Week 05 Lectures

Projection on Primary Key

1/73

No duplicates, so the above approaches are not required.

Method:

Get all pages

```
bR = nPages(Rel)
for i in 0 .. bR-1 {
   P = read page i
   for j in 0 .. nTuples(P) {
      T = getTuple(P,j)
      T' = mkTuple(pk, T)
      if (outBuf is full) write and clear
      append T' to outBuf
   }
}
if (nTuples(outBuf) > 0) write
```

Index-only Projection

2/73

two common of indexing:
1. ID PageID
2. Id_1 ~ Id_n : page_01
   Id_n ~ Id_2n : page_02

Can do projection without accessing data file iff ..

- relation is indexed on $(A_1,A_2,...A_n)$ (indexes described later)
- projected attributes are a prefix of $(A_1,A_2,...A_n)$

Basic idea:

- scan through index file (which is already sorted on attributes)
- duplicates are already adjacent in index, so easy to skip

Cost analysis ...

- index has b_i pages (where $b_i \ll b_R$)
- Cost = b_i reads + b_{Out} writes

Comparison of Projection Methods

3/73

Difficult to compare, since they make different assumptions:

- index-only: needs an appropriate index
- hash-based: needs buffers and good hash functions
- sort-based: needs only buffers ⇒ use as default

Best case scenario for each (assuming $n+1$ in-memory buffers):

- index-only: $b_i + b_{Out} \ll b_R + b_{Out}$
- hash-based: $b_R + 2.b_P + b_{Out}$
- sort-based: $b_R + b_T + 2.b_T.ceil(\log_n b_0) + b_T + b_{Out}$

We normally omit b_{Out}, since each method produces the same result

Projection in PostgreSQL

4/73

Code for projection forms part of execution iterators:

- backend/executor/execQual.c

Functions involved with projection:

- **`ExecProject(projInfo,...)`** ... extracts projected data

- `check_sql_fn_retval(...)` ... makes new tuple via TargetList
- `ExecStoreTuple(newTuple,...)` ... save tuple in buffer

plus many many others ...

Implementing Selection

Varieties of Selection

6/73

Selection: `select * from R where C`

- filters a subset of tuples from one relation `R`
- based on a condition `C` on the attribute values

We consider three distinct styles of selection:

- 1-d (one dimensional) (condition uses only *1* attribute)
- *n*-d (multi-dimensional) (condition uses *>1* attribute)
- similarity (approximate matching, with ranking)

Each style has several possible file-structures/techniques.

... Varieties of Selection

7/73

Examples of different selection types:

- *one*: `select * from R where id = 1234`
- *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`

  note: *rng = range*
- *sim*: `select * from R where name like '%oo%'`

  `select * from Images where similar to SampleImage`

Exercise 1: Query Types

8/73

Using the relation:

All primary key has an index

```
create table Courses (
   id       integer primary key,
   code     char(8),  -- e.g. 'COMP9315'
   title    text,     -- e.g. 'Computing 1'
   year     integer,  -- e.g. 2000..2016
   convenor integer references Staff(id),
   constraint once_per_year unique (code,year)
);
```

give examples of each of the following query types:

1. *a 1-d one query, an n-d one query*
2. *a 1-d pmr query, an n-d pmr query*
3. *a 1-d range query, an n-d range query*

Suggest how many solutions each might produce ...

Implementing Select Efficiently

9/73

Two basic approaches:

- *physical arrangement of tuples*
  - *sorting* *(search strategy)*
  - *hashing* *(static, dynamic, n-dimensional)*
- *additional indexing information*
  - *index files* *(primary, secondary, trees)*
  - *signatures* *(superimposed, disjoint)*

Our analysis assumes 1 input buffer available for each relation.

If more buffers are available, most methods benefit.

Heap Files

*Note: this is **not** "heap" as in the top-to-bottom ordered tree.*
It means simply an unordered collection of tuples in a file.

