Week 4

- Week 04

❖ Week 04

Things to Note ...

- Quiz 2 due before Friday 10 March, 9pm
- Quizzes are *fortnightly* (weeks 2,4,6,8,10)

This Week ...

- Projection
- Selection (heaps, sorted files)
- 1d Selection (hashing)

Coming Up ...

- 1d Selection (indexing), Nd Selection

Produced: 23 Feb 2023

Implementing Projection

❖ The Projection Operation

Consider the query:

```
select distinct name,age from Employee;
```

If the **`Employee`** relation has four tuples such as:

```
(94002, John, Sales, Manager,   32)
(95212, Jane, Admin, Manager,   39)
(96341, John, Admin, Secretary, 32)
(91234, Jane, Admin, Secretary, 21)
```

then the result of the projection is:

```
(Jane, 21)   (Jane, 39)   (John, 32)
```

Note that duplicate tuples (e.g. **`(John,32)`**) are eliminated.

❖ The Projection Operation (cont)

Relies on function **`Tuple projTuple(AttrList, Tuple)`**

- first arg is list of attributes
- second arg is a tuple containing those attributes (and more)
- return value is a new tuple containing only those attributes

Examples, using tuples of type **`(id:int, name:text, degree:int)`**

```
projTuple([id], (1234,'John',3778))
    returns (id=1234)

projTuple([name,degree]), (1234,'John',3778))
    returns (name='John',degree=3778)
```

❖ The Projection Operation (cont)

Without **`distinct`**, projection is straightforward

```
// attrs = [attr1,attr2,...]
bR = nPages(Rel)
for i in 0 .. bR-1 {
   P = read page i
   for j in 0 .. nTuples(P)-1 {
      T = getTuple(P,j)
      T' = projTuple(attrs, T)
      if (outBuf is full) write and clear
      append T' to outBuf
   }
}
if (nTuples(outBuf) > 0) write
```

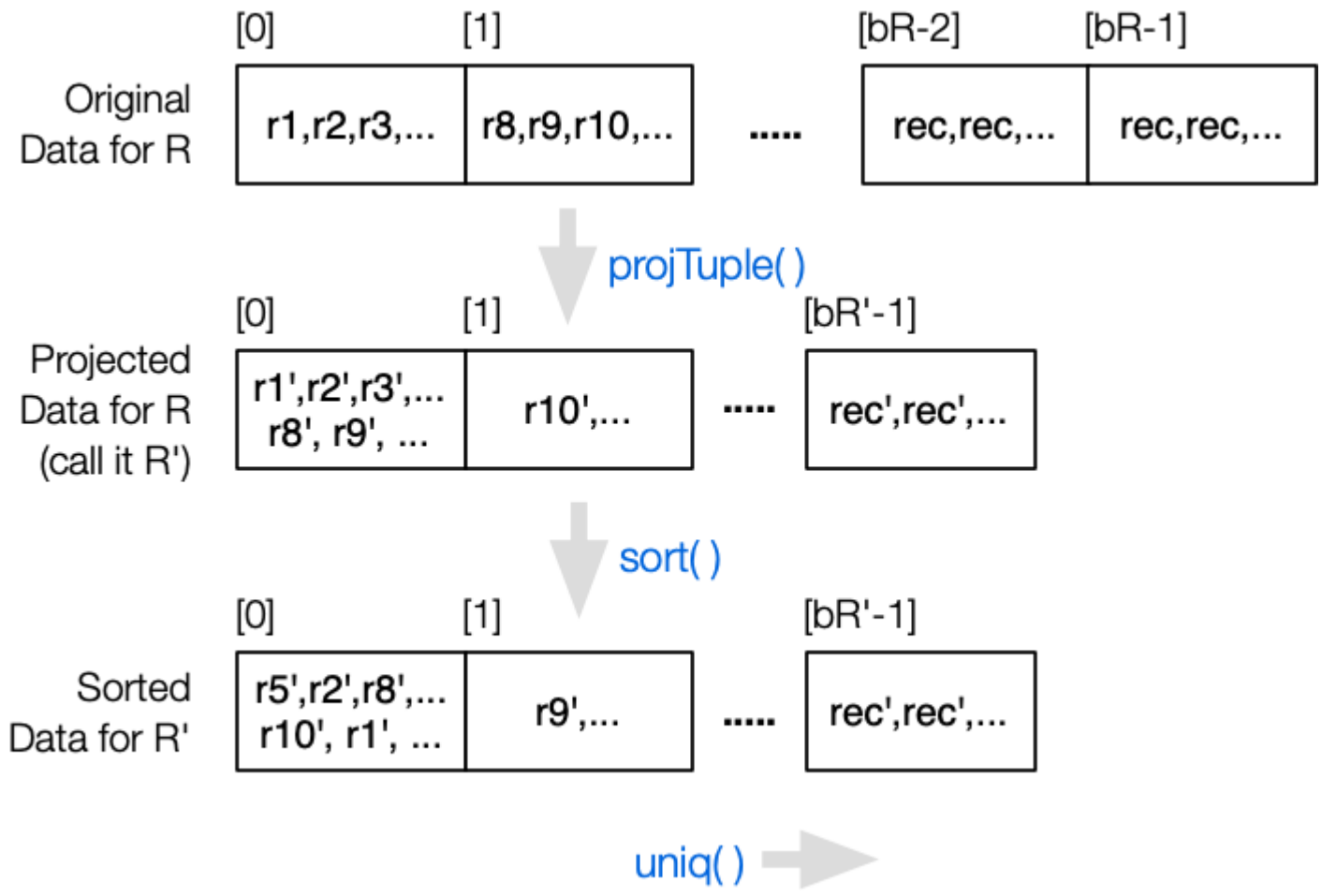
Typically, $b_{OutFile} < b_{InFile}$ (same number of tuples, but tuples are smaller)

❖ The Projection Operation (cont)

With **`distinct`**, the projection operation needs to:

1. scan the entire relation as input
   - already seen how to do scanning
2. create output tuples containing only requested attributes
   - implementation depends on tuple internal structure
   - essentially, make a new tuple with fewer attributes
     and where the values may be computed from existing attributes
3. eliminate any duplicates produced
   - two approaches: sorting or hashing

❖ Sort-based Projection



❖ Sort-based Projection (cont)

Requires a temporary file/relation .

```
for each tuple T in RelFile {
    T' = projTuple([attr1,attr2,...],T)
    add T' to TempFile
}

sort TempFile on [attrs]

for each tuple T in TempFile {
    if (T == Prev) continue
    write T to Result
    Prev = T
}
```

