>>

Week 4 Exercises

^ >>

❖ Things to Note

Quiz 2 ... due Friday 11 March at 9pm

Assignment 1 .... due Friday 18 March at 9pm

"Disk quota exceeded"? Check your disk usage on /localstorage/

- use the **`du`** command to find "hot spots"
- if log is big: stop server, remove log file, restart server

This weekend is Census Weekend ...

- if you want to drop a course and not pay for it, drop it now
- if you want to drop COMP9315, drop it in MyUNSW and email me

<< ^ >>

❖ Assignment 1

Debugging the assignment ...

- can't easily use GDB or vscode; need debugging print's
- can use **`ereport()`** and **`elog()`**; see PG Docs 54.2

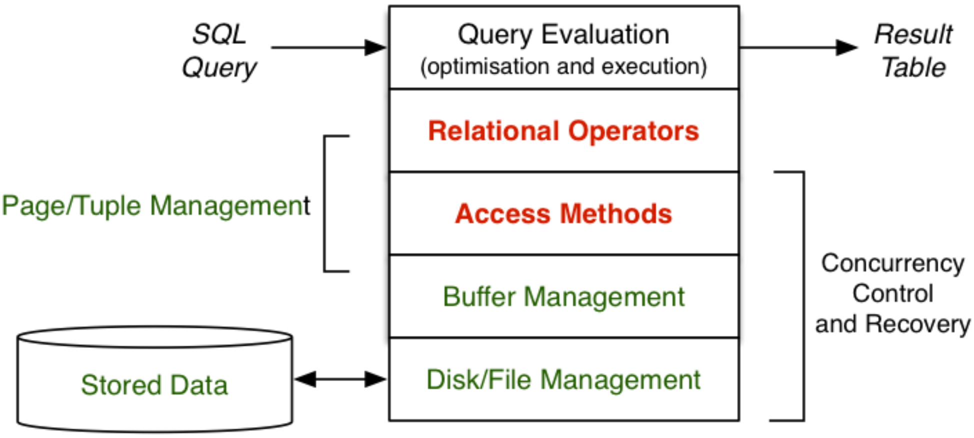
Implementing **`PersonName`** ...

- three data usages: readable, computable, storeable
  - readable → storeable in **`pname_in()`**
  - storeable → readable in **`pname_out()`**
  - storeable → computable in e.g. **`pname_cmp()`**
- reminder: pointers to **`malloc`**'d memory structures are not storeable

Rebuilding PostgreSQL will *not* solve problems in **`pname.*`**

❖ DBMS Architecture (revisited)

Implementation of relational operations in DBMS:



<< ^ >>

❖ Exercise: 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)`**

<< ^ >>

❖ Projection

Projection removes some attributes from tuples

Projection can produce duplicates

R(x,y,z) = [(a,b,6), (a,c,7), (a,b,8)]

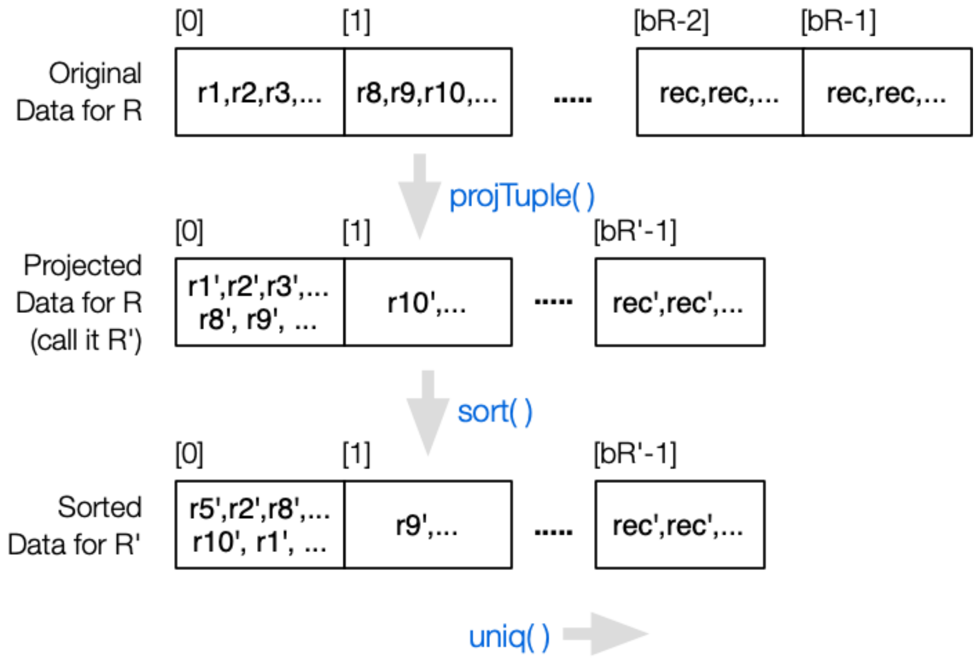
Proj[x,y] R ⇒ [(a,b), (a,c), (a,b)]