Selection in Heaps

11/73

For all selection queries, the only possible strategy is:

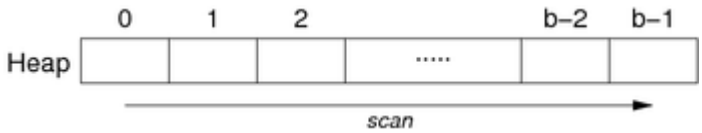
```
// select * from R where C
for each page P in file of relation R {
    for each tuple t in page P {
        if (t satisfies C)
            add tuple t to result set
    }
}
```

i.e. linear scan through file searching for matching tuples

... Selection in Heaps

12/73

The heap is scanned from the first to the last page:



$Cost_{range} = Cost_{pmr} = b$

If we know that only one tuple matches the query (one query),
a simple optimisation is to stop the scan once that tuple is found.

$Cost_{one}$: *Best = 1* *Average = b/2* *Worst = b*

Insertion in Heaps

13/73

Insertion: new tuple is appended to file (in last page).

```
rel = openRelation("R", READ|WRITE);
pid = nPages(rel)-1;
get_page(rel, pid, buf);
if (size(newTup) > size(buf))
   { deal with oversize tuple }
else {
   if (!hasSpace(buf,newTup))
      { pid++; nPages(rel)++; clear(buf); }
   insert_record(buf,newTup);
   put_page(rel, pid, buf);
}
```

Always insert tuple at last pages

$Cost_{insert} = 1_r + 1_w$

... Insertion in Heaps

Alternative strategy:

- *find any page from `R` with enough space*
- *preferably a page already loaded into memory buffer*

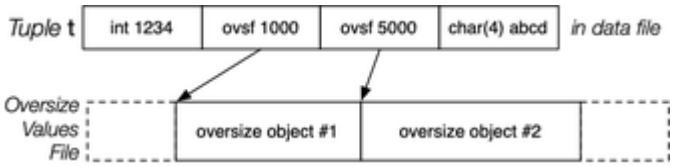
PostgreSQL's strategy:

- *use last updated page of `R` in buffer pool*
- *otherwise, search buffer pool for page with enough space*
- *assisted by free space map (FSM) associated with each table*
- *for details: **`backend/access/heap/{heapam.c,hio.c}`***

... Insertion in Heaps

Dealing with oversize tuple `t`:

```
for i in 1 .. nAttr(t) {
   if (t[i] not oversized) continue
   off = appendToFile(ovsf, t[i])
   t[i] = (OVERSIZE, off)
}
insert into buf as before
```



... Insertion in Heaps

PostgreSQL's tuple insertion:

```
heap_insert(Relation relation,     // relation desc
            HeapTuple newtup,      // new tuple data
            CommandId cid, ...)    // SQL statement
```

- *finds page which has enough free space for `newtup`*
- *ensures page loaded into buffer pool and locked*
- *copies tuple data into page buffer, sets `xmin`, etc.*
- *marks buffer as dirty*
- *writes details of insertion into transaction log*
- *returns OID of new tuple if relation has OIDs*

Deletion in Heaps

SQL: `delete from R where Condition`

Implementation of deletion:

```
rel = openRelation("R",READ|WRITE);
for (p = 0; p < nPages(rel); p++) {
    get_page(rel, p, buf);
    ndels = 0;
    for (i = 0; i < nTuples(buf); i++) {
        tup = get_record(buf,i);
        if (tup satisfies Condition)
            { ndels++; delete_record(buf,i); }
    }
```

```
    if (ndels > 0) put_page(rel, p, buf);
    if (ndels > 0 && unique) break;
}
```

Exercise 2: Cost of Deletion in Heaps

Consider the following queries ...

```
delete from Employees where id = 12345  -- one
delete from Employees where dept = 'Marketing'  -- pmr
delete from Employees where 40 <= age and age < 50  -- range
```

Show how each will be executed and estimate the cost, assuming:

- $b = 100,\ b_{q2} = 3,\ b_{q3} = 20$

State any other assumptions.