Reminder: "**`for each tuple`**" means page-by-page, tuple-by-tuple

❖ Cost of Sort-based Projection

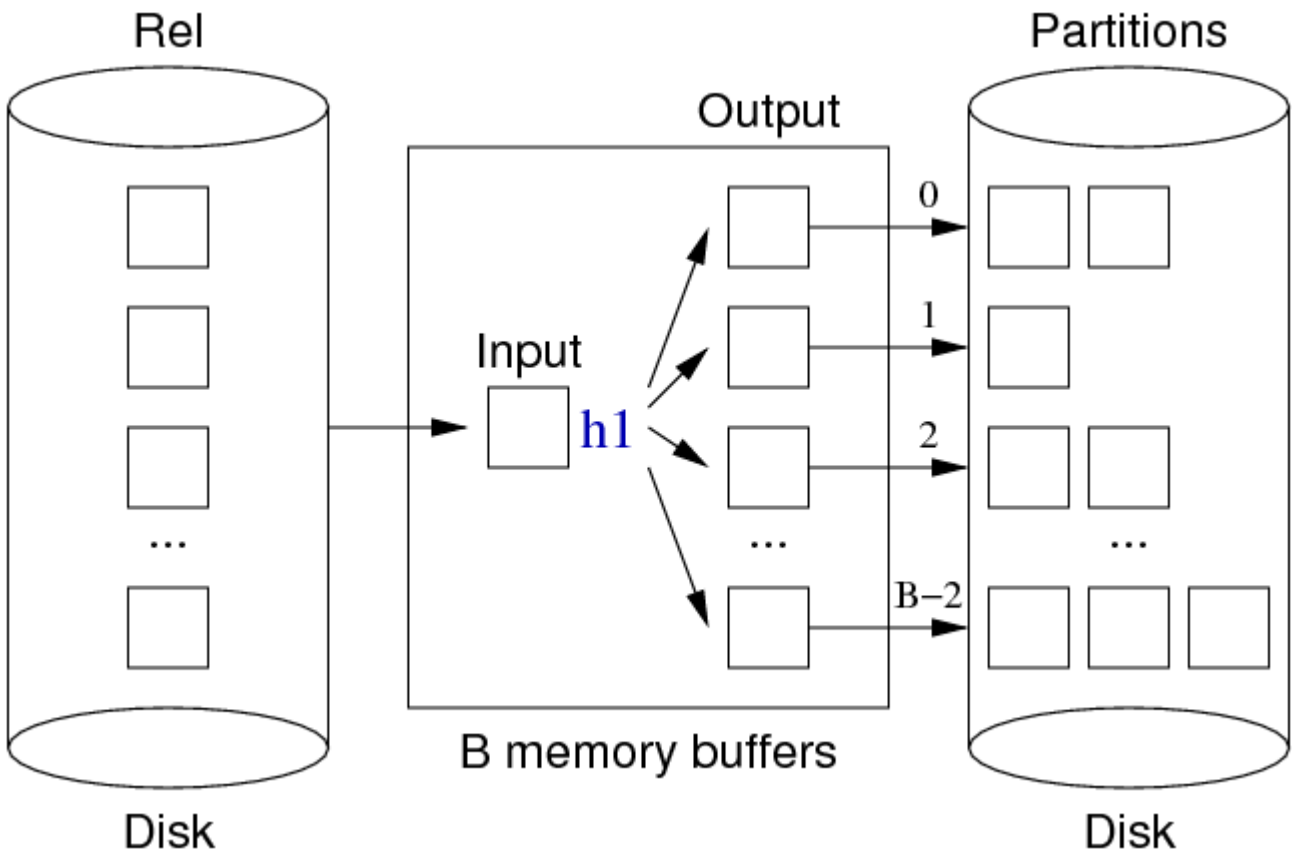
The costs involved are (assuming $B=n+1$ buffers for sort):

- scanning original relation **`Rel`**: b_R (with c_R)
- writing **`Temp`** relation: b_T (smaller tuples, $c_T > c_R$, sorted)
- sorting **`Temp`** relation:
  $2.b_T.ceil(\log_n b_0)$ where $b_0 = ceil(b_T/B)$
- scanning **`Temp`**, removing duplicates: b_T
- writing the result relation: b_{Out} (maybe less tuples)

Cost = sum of above = $b_R + b_T + 2.b_T.ceil(\log_n b_0) + b_T + b_{Out}$

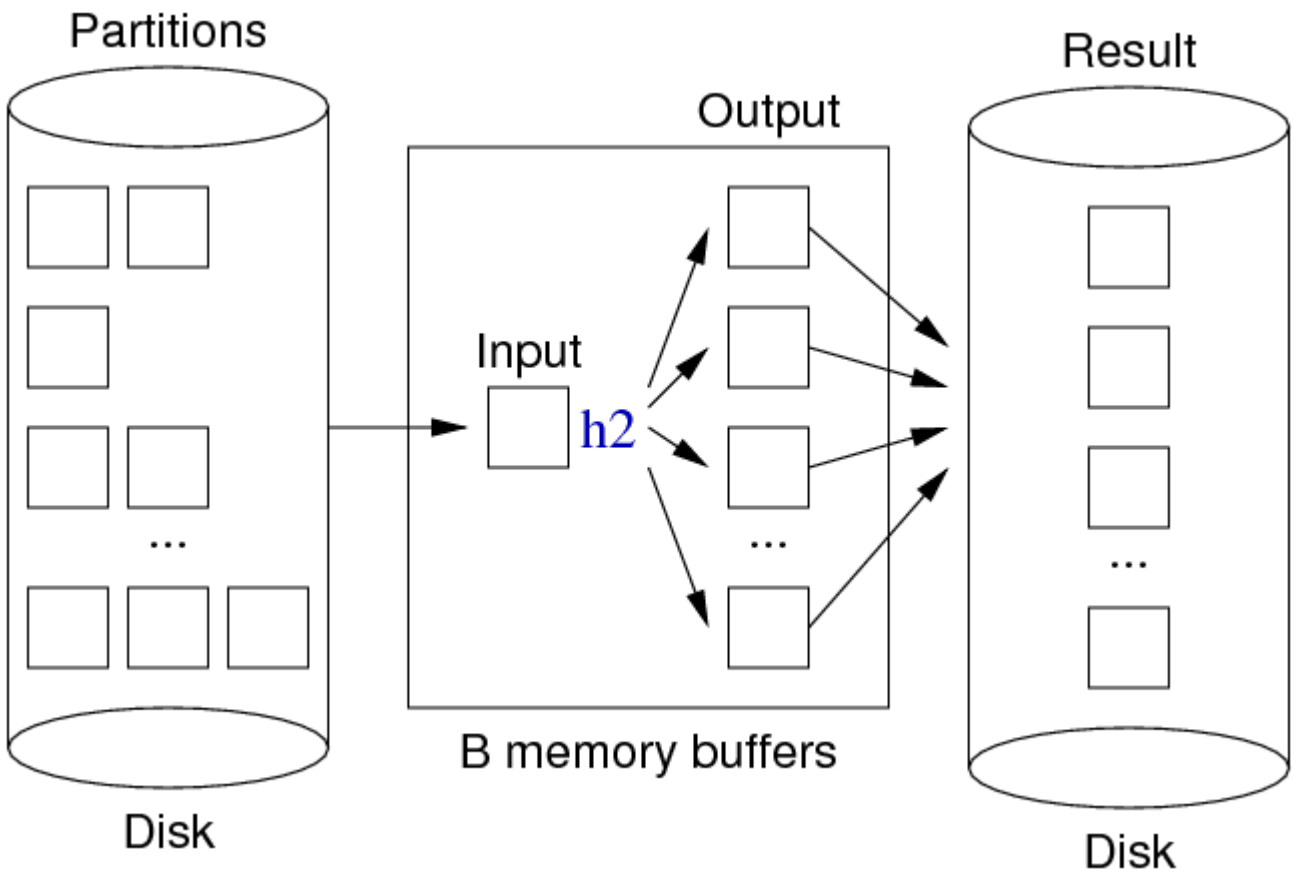
❖ Hash-based Projection

Partitioning phase:



❖ Hash-based Projection (cont)

Duplicate elimination phase:



❖ Hash-based Projection (cont)

Algorithm for both phases:

```
for each tuple T in relation Rel {
    T' = mkTuple(attrs,T)
    H = h1(T', n)
    B = buffer for partition[H]
    if (B full) write and clear B
    insert T' into B
}
for each partition P in 0..n-1 {
    for each tuple T in partition P {
        H = h2(T, n)
        B = buffer for hash value H
        if (T not in B) insert T into B
        // assumes B never gets full
    }
    write and clear all buffers
}
```

Reminder: "**`for each tuple`**" means page-by-page, tuple-by-tuple

❖ Cost of Hash-based Projection

The total cost is the sum of the following:

- scanning original relation **R**: b_R
- writing partitions: b_P (b_R vs b_P ?)
- re-reading partitions: b_P
- writing the result relation: b_{Out}

Cost = $b_R + 2b_P + b_{Out}$

To ensure that *n* is larger than the largest partition ...

