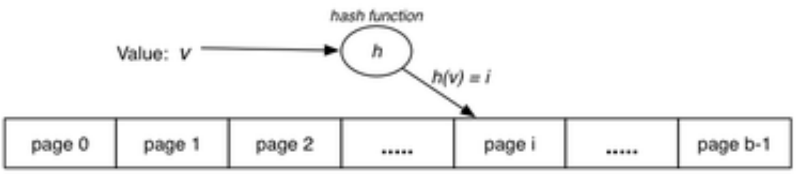
Cost depends on selectivity of selection condition

Thus, $Cost_{delete} = Cost_{select} + b_{qw}$

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

PostgreSQL hash function (simplified):

```
uint32 hash_any(unsigned char *k, register int keylen)
{
   register uint32 a, b, c, len;
   /* Set up the internal state */
   len = keylen;   a = b = 0x9e3779b9;   c = 3923095;
   /* handle most of the key */
   while (len >= 12) {
      a += (k[0] + (k[1]<<8) + (k[2]<<16) + (k[3]<<24));
      b += (k[4] + (k[5]<<8) + (k[6]<<16) + (k[7]<<24));
      c += (k[8] + (k[9]<<8) + (k[10]<<16) + (k[11]<<24));
      mix(a, b, c);   k += 12; len -= 12;
```

```
    }
    /* collect any data from last 11 bytes into a,b,c */
    mix(a, b, c);
    return c;
}
```

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

... Hashing

23/54

`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

24/54

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

25/54

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	≅ *1*	0
Worst	*>> 1*	**
Average	*< 1*	*0<Ov<1*

(** performance is same as Heap File)

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

Selection with Hashing

Best performance occurs for *one* queries on hash key field.

Basic strategy:

- compute page address via hash function *hash(val)*
- fetch that page and look for matching tuple
- possibly fetch additional pages from overflow chain

Best $Cost_{one} = 1$ (find in data page)

Average $Cost_{one} = 1+Ov/2$ (scan half of ovflow chain)

Worst $Cost_{one} = 1+max(OvLen)$ (find in last page of ovflow chain)

... Selection with Hashing

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

```
// select * from R where k = val
f = openFile(relName("R"),READ);
p = hash(val) % nPages(f);
buf = getPage(f, p)
for (i = 0; i < nTuples(buf); i++) {
    tup = getTuple(buf,i);
    if (tup.k == val) return tup;
}
ovp = ovflow(buf);
while (ovp != NO_PAGE) {
    buf = getPage(ovf,ovp);
    for (i = 0; i < nTuples(Buf); i++) {
        tup = getTuple(buf,i);
        if (tup.k == val) return tup;
}   }
```

... Selection with Hashing

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

```
// select * from R where k = val
f = openFile(relName("R"),READ);
p = hash(val) % nPages(f);
buf = getPage(f, p)
for (i = 0; i < nTuples(buf); i++) {
    tup = getTuple(buf,i);
    if (tup.k == val) append tup to results
}
ovp = ovflow(buf);
while (ovp != NO_PAGE) {
    buf = getPage(ovf,ovp);
    for (i = 0; i < nTuples(Buf); i++) {
        tup = getTuple(buf,i);
        if (tup.k == val) append tup to results
}   }
```

$Cost_{pmr} = 1 + Ov$

... Selection with Hashing

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
// f = data file ... ovf = ovflow file
p = hash(val) % nPages(R)
P = getPage(f,p)
if (tup fits in page P)
    { insert t into P; return }
for each overflow page Q of P {
    if (tup fits in page Q)
        { insert t into Q; return }
}
add new overflow page Q
link Q to previous overflow page
insert t into Q
```

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

Exercise 3: Insertion into Static Hashed File

Consider a file with *b=4*, *c=3*, *d=2*, *h(x) = bits(d,hash(x))*

Insert tuples in alpha order with the following keys and hashes:

k	*hash(k)*		*k*	*hash(k)*		*k*	*hash(k)*		*k*	*hash(k)*
a	10001		g	00000		m	11001		s	01110
b	11010		h	00000		n	01000		t	10011
c	01111		i	10010		o	00110		u	00010
d	01111		j	10110		p	11101		v	11111
e	01100		k	00101		q	00010		w	10000
f	00010		l	00101		r	00000		x	00111

The hash values are the 5 lower-order bits from the full 32-bit hash.

Deletion with Hashing

Similar performance to select:

```
// delete from R where k = val
```

```
// f = data file ... ovf = ovflow file
p = hash(val) % nPages(R)
buf = getPage(f,p)
ndel = delTuples(buf,k,val)
if (ndel > 0) putPage(f,buf,p)
p = ovFlow(buf)
while (p != NO_PAGE) {
    buf = getPage(ovf,p)
    ndel = delTuples(buf,k,val)
    if (ndel > 0) putPage(ovf,buf,p)
    p = ovFlow(buf)
}
```

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

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

Problem with Hashing...

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

Change file size ⇒ change hash function ⇒ rebuild file

Methods for hashing with dynamic files:

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

... Problem with 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.

Exercise 4: Bit Manipulation

1. Write a function to display `uint32` values as `01010110...`

   ```
   char *showBits(uint32 val, char *buf);
   ```

   Analogous to `gets()` (assumes supplied buffer large enough)

2. Write a function to extract the *d* bits of a `uint32`

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

If $d > 0$, gives low-order bits; if $d < 0$, gives high-order bits

---

... Problem with Hashing...

Important concept for flexible hashing: *splitting*

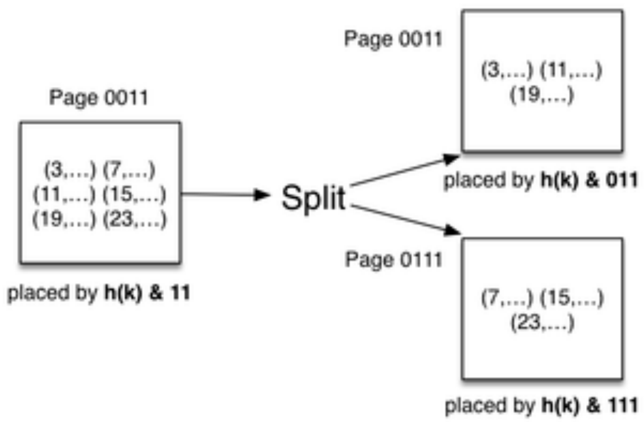
- consider one page (all tuples have same hash value)
- recompute page numbers by considering one extra bit
- if current page is `101`, new pages have hashes `0101` and `1101`
- some tuples stay in page `0101` (was `101`)
- some tuples move to page `1101` (new page)
- also, rehash any tuples in overflow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

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

---

... Problem with Hashing...

Example of splitting:



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

Linear Hashing

File organisation:

- file of primary data blocks
- file of overflow data blocks
- a register called the *split pointer*

Uses systematic method of growing data file ...

- hash function "adapts" to changing address range
- systematic splitting controls length of overflow chains

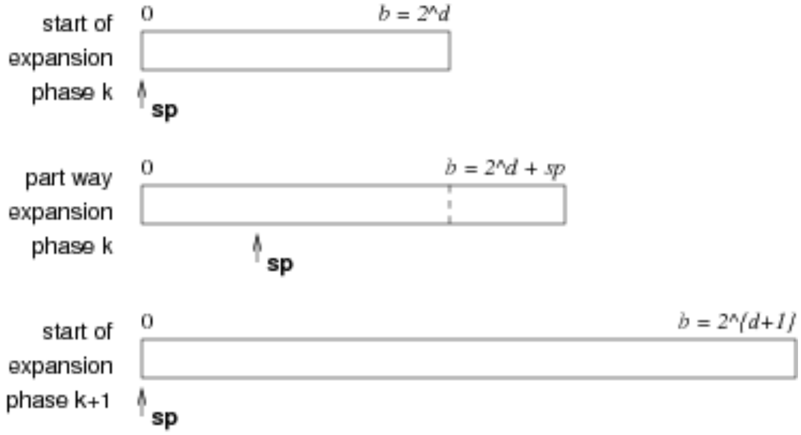
Advantage: does *not* require auxiliary storage for a directory

Disadvantage: requires overflow pages (splits don't occur on full pages)

... Linear Hashing

File grows linearly (one block at a time, at regular intervals).

Has "phases" of expansion; during each phase, *b* doubles.



Selection with Lin.Hashing

If $b=2^d$, the file behaves exactly like standard hashing.

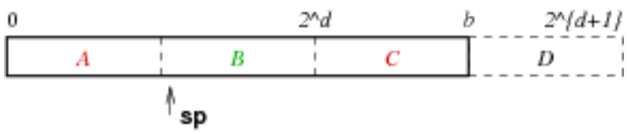
Use *d* bits of hash to compute block address.