If duplicates are ok, projection is trivial

```
for each tuple t {
   t' = t projected on some attributes
   output t'
}
```

<< ∧ >>

❖ Sort-based Projection



<< ^ >>

❖ Sort-based Projection (cont)

Duplicate removal in a sorted file

```
curr = NULL
for each tuple t {
   if (t == curr) continue
   output t
   curr = t
}
```

Requires scan of all b_{in} pages in input; writes b_{out} pages; $b_{out} \leq b_{in}$

COMP9315 21T1 ♢ Week 4 Exercises ♢ [7/29]

<< ^ >>

❖ Exercise: 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)`**

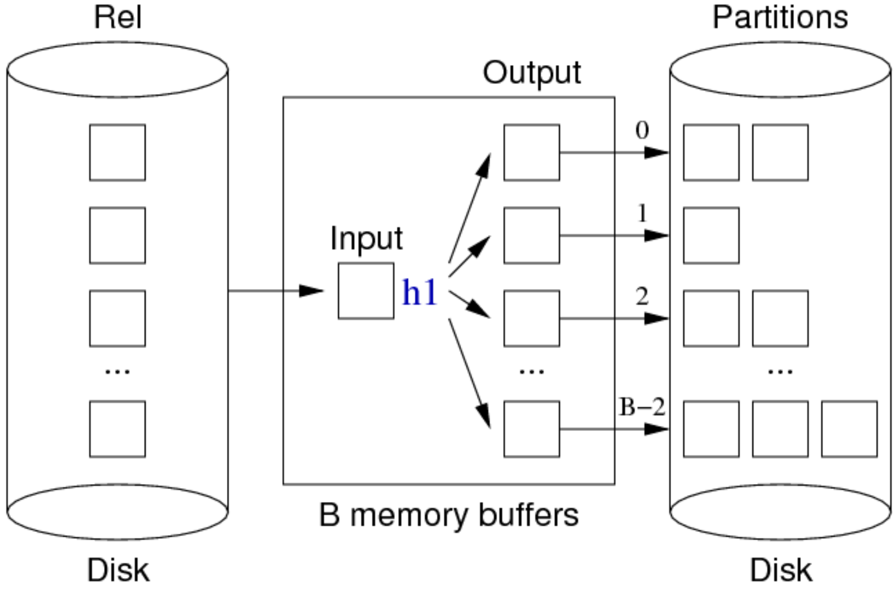
Assuming:

- 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.

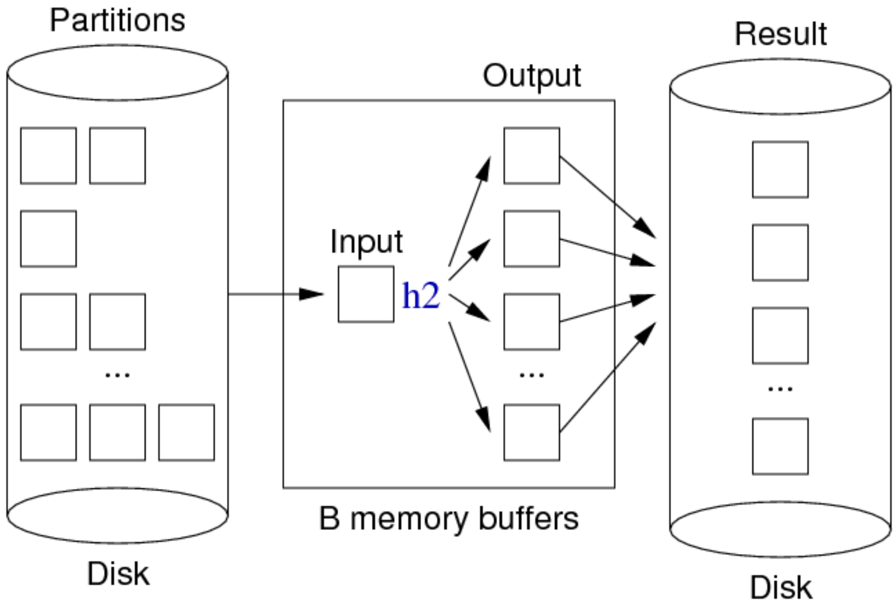
❖ Hash-based Projection

Partitioning phase:



❖ Hash-based Projection (cont)

Duplicate elimination phase:



<< ^ >>

❖ Hash-based Projection (cont)

Pseudo-code for hash-based projection (with N buffers)

```
open N-1 partition files (f[0], f[1], ... f[N-2])
clear all buffers (b[0], b[1], ... b[N-2])
for each tuple t {   -- via input buffer
   h = h1(t)
   add t to b[h]
   if (b[h] is full) { write to f[h]; clear b[h] }
}
rewind all files f[i]
for each partition p in 0 .. N-1 {
   for each tuple t in f[p] {   -- via input buffer
      h = h2(t)
      if (t in b[h]) continue
      add t to b[h]
   }
   for each buffer b[i] write b[i] to output file
}
```

<< ∧ >>

❖ Exercise: 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.

❖ Query Types

Type	SQL	RelAlg	a.k.a.
Scan	`select * from R`	*R*	-
Proj	`select` *x,y* `from R`	*Proj[x,y]R*	-
Sort	`select * from R order by` *x*	*Sort[x]R*	*ord*
Sel_1	`select * from R where id =` *k*	*Sel[id=k]R*	*one*
Sel_n	`select * from R where a =` *k*	*Sel[a=k]R*	-
$Join_1$	`select * from R,S where R.id = S.r`	*R Join[id=r] S*	-

Different query classes exhibit different query processing behaviours.

<< ^ >>

❖ Exercise: 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:

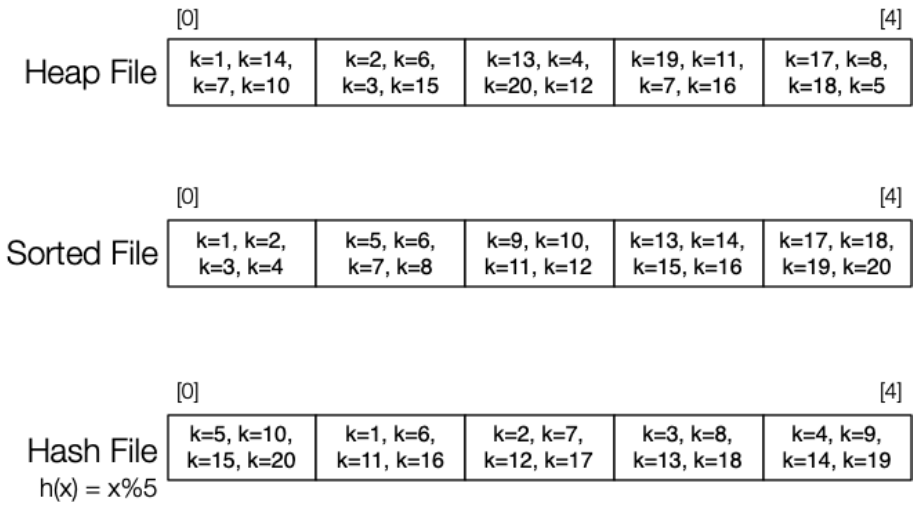
1. a 1-d *one* query,   an n-d *one* query
2. a 1-d *pmr* query,   an n-d *pmr* query
3. a 1-d *range* query,   an n-d *range* query

Suggest how many solutions each might produce ...

<< ^ >>

❖ Data File Structures

Three common file types (*b = 5, c = 4*)



<< ^ >>

❖ Exercise: 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:

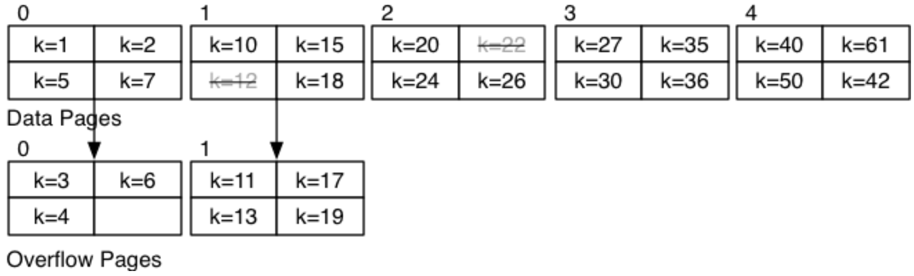
- $b = 100$, $b_{q2} = 3$, $b_{q3} = 20$

State any other assumptions.

<< ^ >>

❖ Exercise: 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: 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: 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.