- use hash functions (h1,h2) with uniform spread
- allocate at least $sqrt(b_R)+1$ buffers
- if insufficient buffers, significant re-reading overhead

❖ Projection on Primary Key

No duplicates, so simple approach from above works:

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

❖ Index-only Projection

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 $b_i < b_R$)
- Cost = b_i reads + b_{Out} writes

❖ Comparison of Projection Methods

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: $b_i + b_{Out} < b_R + b_{Out}$
- 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} ... each method produces the same result

❖ Projection in PostgreSQL

Code for projection forms part of execution iterators:

- include/nodes/execnodes.h
- backend/executor/execQual.c

Types:

- **`ProjectionInfo { type, pi_state, pi_exprContext }`**
- **`ExprState { tag, flags, resnull, resvalue, ... }`**

Functions:

- **`ExecProject(projInfo,...)`** ... extracts projected data
- **`check_sql_fn_retval(...)`** ... evaluates attribute value

Produced: 22 Feb 2023

Selection Overview

- Varieties of Selection
- Implementing Select Efficiently

❖ Varieties of Selection

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:

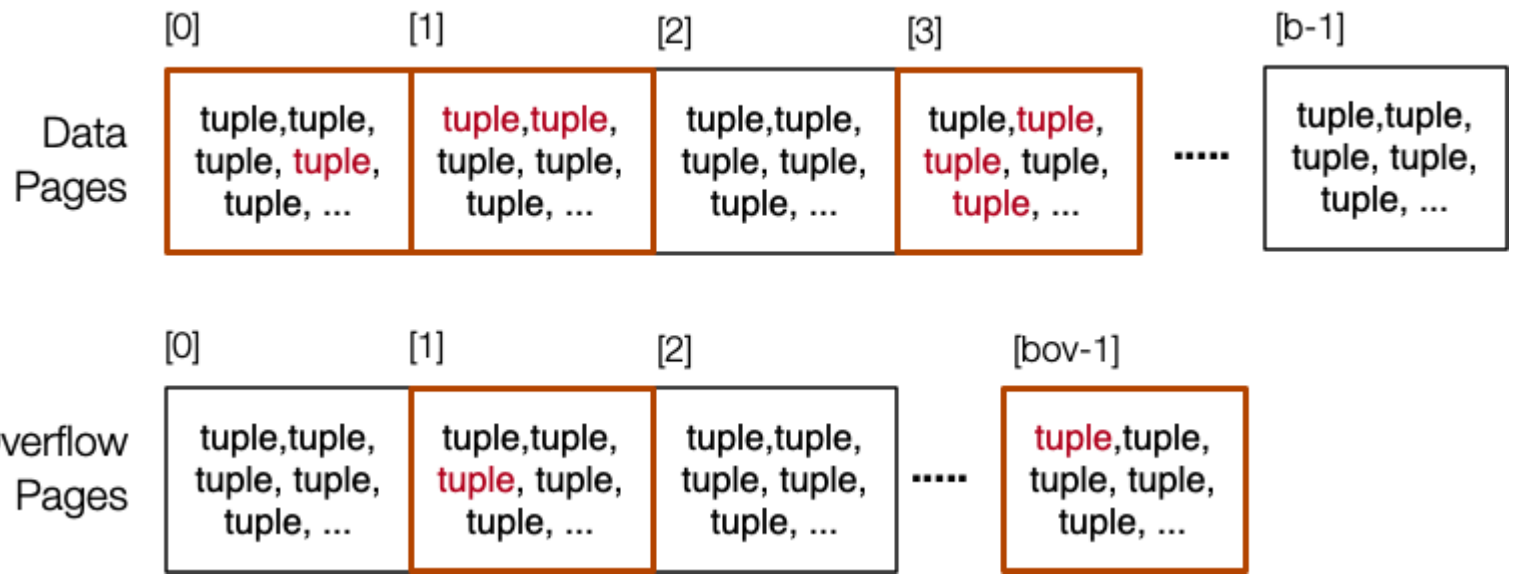
- 1-d (one dimensional) (condition uses only *1* attribute)
- *n*-d (multi-dimensional) (condition uses *>1* attribute)
- similarity (approximate matching, with ranking)

Each style has several possible file-structures/techniques.

❖ Varieties of Selection (cont)

Selection returns a subset of tuples from a table

- r_q = number of tuples that match query *q*
- b_q = number of pages containing tuples that match query *q*



In the diagram, $r_q = 8$, $b_q = 5$

❖ Varieties of Selection (cont)

Different categories of selection queries:

one ... queries with at most 1 result ... $0 \le r_q \le 1$, $0 \le b_q \le 1$

- typically, equality test on primary key attribute, e.g.
- **`select * from R where id = 1234`**

pmr ... partial match retrieval ... $0 \le r_q \le r$, $0 \le b_q \le b+b_{ov}$

- conjunction of equality tests on multiple attributes, e.g.
- **`select * from R where age=65`** (1-d)
- **`select * from R where age=65 and gender='m'`** (n-d)

❖ Varieties of Selection (cont)

More categories of selection queries:

rng ... range queries ... $0 \le r_q \le r$, $0 \le b_q \le b+b_{ov}$

- conjunction of inequalities, on one or more attributes, e.g.
- **`select * from R where age≥18 and age≤21`** (1-d)
- **`select * from R where 18≤age≤21 and 160≤height≤190`** (n-d)

pat ... pattern-based queries ... $0 \le r_q \le r$, $0 \le b_q \le b+b_{ov}$

- string-based matching using **`like`** or regular expressions
- **`select * from R where name like '%oo%'`**
- **`select * from R where name ~ '^Smi'`**

❖ Varieties of Selection (cont)

More categories of selection queries:

sim ... similarity matching ... in theory, $r_q = r$ ... everything matches to some degree

- uses "similarity" measure (0 ≤ *sim* ≤ 1, 0=different, 1=identical)
- **`select * from`** *Images* **`where similar to`** *SampleImage*
- results are ranked by *sim* value, from most to least similar
- can become a filter via
  - threshold ... only items where *sim* ≥ min similarity
  - top-k ... *k* items with highest similarities

We focus on one, pmr and rng queries, but will discuss others

❖ Implementing Select Efficiently

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.

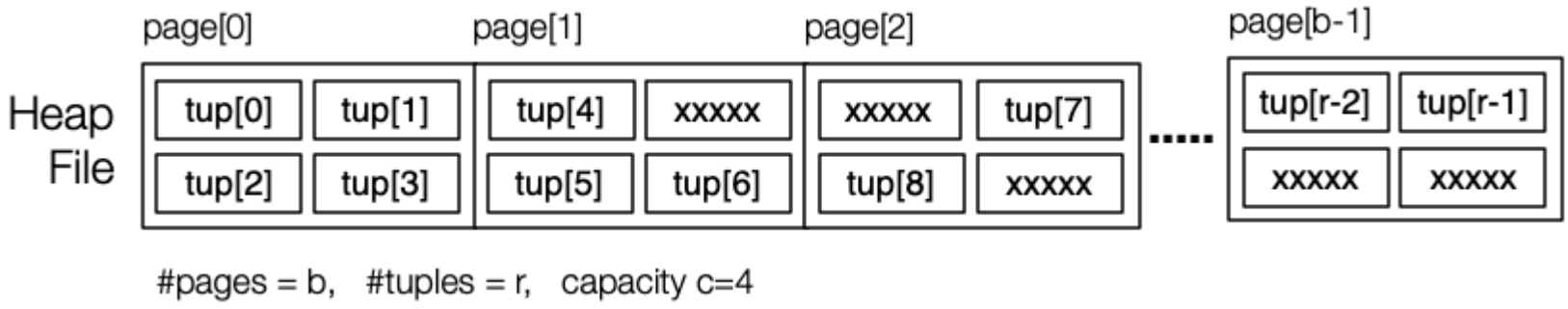
Produced: 22 Feb 2023

Heap Files

❖ Heap Files

Heap files

- sequence of pages containing tuples
- no inherent ordering of tuples (added in next free slot)
- pages may contain free space from deleted tuples
- does not generally involve overflow pages



Note: this is **not** "heap" as in the top-to-bottom ordered tree.

❖ Selection in Heaps

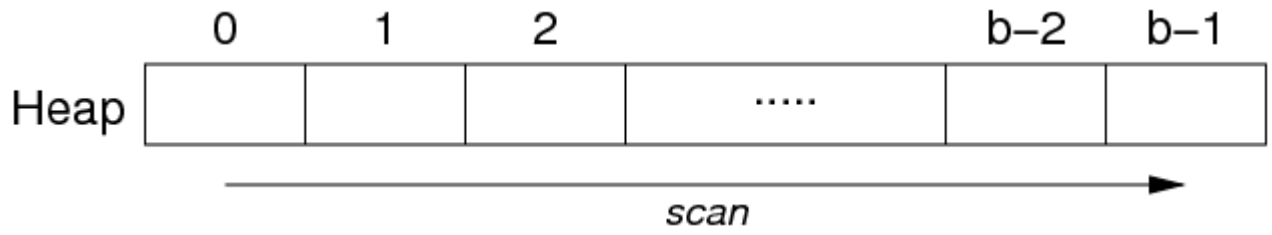
For all selection queries, the only possible strategy is:

```
// select * from R where C
rel = openRelation("R", READ);
for (p = 0; p < nPages(rel); p++) {
    get_page(rel, p, buf);
    for (i = 0; i < nTuples(buf); i++) {
        T = get_tuple(buf, i);
        if (T satisfies C)
            add tuple T to result set
    }
}
```