```
// select * from R where k = val
h = hash(val);
P = bits(d,h);   // lower-order bits
for each tuple t in page P
         and its overflow pages {
    if (t.k == val) return t;
}
```

Average $Cost_{one}$ = $1+Ov$

... Selection with Lin.Hashing

If b != 2^d, treat different parts of the file differently.



Parts *A* and *C* are treated as if part of a file of size 2^{d+1}.

Part *B* is treated as if part of a file of size 2^d.

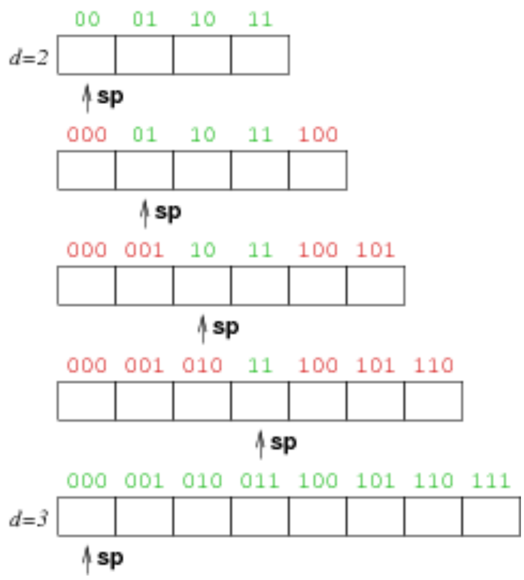
Part *D* does not yet exist (*B* expands into it).

... Selection with Lin.Hashing

Modified search algorithm:

```
// select * from R where k = val
h = hash(val);
P = bits(d,h);
if (P < sp) { P = bits(d+1,h); }
for each tuple t in page P
         and its overflow blocks {
    if (t.k == val) return R;
}
```

File Expansion with Lin.Hashing



Exercise 5: Insertion into Linear Hashed File

Consider a file with *b=4*, *c=3*, *d=2*, *sp=0*, *hash(x)* as above

Insert tuples in alpha order with the following keys and hashes:

k	*hash(k)*		*k*	*hash(k)*		*k*	*hash(k)*		*k*	*hash(k)*
a	10001		g	00000		m	11001		s	01110
b	11010		h	00000		n	01000		t	10011
c	01111		i	10010		o	00110		u	00010
d	01111		j	10110		p	11101		v	11111
e	01100		k	00101		q	00010		w	10000
f	00010		l	00101		r	00000		x	00111

The hash values are the 5 lower-order bits from the full 32-bit hash.

Insertion with Lin.Hashing

Abstract view:

```
P = bits(d,hash(val));
if (P < sp) P = bits(d+1,hash(val));
// bucket P = page P + its overflow pages
for each page Q in bucket P {
    if (space in Q) {
        insert tuple into Q
        break
    }
}
if (no insertion) {
    add new ovflow page to bucket P
    insert tuple into new page
}
if (need to split) {
    partition tuples from bucket sp
```

```
          into buckets sp and sp+2^d
    sp++;
    if (sp == 2^d) { d++; sp = 0; }
}
```

Splitting

How to decide that we "need to split"?

Two approaches to triggering a split:

- split every time a tuple is inserted into full block
- split when load factor reaches threshold (every *k* inserts)

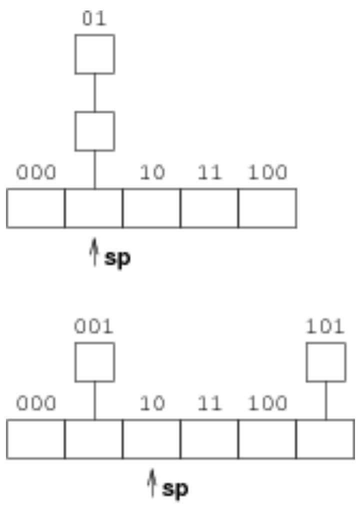
Note: always split block *sp*, even if not full/"current"

Systematic splitting like this ...

- eventually reduces length of every overflow chain
- helps to maintain short average overflow chain length

... Splitting

Splitting process for block *sp*=`01`:



... Splitting

Detailed splitting algorithm:

```
// partitions tuples between two buckets
newp = sp + 2^d; oldp = sp;
buf = getPage(f,sp);
clear(oldBuf); clear(newBuf);
for (i = 0; i < nTuples(buf); i++) {
    tup = getTuple(buf,i);
    p = bits(d+1,hash(tup.k));
    if (p == newp)
        addTuple(newBuf,tup);
    else
        addTuple(oldBuf,tup);
}
p = ovflow(buf);  oldOv = newOv = 0;
while (p != NO_PAGE) {
    ovbuf = getPage(ovf,p);
    for (i = 0; i < nTuples(ovbuf); i++) {
        tup = getTuple(buf,i);
```