... Deletion in Heaps

PostgreSQL tuple deletion:

```
heap_delete(Relation relation,    // relation desc
            ItemPointer tid, ..., // tupleID
            CommandId cid, ...)   // SQL statement
```

- *gets page containing tuple `tid` into buffer pool and locks it*
- *sets flags, commandID and `xmax` in tuple; dirties buffer*
- *writes indication of deletion to transaction log*

Vacuuming eventually compacts space in each page.

Updates in Heaps == delete + insert

SQL: `update R set F = val where Condition`

Analysis for updates is similar to that for deletion

- *scan all pages*
- *replace any updated tuples (within each page)*
- *write affected pages to disk*

$Cost_{update} = b_r + b_{qw}$

Complication: new tuple larger than old version (too big for page)

Solution: delete, re-organise free space, then insert

... Updates in Heaps

PostgreSQL tuple update:

```
heap_update(Relation relation,     // relation desc
            ItemPointer otid,      // old tupleID
            HeapTuple newtup, ..., // new tuple data
            CommandId cid, ...)    // SQL statement
```

- *essentially does `delete(otid)`, then `insert(newtup)`*
- *also, sets old tuple's `ctid` field to reference new tuple*
- *can also update-in-place if no referencing transactions*

Heaps in PostgreSQL

PostgreSQL stores all table data in heap files (by default).

Typically there are also associated index files.

If a file is more useful in some other form:

- *PostgreSQL may make a transformed copy during query execution*
- *programmer can set it via* `create index...using hash`

Heap file implementation: *src/backend/access/heap*

... Heaps in PostgreSQL

23/73

PostgreSQL "heap file" may use multiple physical files

- *files are named after the OID of the corresponding table*
- *first data file is called simply* `OID`
- *if size exceeds 1GB, create a fork called* `OID.1`
- *add more forks as data size grows (one fork for each 1GB)*
- *other files:*
  - *free space map (`OID_fsm`), visibility map (`OID_vm`)*
  - *optionally, TOAST file (if table has varlen attributes)*
- *for details: Chapter 68 in PostgreSQL v12 documentation*

Sorted Files

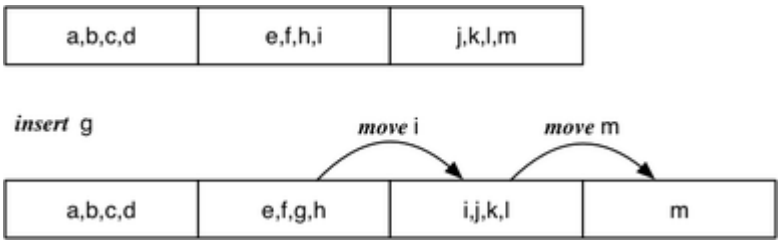
Sorted Files

25/73

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

Makes searching more efficient; makes insertion less efficient

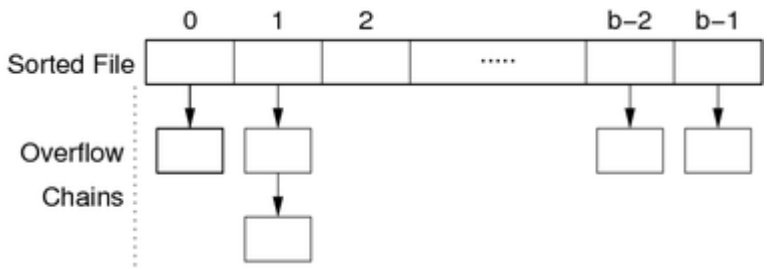
E.g. assume c = 4



... Sorted Files

26/73

In order to mitigate insertion costs, use overflow pages.



Total number of overflow pages = b_{ov}.

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

Bucket = data page + its overflow page(s)

keep records of the max and min value of each page(bucket, including the overflow page)

Selection in Sorted Files

27/73

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) / 2;  // int division with truncation
    (tup,loVal,hiVal) = searchBucket(f,mid,x,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

28/73

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

Note: `getPage(f,pid) = { read_page(relOf(f),pid,buf); return buf; }`

... Selection in Sorted Files

29/73

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

30/73

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 needs assumptions (e.g. data distribution)

Exercise 3: Searching in Sorted File

31/73

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` 1 ? ? ?
- `select * from R where k = 3` 3
- `select * from R where k = 14`
- `select max(k) from R` 1

Exercise 4: Optimising Sorted-file Search

32/73

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.

