>>

COMP9315 Week 2 Sessions

^ >>

❖ Things to Note

Assignment 1

- worth 15%, due Friday 20 March, 5% / day late penalty

Quiz 1

- due before Friday 23:59 ... so far 30/410 submissions

Dropping/Enrolling

- if you drop COMP9315 now, will need help enrolling in replacement

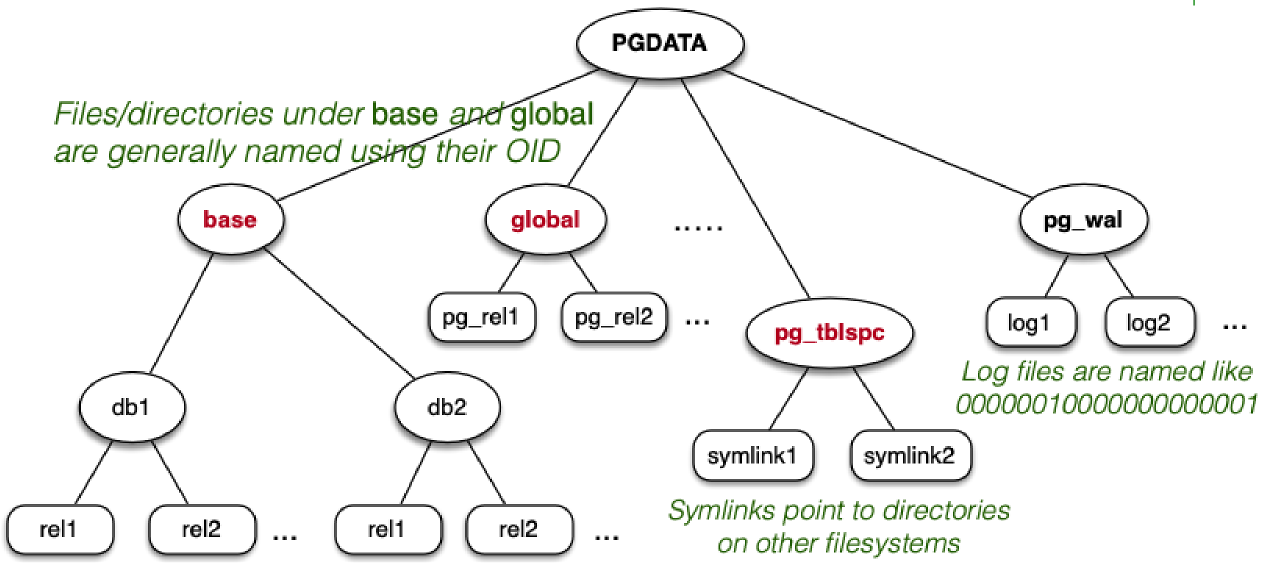
Unix skills

- Home Computing playlist on https://www.youtube.com/channel/UCi3Kf5eONIwV6QgNHiYqVzg

<< ^ >>

❖ PostgreSQL File Manager

PostgreSQL uses the following file organisation ...



Very large tables are split into multiple files (forks)

<< ^ >>

❖ Exercise: PostgreSQL Files

In your PostgreSQL server

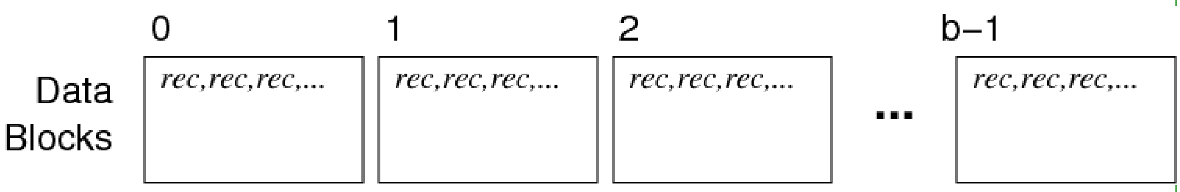
- examine the content of the **`$PGDATA/base`** directory
- find the directory containing the **`uni`** database (from P03)
- find the file in this directory for the **`People`** table
- examine the contents of the **`People`** file
- what are the other files in the directory?
- are there forks in any of your database?