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

❖ Selection in Heaps (cont)

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

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);
}
```

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

❖ Insertion in Heaps (cont)

Alternative strategy:

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

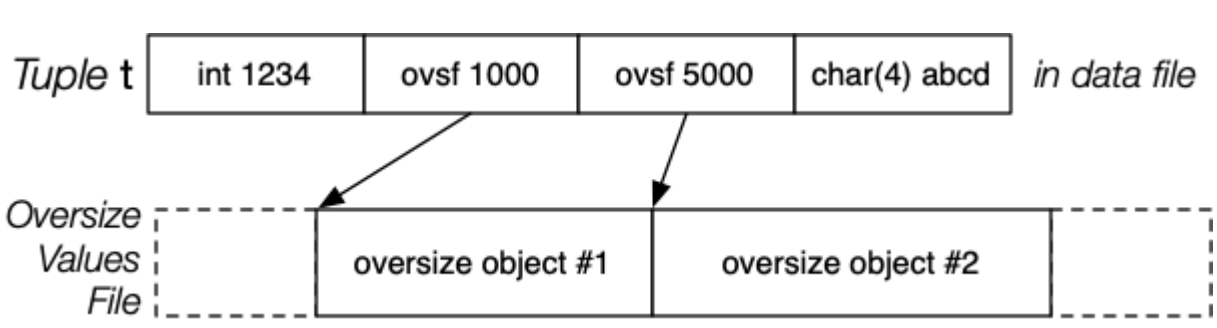
PostgreSQL's strategy:

- 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 (cont)

Dealing with oversize tuple **t**:

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



❖ Insertion in Heaps (cont)

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_tuple(buf,i);
        if (tup satisfies Condition)
            { ndels++; delete_record(buf,i); }
    }
    if (ndels > 0) put_page(rel, p, buf);
    if (ndels > 0 && unique) break;
}
```

❖ Deletion in Heaps (cont)

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

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 (cont)

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 (cont)

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 large varlen attributes)
- for details: Chapter 72 in PostgreSQL v15 documentation

Produced: 6 Mar 2023

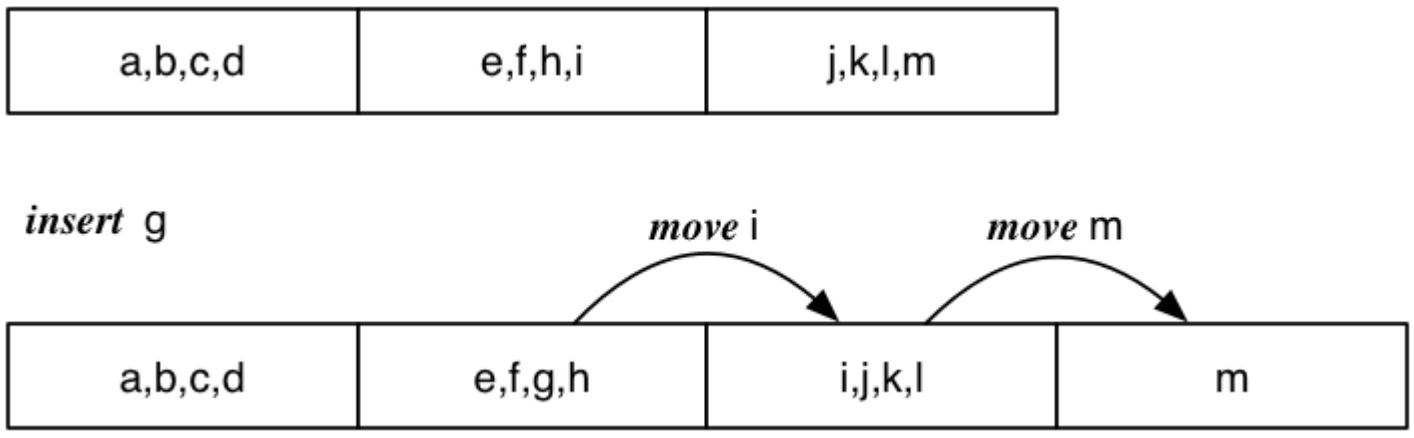
Sorted Files

❖ Sorted Files

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

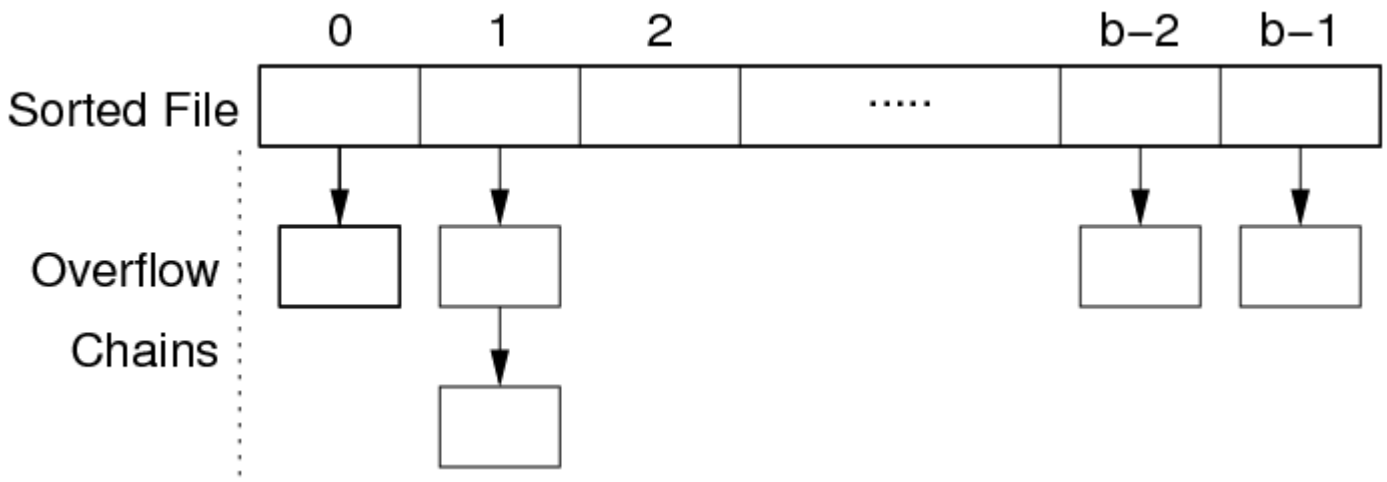
Makes searching more efficient; makes insertion less efficient

E.g. assume *c = 4*



❖ Sorted Files (cont)

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)

❖ 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 = nPages(rel)-1
while (lo <= hi) {
    mid = (lo+hi) / 2;  // int division with truncation
    (tup,loVal,hiVal) = searchBucket(rel,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 **`rel`** is relation handle, **`mid,lo,hi`** are page indexes,
`k` is a field/attr, **`val,loVal,hiVal`** are values for **`k`**

❖ Selection in Sorted Files (cont)