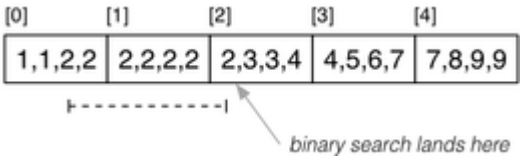
... Selection in Sorted Files

33/73

For pmr query, on non-unique attribute k, where file is sorted on k

- *tuples containing k may span several pages*

E.g. `select * from R where k = 2`



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

Scan backwards and forwards from p to find matches.

Thus, $Cost_{pmr} = Cost_{one} + (b_q-1).(1+Ov)$ What is b_q here? Here b_q = 3

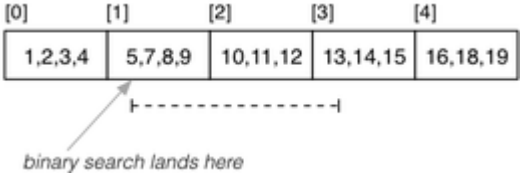
... Selection in Sorted Files

34/73

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

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

E.g. `select * from R where k >= 5 and k <= 13`



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

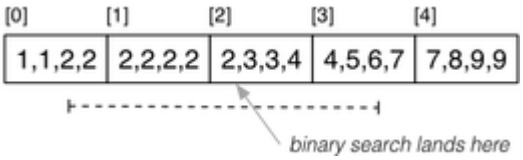
... Selection in Sorted Files

35/73

For range queries on non-unique sort key, similar method to pmr.

- *binary search to find lower bound*
- *then go backwards to start of run*
- *then go forwards to last occurence of upper-bound*

E.g. `select * from R where k >= 2 and k <= 6`



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

... Selection in Sorted Files

36/73

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

But what about `select * from R where j = 100.0` *?*

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*

Insertion into Sorted Files

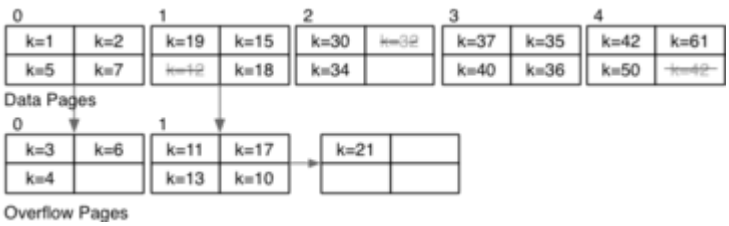
37/73

***Insertion** approach:*

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

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

Consider insertions of k=33, k=25, k=99 into:



Deletion from Sorted Files

38/73

E.g. `delete from R where k = 2`

Deletion strategy:

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

Cost depends on *selectivity* of selection condition

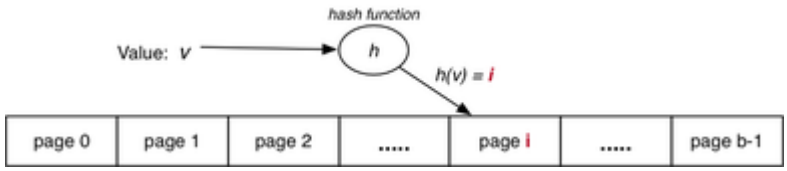
Recall: selectivity determines b_q (# pages with matches)

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):

```
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()`)

... Hashing

`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

r: total number of tuples

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 $0.75 \leq L \leq 0.9$.

Selection with Hashing

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

```
// select * from R where k = val
P = getPageViaHash(val,R)
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

Working out which page, given a key ...

```
Page getPageViaHash(Value key, Reln R)
{
   Page p; // eventually references a buffer
   uint32 h = hash_any(key, len(key));
   PageID pid = h % nPages(R);
   p = getPage(dataFile(R), pid);
   return p;
}
```

... Selection with Hashing

47/73

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

```
// select * from R where nk = val
P = getPageViaHash(val,R)
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

48/73

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

49/73

Insertion uses similar process to *one* queries.