<< ^ >>

❖ DBMS Files

Our view of relations/tables in DBMSs:

- a relation is a set of r tuples, with average size R bytes
- the tuples are stored in b data pages on disk
- each page has size B bytes and contains up to c tuples
- data is transferred disk↔memory in whole pages
- cost of disk↔memory transfer T_r, T_w dominates other costs



<< ∧ >>

❖ File Scan

Scanning a relation file involves a process something like ...

```
buf = malloc(B);
in = open("DB_file_name",O_RDONLY);
while (read(in, buf, B) == B) {
   for each tuple T in B {
      process T
   }
}
```

We look at how to extract tuples from a buffer later

❖ Exercise: Relation Scan Cost

Consider a table *R(x,y,z)* with 10^5 tuples, implemented as

- number of tuples *r = 10,000*
- average size of tuples *R = 200* bytes
- size of data pages *B = 4096* bytes
- time to read one data page $T_r = 10msec$
- time to check one tuple *1 usec*
- time to form one result tuple *1 usec*
- time to write one result page $T_w = 10msec$

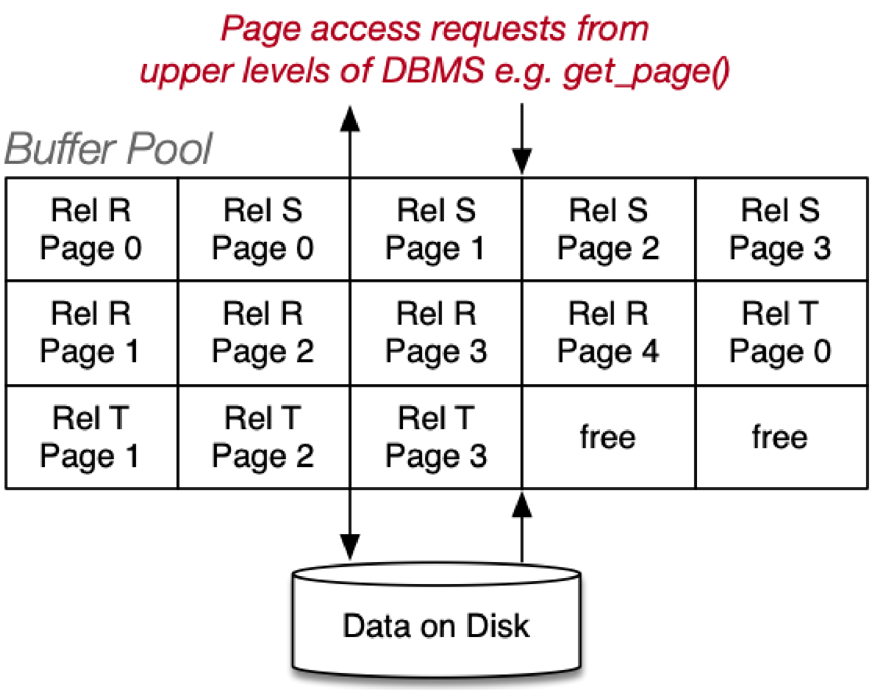
Calculate the total time-cost for answering the query:

```
insert into S select * from R where x > 10;
```

if 50% of the tuples satisfy the condition.

<< ∧ >>

❖ Buffer Pool


Page access requests from
upper levels of DBMS e.g. get_page()
Buffer Pool
Rel R Page 0
Rel S Page 0
Rel S Page 1
Rel S Page 2
Rel S Page 3
Rel R Page 1
Rel R Page 2
Rel R Page 3
Rel R Page 4
Rel T Page 0
Rel T Page 1
Rel T Page 2
Rel T Page 3
free
free
Data on Disk

❖ Buffer Pool (cont)

Higher levels of DBMS no longer use **`read`**

```
buf = get_page_via_pool(Reln, PageID)
```

Generally, buffer pool is full when we request a page

- if page in pool, use it
- if page not in pool, eject a page and use its slot

Strategies for ejecting a page

- prefer to eject non-modified page
- MRU = eject most recently used page
- LRU = eject least recently used page

<< ^ >>

❖ Exercise: Buffer Cost Benefit (i)

Consider two relations being joined

```
select * from Customer join Employee
```

To evaluate *C join E* (in pseudo Python notation)

```
for each page P_C of C:
   for each page P_E of E:
      for each tuple T_C in P_C:
         for each tuple T_E in P_E:
            if (T_C matches T_E):
               add T_C.T_E to P_out
               if (P_out full):
                  write and clear P_out
```

<< ∧ >>

❖ Exercise: Buffer Cost Benefit (i) (cont)

Assume that:

- the **`Customer`** relation has b_C pages (e.g. 10)
- the **`Employee`** relation has b_E pages (e.g. 4)

Compute how many page reads occur for *C join E* ...