<< ^ >>

❖ Things to Note

Quiz 2 ... due Friday 11 March (tomorrow!) at 9pm

Assignment 1 ... due Friday 18 March at 9pm

- need a hash function? write your own, or use a PostgreSQL built-in one

Assignment 2 ... implement a multi-attribute linear-hashed file

This weekend is Census Weekend ...

- if you want to drop a course and not pay for it, drop it now
- if you want to drop COMP9315, drop it in MyUNSW and email me

<< ^ >>

❖ 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: 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* its of a **`uint32`**

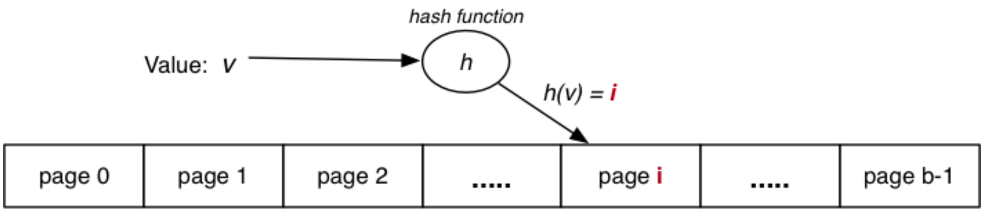
   ```
   uint32 bits(int d, uint32 val);
   ```

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

<< ∧ >>

❖ 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 *KeyVal → [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 Performance

Aims:

- distribute tuples evenly amongst buckets (data + overflow pages)
- have most data pages nearly full (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)

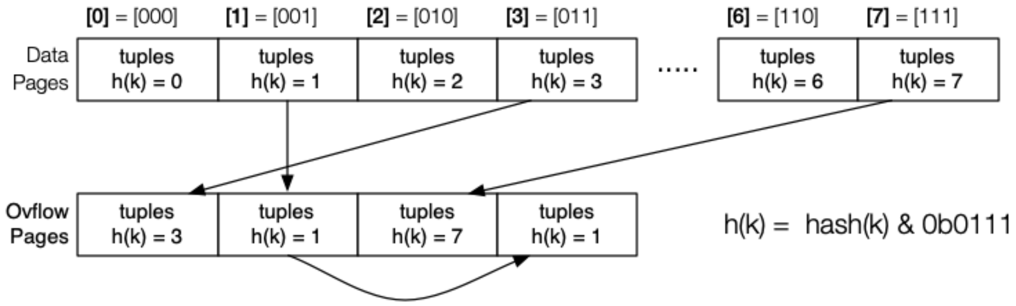
All tuples in each bucket have same hash value.

<< ^ >>

❖ Hashed Files

Hash function maps key → 0 .. *b-1*

E.g. if $b = 2^d$, simply use lower-order *d* bits of hash



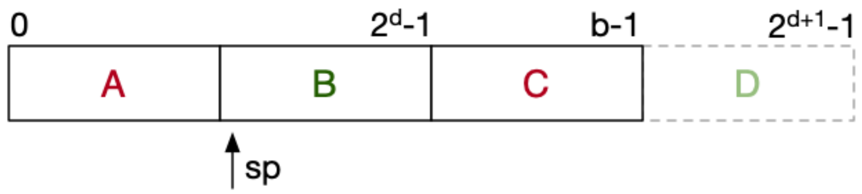
Static hashing: fixed-size file ⇒ overflow chains grow without bound

Linear hashing: variable-size file, using sp = split pointer, d = depth

❖ Linear-hashed Files

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

If $b \mathrel{!}= 2^d$, 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)

❖ Linear-hashed Files (cont)

Mapping a hash value to a bucket id for searching

```
// select * from R where k = val
h = hash(val);
pid = bits(d,h);  // take lower-order d bits from h
if (pid < sp) { pid = bits(d+1,h); }

P = getPage(datafile(reln), pid)
for each tuple t in page P
    if (t.k == val) add t to answerSet

pid = ovflow(P)
while (pid != NULL) {
    P = getPage(ovfile(reln), pid)
    for each tuple t in page P
        if (t.k == val) add t to answerSet
    pid = ovflow(P)
}
```

<< ^ >>

❖ Linear-hashed Files (cont)

Periodically, extend file by one page

- new page has address $sp + 2^d$
- tuples in bucket *sp* are redistributed
  - by considering $d+1$ bits of hash (e.g. **`0111`** → **`?0111`**)
  - tuples where extra bit is 0, stay in bucket *sp*
  - tuples where extra bit is 1, move to bucket $sp + 2^d$
- after this, *sp++*; if ($sp == 2^d$) { *sp = 0; d++* }

This process is called splitting

It can reduce, but may not remove, overflow chain for bucket *sp*

<< ^

❖ Exercise: 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: 10 Mar 2022