```
// insert tuple t with key=val into rel R
P = getPageViaHash(val,R)
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: $1+max(OvLen))_r + 2_w$

Need to add a pointer for the new–added page

Exercise 5: Insertion into Static Hashed File

50/73

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

51/73

Similar performance to select on non-unique key:

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

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

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

Problem with Hashing...

52/73

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

53/73

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 6: Bit Manipulation

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

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

   Analogous to `gets()` (assumes supplied buffer is 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

---

... Flexible Hashing

Important concept for flexible hashing: *splitting*

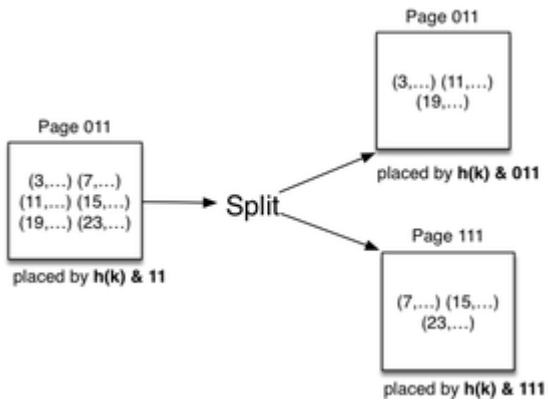
- consider one page (all tuples have same hash value)
- recompute page numbers by considering one extra bit
- if current page is `101`, new pages have hashes `0101` and `1101`
- some tuples stay in page `0101` (was `101`)
- some tuples move to page `1101` (new page)
- also, rehash any tuples in overflow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

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

---

... Flexible Hashing

Example of splitting:



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

Linear Hashing

File organisation:

- file of primary data pages
- file of overflow data pages
- a register called the *split pointer* (sp)

Uses systematic method of growing data file ...

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

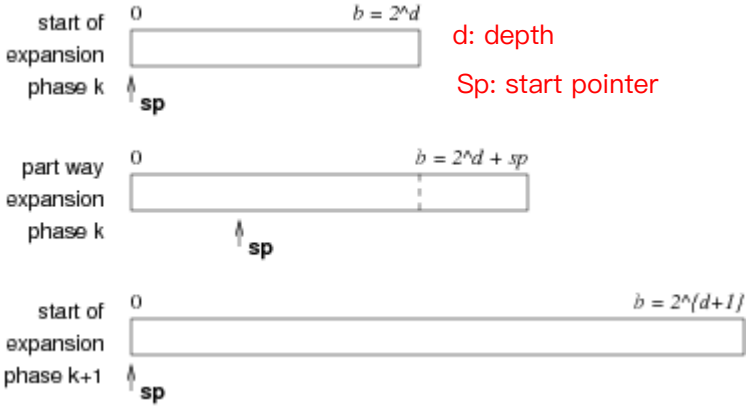
Advantage: does *not* require auxiliary storage for a directory

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

... Linear Hashing

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

Has "phases" of expansion; over 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 page 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 (tuples in *B* may eventually move into it).

... Selection with Lin.Hashing

Modified search algorithm:

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

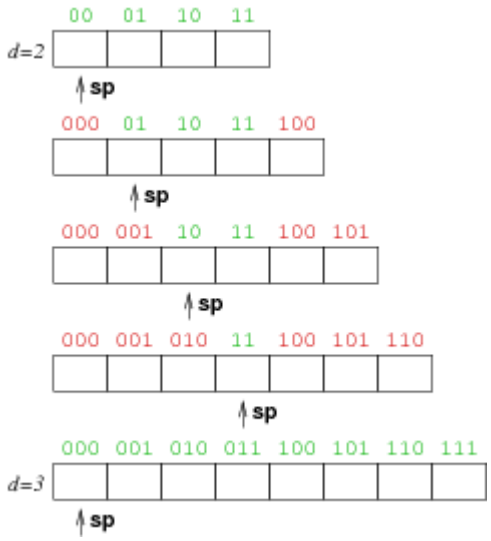
Require a function to extract *d* bits from bit-string:

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

```
    if (t.k == val) return R;
}
```

File Expansion with Lin.Hashing

62/73



Insertion with Lin.Hashing

63/73

Abstract view:

```
pid = bits(d,hash(val));
if (pid < sp) pid = bits(d+1,hash(val));
// bucket P = page P + its overflow pages
P = getPage(f,pid)
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

64/73

How to decide that we "need to split"?

Two approaches to triggering a split:

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

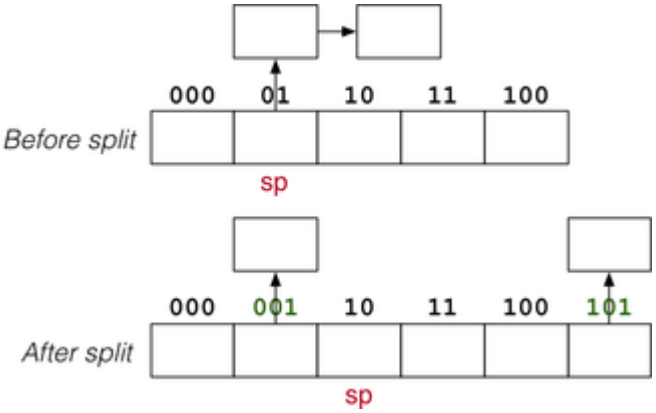
Note: always split page *sp*, even if not full or "current"

Systematic splitting like this ...

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

... Splitting

Splitting process for page *sp*=01:



Exercise 7: Insertion into Linear Hashed File

4 pages, each page has 3 tuples

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.

... Splitting

Splitting algorithm:

```
// partition tuples between two buckets
newp = sp + 2^d; oldp = sp;
for all tuples t in P[oldp] and its overflows {
    p = bits(d+1,hash(t.k));
    if (p == newp)
        add tuple t to bucket[newp]
    else
        add tuple t to bucket[oldp]
}
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 page *sp* (plus all of its overflow pages)
- write page *sp* (and its new overflow pages)
- write page $sp+2^d$ (and its new overflow pages)

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).
- merge tuples from bucket with its buddy page (using d-1 hash bits)

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

PostgreSQL uses slightly different file organisation ...

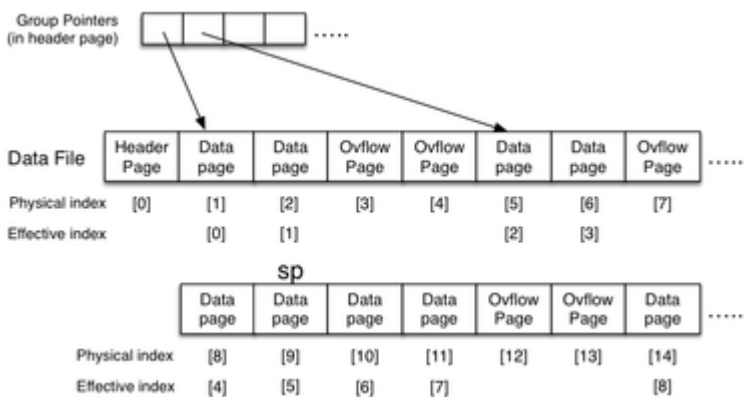
- has a single file containing main and overflow pages
- has groups of main pages of size 2^n
- in between groups, arbitrary number of overflow pages
- maintains collection of group pointers in header page
- each group pointer indicates start of main page group

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

Confusingly, PostgreSQL calls "group pointers" as "split pointers"

... Hash Files in PostgreSQL

PostgreSQL hash file structure:



... Hash Files in PostgreSQL

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: 17 Mar 2020