- if we have only 3 buffers (i.e. effectively no buffer pool)
- if we have 20 buffers
- when a buffer pool with MRU replacement strategy is used
- when a buffer pool with LRU replacement strategy is used

For the last two, buffer pool has $n=3$ slots ($n < b_C$ and $n < b_E$) + output buffer

<< ^ >>

❖ To Note

Quiz 1 ... due Friday (tomorrow) before midnight

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

- create a new base type (**`PersonName`**)
- define operations on that type (read,write,compare,...)
- link it into the PostgreSQL server

<< ^ >>

❖ Assignment 1

Creating a new base type requires

- telling the SQL front-end about it
- building C functions to manipulate values of the type
- setting up ordering to allow indexing

At the SQL level (**`pname.source`**)...

```
create type PersonName ( type info and function links )
```

Also useful to define comparison operators on the type (e.g. < > =)

<< ^ >>

❖ Assignment 1 (cont)

Once created, the type can be used in client SQL programs ...

```
e.g. in schemas ...
create table People ( id serial, name PersonName, ...);

e.g. inserting data ...
insert into People values (default, 'Smith, John', ...);

e.g. retrieving ...
select * from People
where  family(name) = 'Smith';

select given(name) from People
where  family(name) = 'Wang';

select * from People
where  name > 'Solomon, David';
```

<< ^ >>

❖ Assignment 1 (cont)

At the C level (**`pname.c`**) ...

```
PG_FUNCTION_INFO_V1(pname_in);

Datum
pname_in(PG_FUNCTION_ARGS)
{
   // parse input string
   // convert to internal representation
   // return value as Datum
}
```

Link between C and SQL (**`pname.source`**) ...

```
CREATE FUNCTION family(PersonName)
   RETURNS text
   AS '_OBJWD_/pname'
   LANGUAGE C IMMUTABLE STRICT;
```

<< ^ >>

❖ Assignment 1 (cont)

Required functions for type *T*

- ***T*_in()** ... invoked when PG receives value of type *T*
- ***T*_out()** ... convert value of type *T* to printable

Useful options for new type

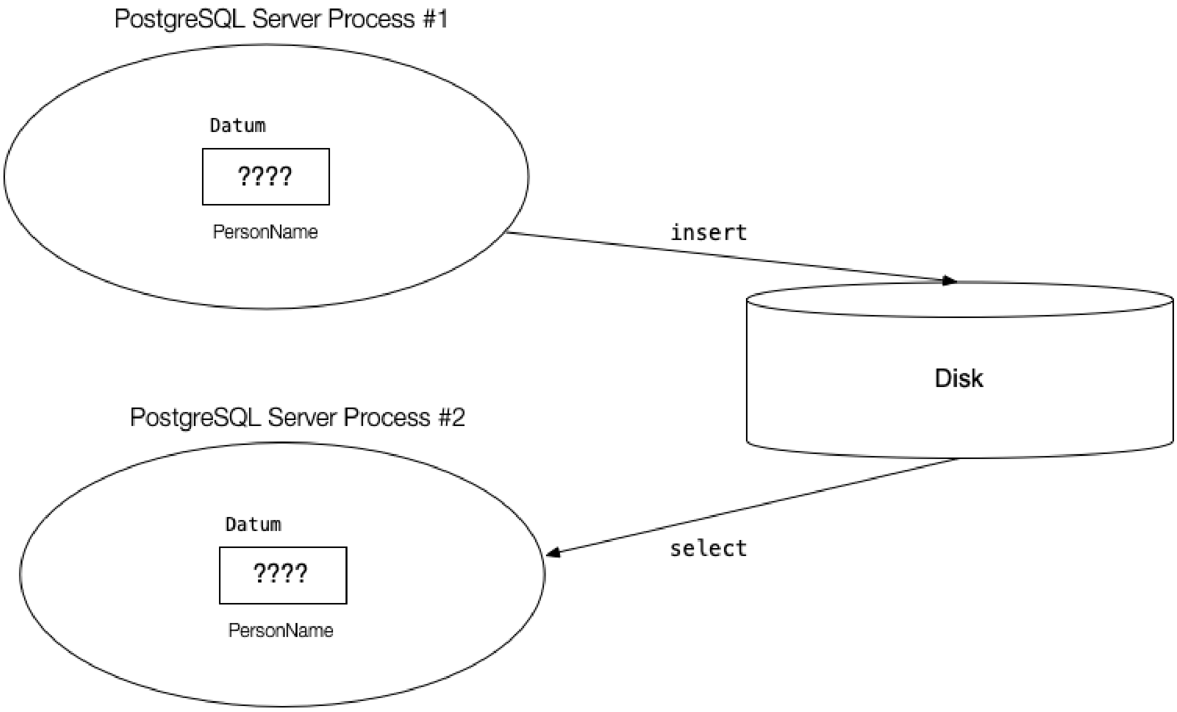
- **INTERNALLENGTH =** ***nbytes*|VARIABLE**
- **ALIGNMENT = char|int2|int4|double**