Search a page and its overflow chain for a key value

```
searchBucket(rel,p,k,val)
{
    get_page(rel,p,buf);
    (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) {
        get_page(ovf,ovp,buf);
        (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 (cont)

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 = get_tuple(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 (cont)

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 = $log_2\, b + b_{ov}$

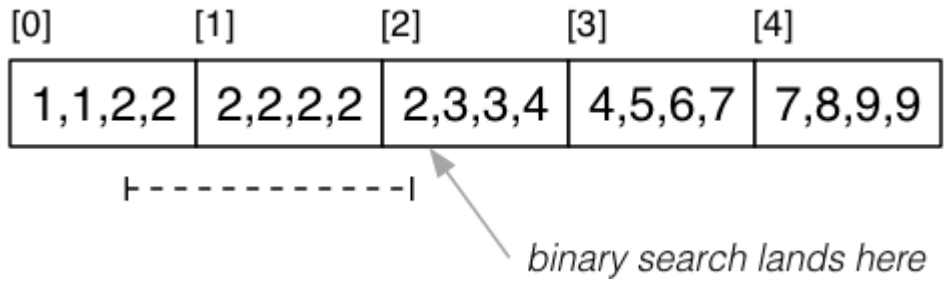
Average case cost analysis needs assumptions (e.g. data distribution)

❖ Selection in Sorted Files (cont)

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.

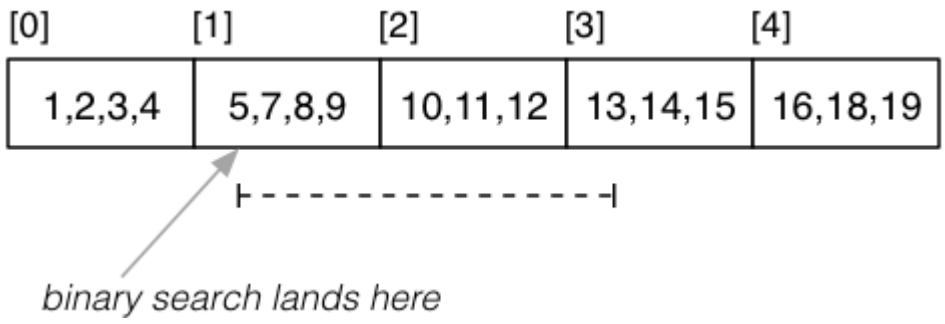
Thus, $Cost_{pmr} = Cost_{one} + (b_q\text{-}1).(1+Ov)$

❖ Selection in Sorted Files (cont)

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`**



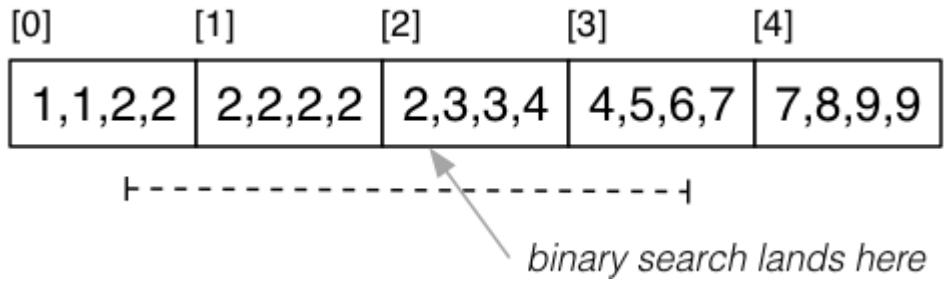
$Cost_{range} = Cost_{one} + (b_q\text{-}1).(1+Ov)$

❖ Selection in Sorted Files (cont)

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`**



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

❖ Selection in Sorted Files (cont)

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

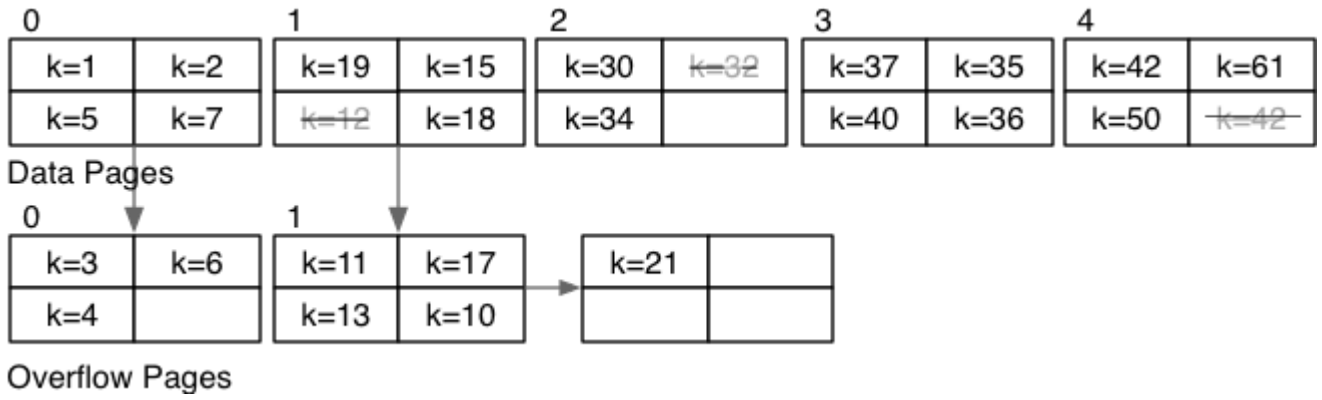
❖ Insertion into Sorted Files

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

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

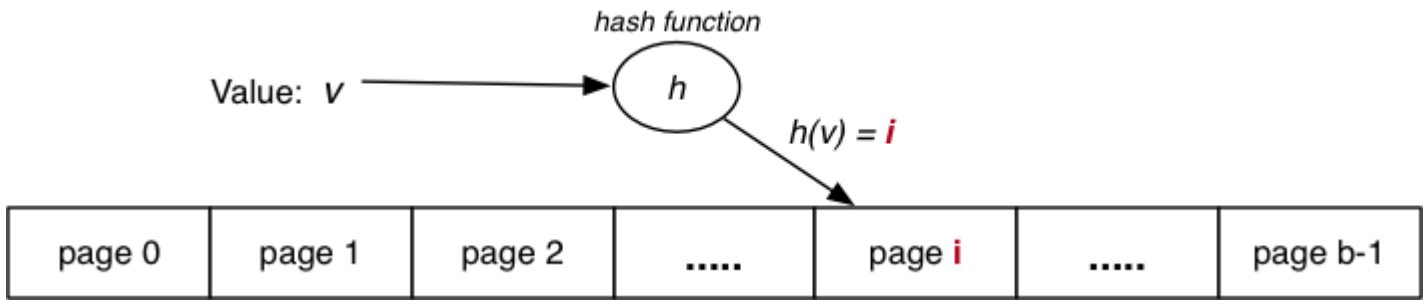
Produced: 22 Feb 2023

Hashed Files

- Hashing
- Hashing Performance
- Selection with Hashing