```
        p = bits(d+1,hash(tup.k));
        if (p == newp) {
            if (isFull(newBuf)) {
                nextp = nextFree(ovf);
                ovflow(newBuf) = nextp;
                outf = newOv ? f : ovf;
                writePage(outf, newp, newBuf);
                newOv++; newp = nextp; clear(newBuf);
            }
            addTuple(newBuf, tup);
        }
        else {
            if (isFull(oldBuf)) {
                nextp = nextFree(ovf);
                ovflow(oldBuf) = nextp;
                outf = oldOv ? f : ovf;
                writePage(outf, oldp, oldBuf);
                oldOv++; oldp = nextp; clear(oldBuf);
            }
            addTuple(oldBuf, tup);
        }
    }
    addToFreeList(ovf,p);
    p = ovflow(buf);
}
sp++;
if (sp == 2^d) { d++; sp = 0; }
```

Insertion Cost

If no split required, cost same as for standard hashing:

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

If split occurs, incur $Cost_{insert}$ plus cost of splitting:

- read block *sp* (plus all of its overflow blocks)
- write block *sp* (and its new overflow blocks)
- write block $sp+2^d$ (and its new overflow blocks)

On average, $Cost_{split} = (1+Ov)_r + (2+Ov)_w$

Deletion with Lin.Hashing

Deletion is similar to ordinary static hash file.

But might wish to contract file when enough tuples removed.

Rationale: *r* shrinks, *b* stays large ⇒ wasted space.

Method: remove last bucket in data file (contracts linearly).

Involves a coalesce procedure which is an inverse split.

Hash Files in PostgreSQL

PostgreSQL uses linear hashing on tables which have been:

`create index Ix on R using hash (k);`

Hash file implementation: **`backend/access/hash`**

- **`hashfunc.c`** ... a family of hash functions

- **hashinsert.c** ... insert, with overflows
- **hashpage.c** ... utilities + splitting
- **hashsearch.c** ... iterator for hash files

Based on "A New Hashing Package for Unix", Margo Seltzer, Winter Usenix 1991

... Hash Files in PostgreSQL

52/54

PostgreSQL uses slightly different file organisation ...

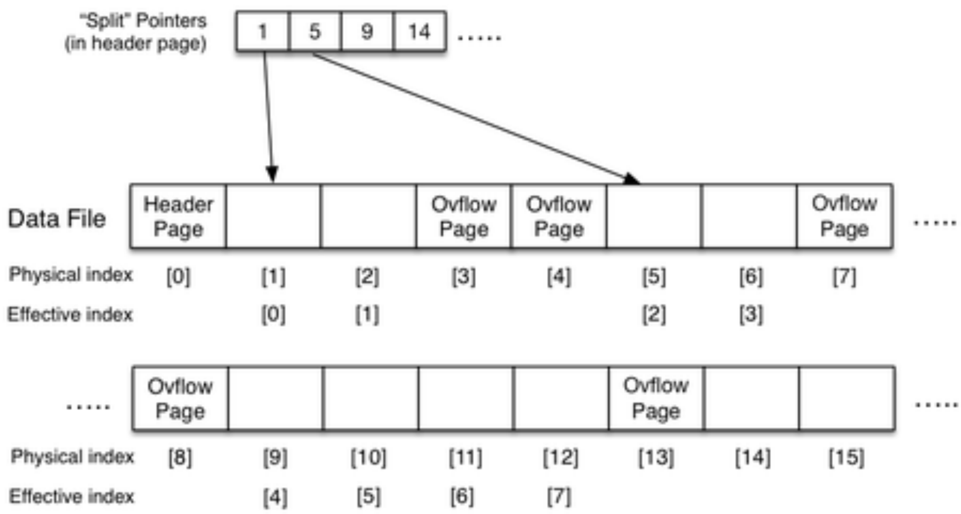
- has a single file containing main and overflow pages
- has groups of main pages of size 2^n
- in between groups, arbitrary number of overflow pages
- maintains collection of "split pointers" in header page
- each split pointer indicates start of main page group

If overflow pages become empty, add to free list and re-use.

... Hash Files in PostgreSQL

53/54

PostgreSQL hash file structure:



... Hash Files in PostgreSQL

54/54

Converting bucket # to page address:

```
// which page is primary page of bucket
uint bucket_to_page(headerp, B) {
   uint *splits = headerp->hashm_spares;
   uint chunk, base, offset, lg2(uint);
   chunk = (B<2) ? 0 : lg2(B+1)-1;
   base = splits[chunk];
   offset = (B<2) ? B : B-(1<chunk);
   return (base + offset);
}
// returns ceil(log_2(n))
int lg2(uint n) {
        int i, v;
        for (i = 0, v = 1; v < n; v <= 1) i++;
        return i;
}
```

Produced: 12 Apr 2016