See PG Manual **CREATE TYPE** under SQL Commands for more details.

<< ^ >>

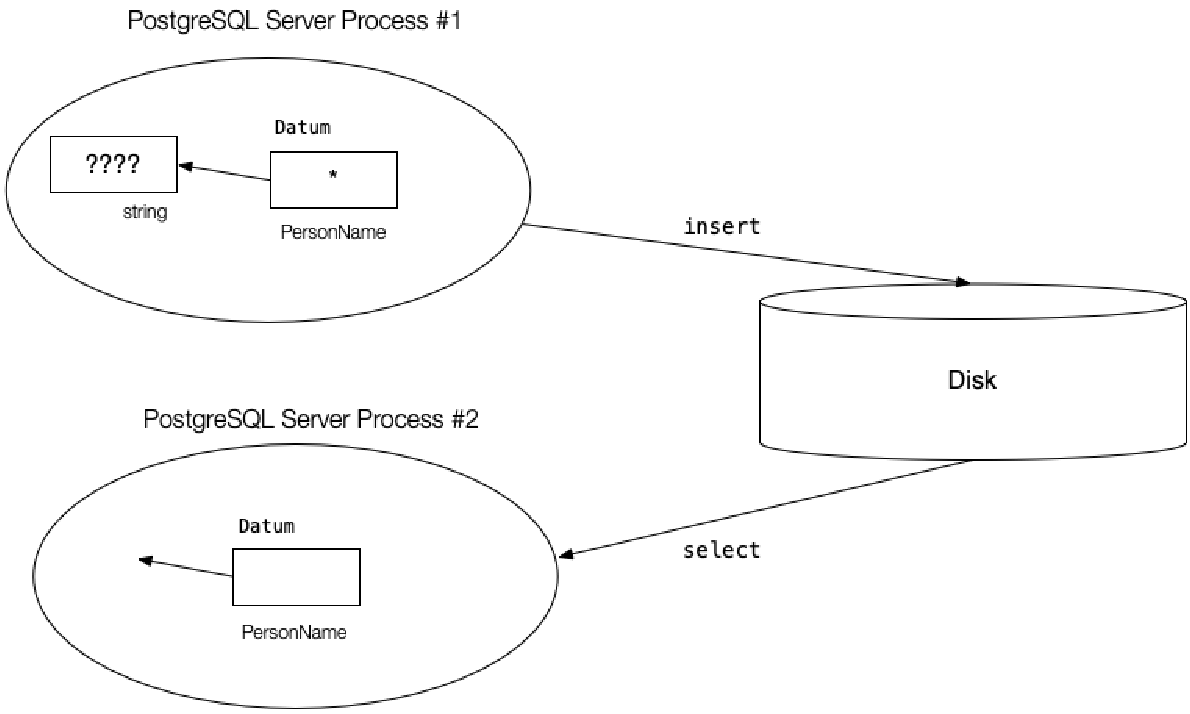
❖ Assignment 1 (cont)

Data transfers that *should* happen ...



❖ Assignment 1 (cont)

Data transfers that students sometimes implement ...