- Insertion with Hashing
- Deletion with Hashing
- Problem with Hashing...
- Flexible Hashing

❖ 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()**)

❖ 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	*≅ 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
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? ** ... (integer values vs float values)

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

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

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

** what if 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)

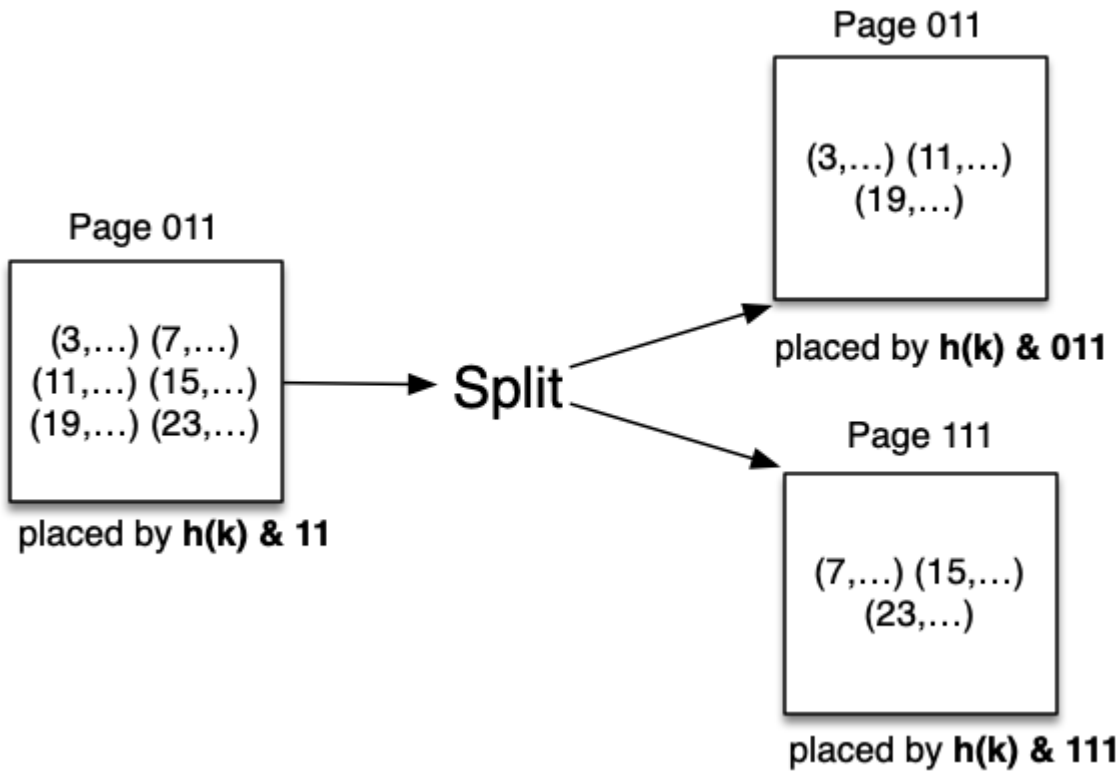
Important concept for flexible hashing: splitting

- 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*

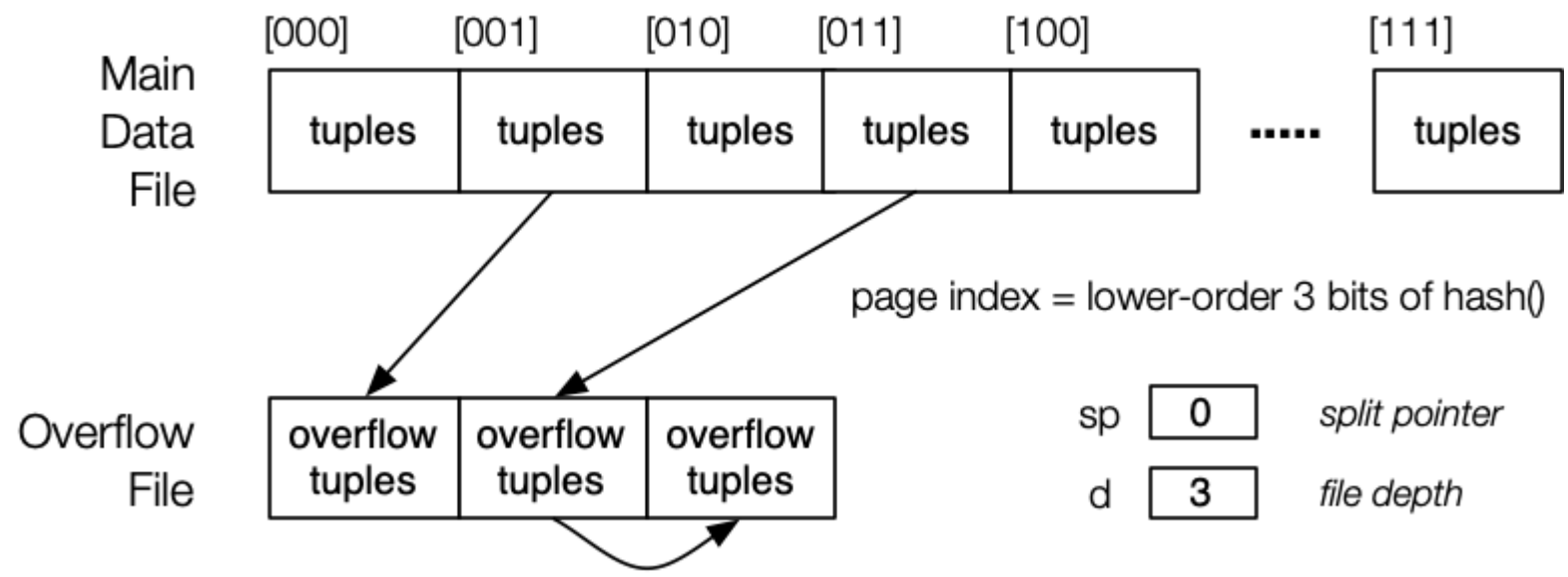
Produced: 6 Mar 2023

Linear Hashing

❖ Linear Hashing

File organisation:

- file of primary data pages
- file of overflow data pages
- registers called *split pointer* (sp) and *depth* (d)



❖ Linear Hashing (cont)

Linear Hashing uses a systematic method of growing data file ...

- hash function "adapts" to changing address range (via *sp* and *d*)
- systematic splitting controls length of overflow chains

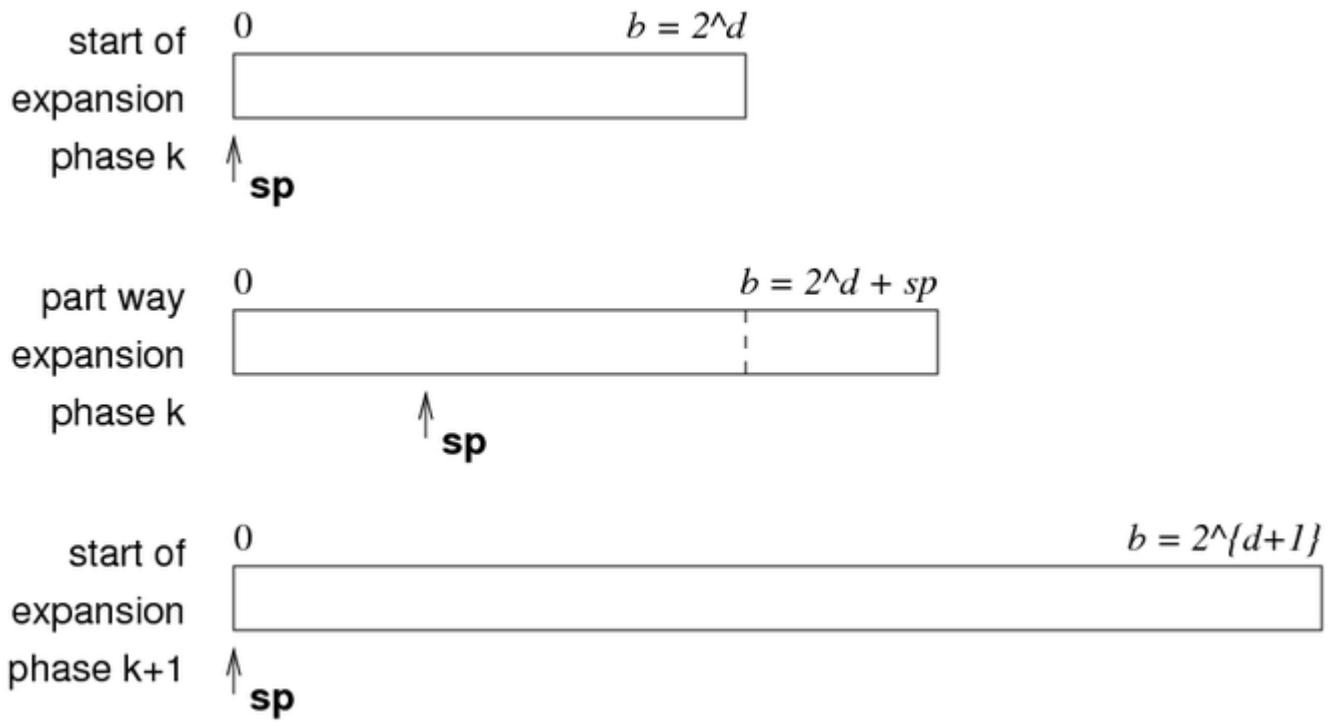
Advantage: does not require auxiliary storage for a directory

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

❖ Linear Hashing (cont)

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.

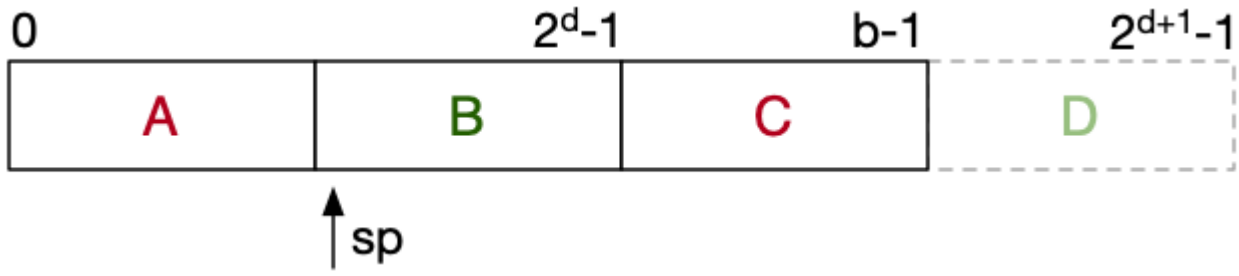
Use *d* bits of hash to compute page address.