❖ Assignment 1 (cont)

Copy **`complex.c`** and **`complex.source`** ... BUT

- **`Complex`** is a fixed-length type (two **`int`**s)
- do not make **`PersonName`** fixed-length (how long?)
- change all references to **`Complex`** to **`PersonName`**
- make sure **`_OBJWD_`** references the correct directory

This assignment requires some prior reading (code + doco)

- do not leave it to the last minute

<< ^ >>

❖ Exercise: Buffer Cost Benefit (ii)

If the tables were larger, the above analysis would be tedious.

Write a C program to simulate buffer pool usage

- assuming a nested loop join as above
- `argv[1]` gives number of pages in "outer" table
- `argv[2]` gives number of pages in "inner" table
- `argv[3]` gives number of slots in buffer pool
- `argv[4]` gives replacement strategy (LRU,MRU,FIFO-Q)

<< ∧ >>

❖ Buffer Replacement Strategies

Typically implemented using list of free buffers (pin count = 0)

Order of list determined by LRU/MRU strategy

When buffer on list is accessed, removed from list

<< ^ >>

❖ Clock-sweep Replacement Strategy

PostgreSQL page replacement strategy: clock-sweep

- treat buffer pool as circular list of buffer slots
- **`NextVictimBuffer`** (NVB) holds index of next possible evictee
- if **`Buf[NVB]`** page is pinned or "popular", leave it
    - **`usage_count`** implements "popularity/recency" measure
    - incremented on each access to buffer (up to small limit)
    - decremented each time considered for eviction
- else if **`pin_count`** = 0 and **`usage_count`** = 0 then grab this buffer
- increment **`NextVictimBuffer`** and try again (wrap at end)

<< ^ >>

❖ Exercise: Clock-sweep Page Replacement

Using the following data type for buffer frame descriptors:

```
struct FrameDesc {
    Tag pid;      // ID of page in frame e.g. "R0"
    int pin;      // number tx's using this page
    int usage;    // clock-sweep usage counter
}
```

Show how the buffer pool changes for

- $n = 4$, $b_R = 3$, $b_S = 4$, $b_T = 6$
- when executing **`select * from T`** via sequential scan
- when executing **`select * from R join S`** using nested-loop join

<< ^ >>

❖ Tuples and Records

Tuple = collection of attribute values based on a schema, e.g.

(33357462,	'Neil Young',	'Musician',	277)
iid:integer	name:varchar(20)	job:varchar(10)	dept: smallint

Record = sequence of bytes, containing data for one tuple, e.g.

01101001	11001100	01010101	00111100	10100011	01011111	01011010	

Bytes need to be interpreted relative to schema to get tuple

<< ^ >>

❖ Exercise: Fixed-length Records

Give examples of table definitions

- which result in fixed-length records
- which result in variable-length records

```
create table R ( ... );
```

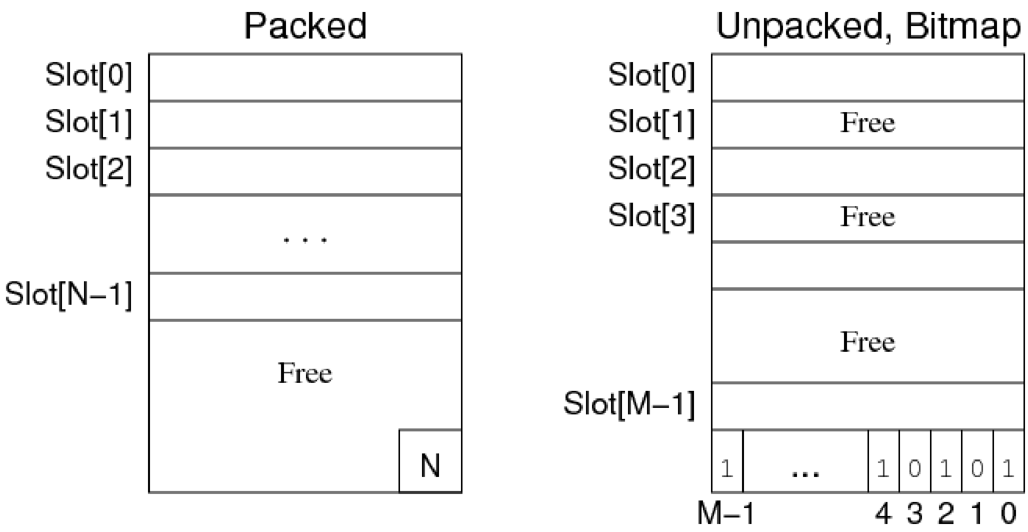
What are the common features of each type of table?