```
// 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 (cont)

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.

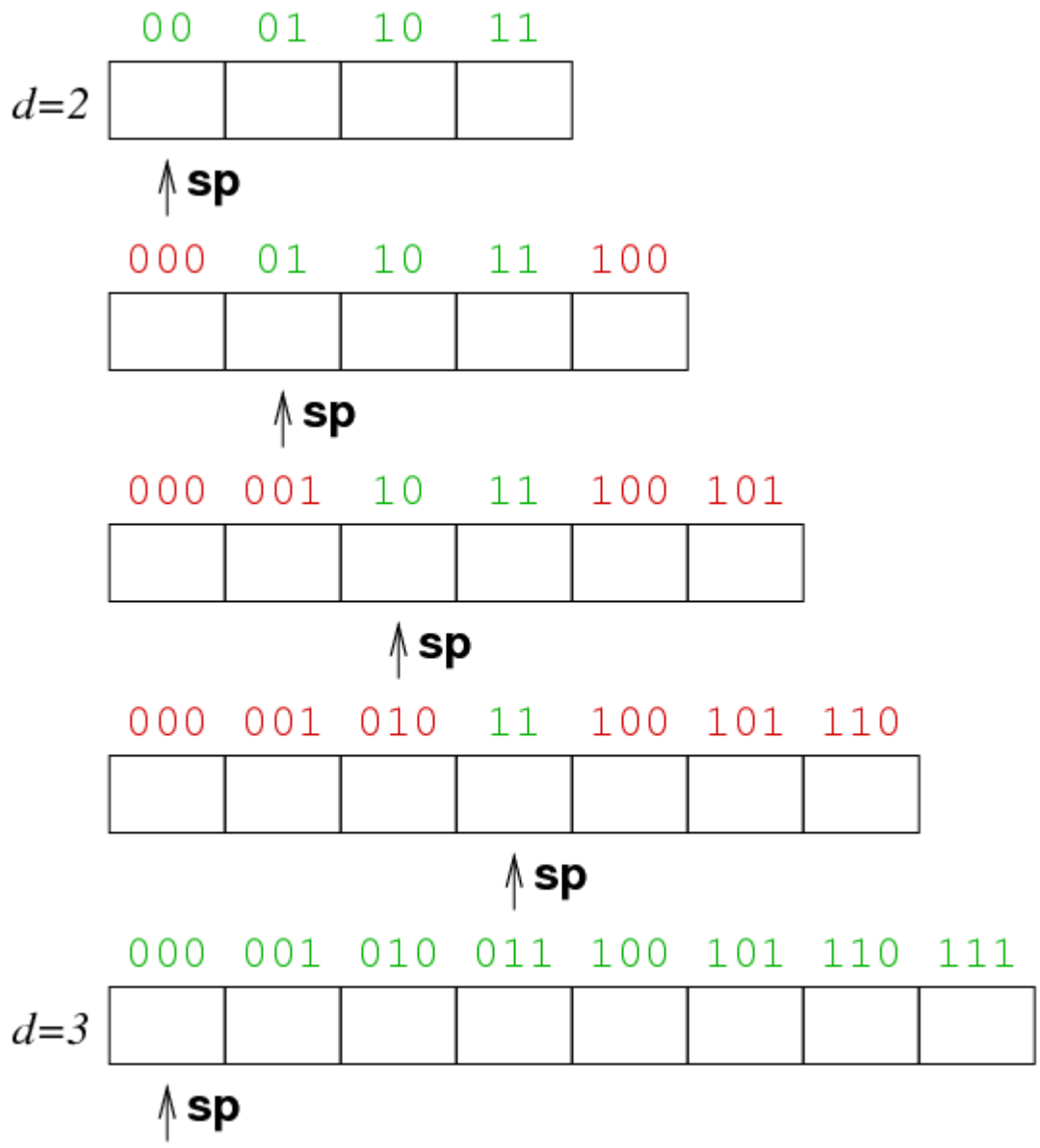
Part *D* does not yet exist (tuples in *B* may eventually move into it).

❖ Selection with Lin.Hashing (cont)

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

❖ File Expansion with Lin.Hashing



❖ Insertion with Lin.Hashing

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

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)

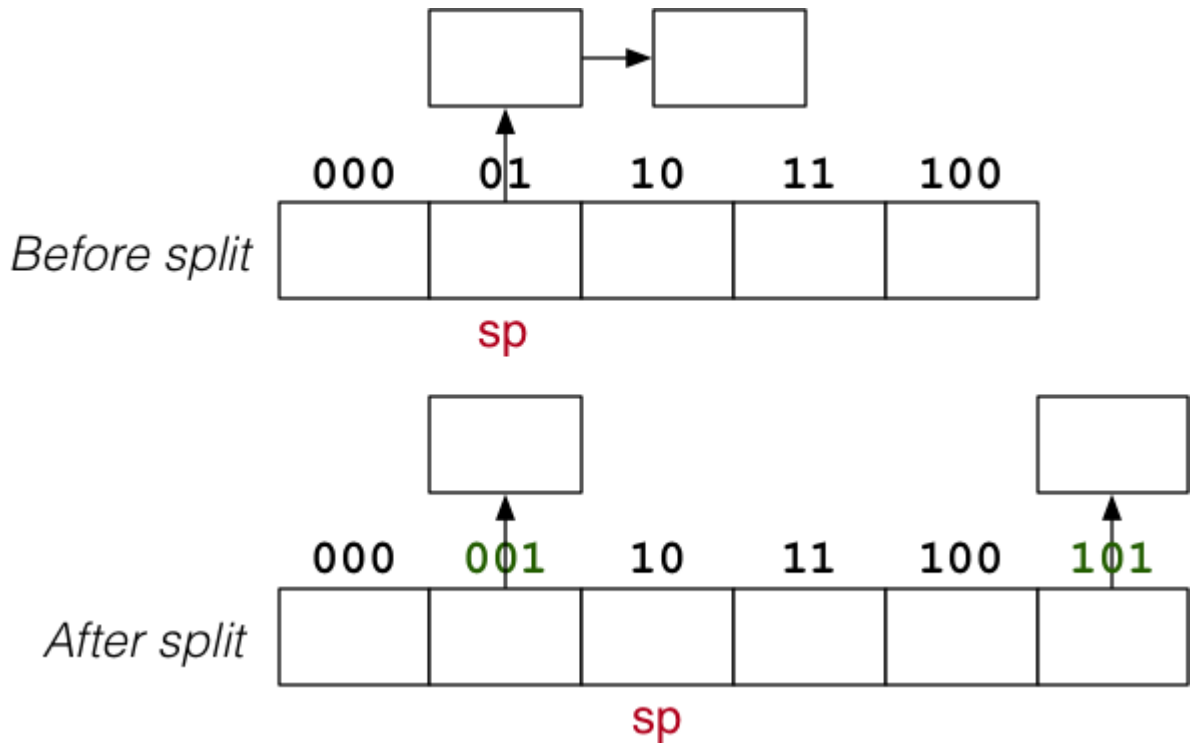
Note: always split page *sp*, even if not full or "current"

Systematic splitting like this ...

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

❖ Splitting (cont)

Splitting process for page *sp*=**01**:



❖ Splitting (cont)

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)

Produced: 23 Feb 2023

Hashing in PostgreSQL

❖ Hashing 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

Detailed info in **src/backend/access/hash/README**

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

❖ PostgreSQL Hash Function

PostgreSQL generic 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()**)

❖ PostgreSQL Hash Function (cont)

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

Typically invoked from a type-specific function, e.g.

```
Datum
hashint4(PG_FUNCTION_ARGS)
{
    return hash_uint32(PG_GETARG_INT32(0));
}
```

where **`hash_uint32()`** is a faster version of **`hash_any()`**

Hash value is "wrapped" as a **`Datum`**

❖ PostgreSQL Hash Function (cont)

Implementation of hash → page ID

```
Bucket
_hash_hashkey2bucket(uint32 hashkey, uint32 maxbucket,
                     uint32 highmask, uint32 lowmask)
{
    Bucket      bucket;

    bucket = hashkey & highmask;
    if (bucket > maxbucket)
        bucket = bucket & lowmask;

    return bucket;
}
```

❖ Hash Files in PostgreSQL

PostgreSQL uses different file organisation ...