<< ^ >>

❖ Tuples in Pages

For fixed-length records, use record slots.

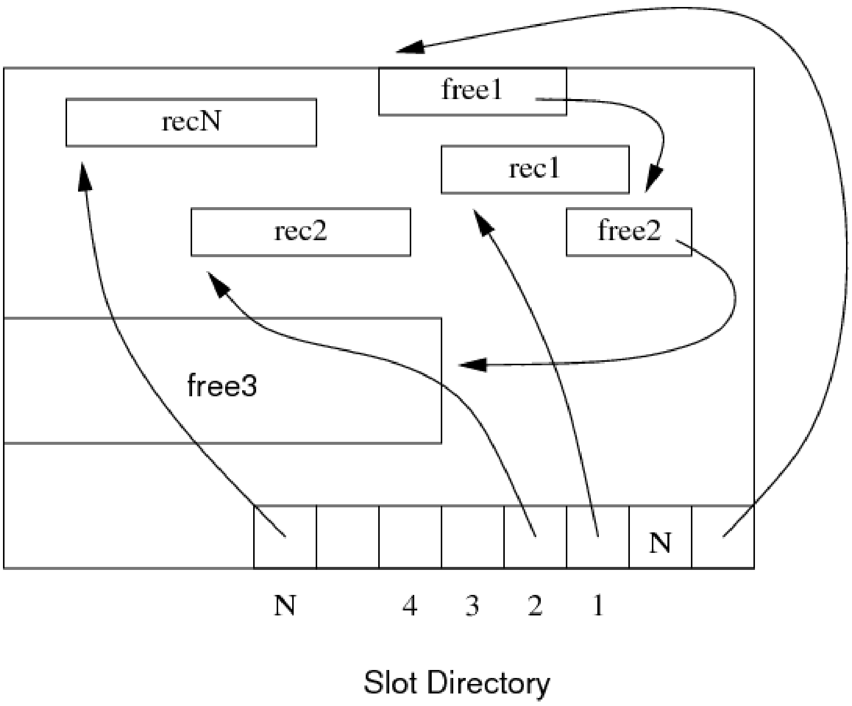
- insert: place new record in first available slot
- delete: two possibilities for handling free record slots:



<< ^ >>

❖ Page Formats

For variable-length records, must use slot directory.



❖ Exercise: Inserting/Deleting Fixed-length Records

For each of the following Page formats:

- compacted/packed free space
- unpacked free space (with bitmap)

Implement

- a suitable data structure to represent a **`Page`**
- a function to insert a new record
- a function to delete a record

<< ∧ >>

❖ Exercise: Inserting Variable-length Records

For both of the following page formats

1. variable-length records, with compacted free space
2. variable-length records, with fragmented free space

implement the `insert()` function.

Use the above page format, but also assume:

- page size is 1024 bytes
- tuples start on 4-byte boundaries
- references into page are all 8-bits (1 byte) long
- a function `recSize(r)` gives size in bytes

❖ Exercise: PostgreSQL Pages

Draw diagrams of a PostgreSQL heap page

- when it is initially empty
- after three tuples have been inserted
  with lengths of 60, 80, and 70 bytes
- after the 80 byte tuple is deleted (but before vacuuming)
- after a new 50 byte tuple is added

Show the values in the tuple header.

Assume that there is no special space in the page.

<< ∧

❖ Exercise: Space Utilisation

Consider the following page/record information:

- page size = 1KB = 1024 bytes = 2^{10} bytes
- records:
  `(a:int,b:varchar(20),c:char(10),d:int)`
- records are all aligned on 4-byte boundaries
- **`c`** field padded to ensure **`d`** starts on 4-byte boundary
- each record has 4 field-offsets at start of record (each 1 byte)
- **`char(10)`** field rounded up to 12-bytes to preserve alignment
- maximum size of **`b`** values = 20 bytes; average size = 16 bytes
- page has 32-bytes of header information, starting at byte 0
- only insertions, no deletions or updates

Calculate c = average number of records per page.

Produced: 24 Feb 2022