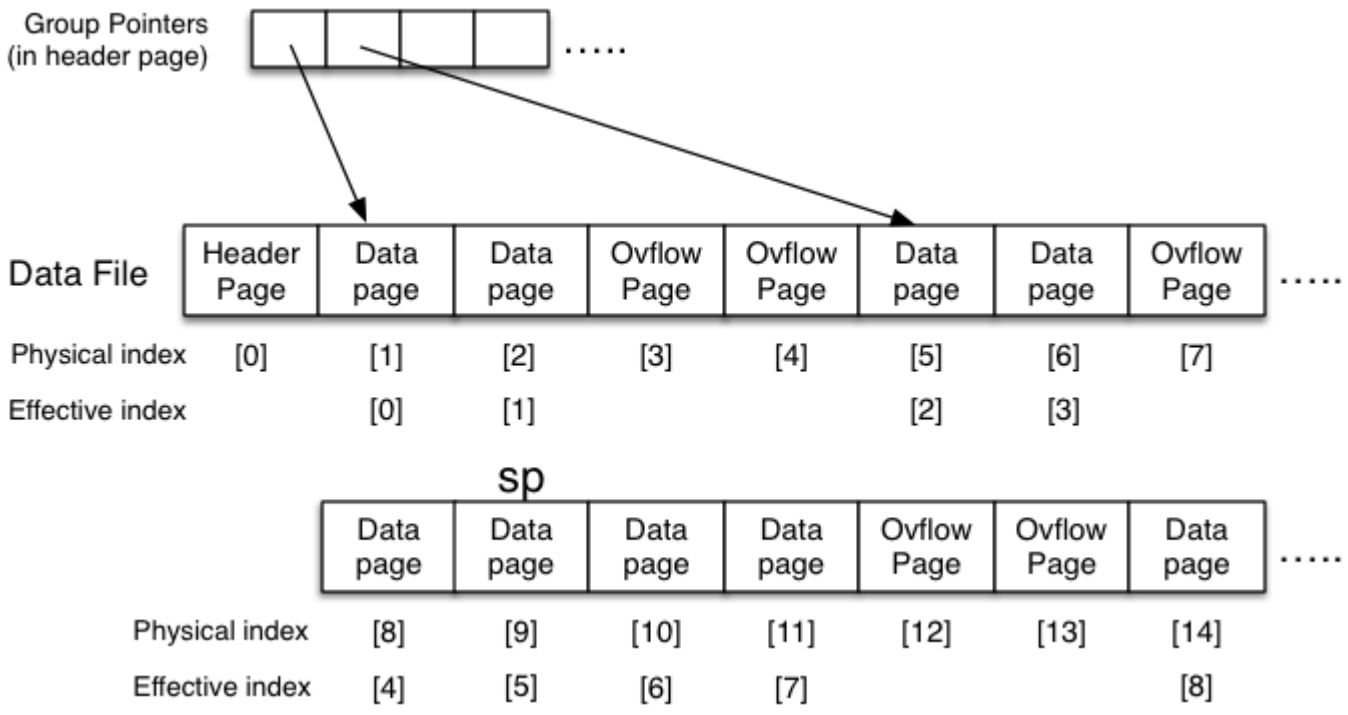
- has a single file containing header, main and overflow pages
- has groups of main pages of size 2^n
- in between groups, arbitrary number of overflow pages
- maintains collection of group pointers in header page
- each group pointer indicates start of main page group

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

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

❖ Hash Files in PostgreSQL (cont)

PostgreSQL hash file structure:



❖ Hash Files in PostgreSQL (cont)

Approximate method for 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: 23 Feb 2023

Week 4 Exercises

❖ Exercise 1: File Merging

Implement a merging algorithm

- for two sorted files, using 3 buffers, with b_1=5, b_2=3
- for one unsorted file, using 3 buffers, with b = 12
- for one unsorted file, using 5 buffers, with b = 27

Assume that we have functions

- **`get_page(rel, pid, buf)`** ... read specified page into buffer
- **`put_page(rel, pid, buf)`** ... write a page to disk, at position pid
- **`clear_page(rel, buf)`** ... make page have zero tuples
- **`sort_page(buf)`** ... in-memory sort of tuples in page
- **`nPages(rel)`**, **`nTuples(buf)`**, **`get_tuple(buf, tid)`**

❖ Exercise 2: Sort-based Projection

Consider a table *R(x,y,z)* with tuples:

```
Page 0:  (1,1,'a')   (11,2,'a')  (3,3,'c')
Page 1:  (13,5,'c')  (2,6,'b')   (9,4,'a')
Page 2:  (6,2,'a')   (17,7,'a')  (7,3,'b')
Page 3:  (14,6,'a')  (8,4,'c')   (5,2,'b')
Page 4:  (10,1,'b')  (15,5,'b')  (12,6,'b')
Page 5:  (4,2,'a')   (16,9,'c')  (18,8,'c')
```

SQL: **`create T as (select distinct y from R)`**

Assuming:

- 3 memory buffers, 2 for input, one for output
- pages/buffers hold 3 **`R`** tuples (i.e. $c_R=3$), 6 **`T`** tuples (i.e. $c_T=6$)

Show how sort-based projection would execute this statement.

❖ Exercise 3: Hash-based Projection

Consider a table *R(x,y,z)* with tuples:

```
Page 0:  (1,1,'a')   (11,2,'a')  (3,3,'c')
Page 1:  (13,5,'c')  (2,6,'b')   (9,4,'a')
Page 2:  (6,2,'a')   (17,7,'a')  (7,3,'b')
Page 3:  (14,6,'a')  (8,4,'c')   (5,2,'b')
Page 4:  (10,1,'b')  (15,5,'b')  (12,6,'b')
Page 5:  (4,2,'a')   (16,9,'c')  (18,8,'c')
-- and then the same tuples repeated for pages 6-11
```

SQL: **`create T as (select distinct y from R)`**

Assuming:

- 4 memory buffers, one for input, 3 for partitioning
- pages/buffers hold 3 **`R`** tuples (i.e. $c_R=3$), 4 **`T`** tuples (i.e. $c_T=4$)
- hash functions: h1(x) = x%3, h2(x) = (x%4)%3

Show how hash-based projection would execute this statement.

❖ Exercise 4: Query Types

Using the relation:

```
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 ...

❖ Exercise 5: 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
```

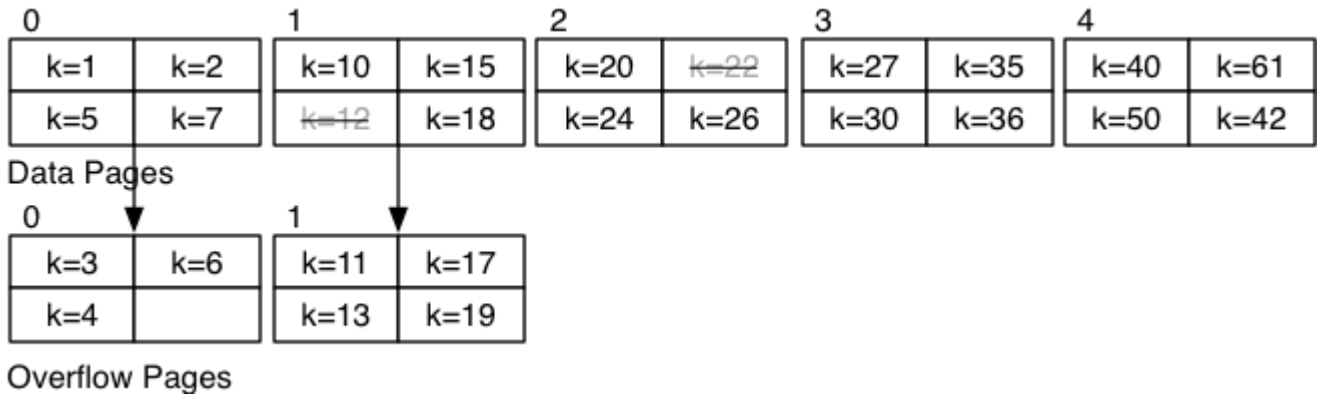
Show how each will be executed and estimate the cost, assuming:

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

State any other assumptions.

❖ Exercise 6.1: 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 6.2: Optimising Sorted-file Search

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.

❖ Exercise 7: 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.

❖ Hash Values and Bit-strings

Hashing requires *h(k)* :: KeyVal → HashVal

HashVal is typically a 32-bit integer, which is mapped to 0 .. *b-1*

For arbitrary *b*, mapping done via **`PageID = h(k) % b`**

If $b = 2^d$, mapping can be done via bitwise AND

E.g. *b == 8*, **`PageID = h(k) & 0b0111`**

For any *d*, use a mask with lower-order *d* bits set to 1

❖ Exercise 8: Bit Manipulation

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

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

(assumes supplied buffer is large enough, like **`gets()`**)

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

❖ Exercise 9: Insertion into Linear Hashed File

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

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.

Split *before* every sixth insert.

Produced: 8 Mar 2023