Week 02 Lectures

Catalogs

Catalogs are tables describing database objects, e.g.

`pg_class` holds core information about tables

- `relname, relnamespace, reltype, relowner, ...`
- `relkind, relnatts, relhaspkey, relacl[], ...`

`pg_attribute` contains information about attributes

- `attrelid, attname, atttypid, attnum, ...`

`pg_type` contains information about types

- `typname, typnamespace, typowner, typlen, ...`
- `typtype, typrelid, typinput, typoutput, ...`

PostgreSQL Catalog

You can explore the PostgreSQl catalog via `psql` commands

- **`\d`** gives a list of all tables and views
- **`\d`** ***`Table`*** gives a schema for *`Table`*
- **`\df`** gives a list of user-defined functions
- **`\df+`** ***`Function`*** gives details of *`Function`*
- **`\ef`** ***`Function`*** allows you to edit *`Function`*
- **`\dv`** gives a list of user-defined views
- **`\d+`** ***`View`*** gives definition of *`View`*

You can also explore via SQL on the catalog tables

Exercise 1: Table Statistics

Using the PostgreSQL catalog, write a PLpgSQL function

- to return table name and #tuples in table
- for all tables in the `public` schema

```
create type TableInfo as (table text, ntuples int);
create function pop() returns setof TableInfo ...
```

Hints:

- `table` is a reserved word
- you will need to use dynamically-generated queries.

Exercise 2: Extracting a Schema

Write a PLpgSQL function:

- `function schema() returns setof text`

- giving a list of table schemas in the `public` schema

It should behave as follows:

```
db=# select * from schema();
            tables
---------------------------
 table1(x, y, z)
 table2(a, b)
 table3(id, name, address)
...
```

Exercise 3: Enumerated Types

PostgreSQL allows you to define enumerated types, e.g.

```
create type Mood as enum ('sad', 'happy');
```

Creates a type with two ordered values `'sad' < 'happy'`

What is created in the catalog for the above definition?

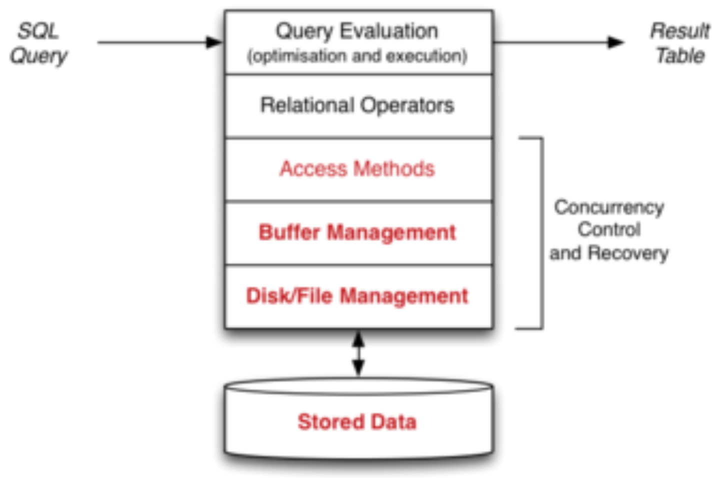
Hint:

```
pg_type(oid, typname, typelen, typetype, ...)
pg_enum(oid, enumtypid, enumlabel)
```

Storage Manager

DBMS Storage Manager

Levels of DBMS related to storage management:



Storage Technology

Persistent storage is

- large, cheap, relatively slow, accessed in blocks
- used for long-term storage of data

Computational storage is

- small, expensive, fast, accessed by byte/word
- used for all analysis of data

Access cost HDD:RAM ≅ 100000:1, e.g.

- 100ms to read block containing two tuples
- 1μs to compare fields in two tuples

Cost Models

Throughout this course, we compare costs of DB operations

Important aspects in determining cost:

- data is always transferred to/from disk as whole blocks (pages)
- cost of manipulating tuples in memory is negligible
- overall cost determined primarily by #data-blocks read/written

Complicating factors in determining costs:

- not all page accesses require disk access (buffer pool)
- tuples typically have variable size (tuples/page ?)

More details later ...

File Management

Aims of file management subsystem:

- organise layout of data within the filesystem
- handle mapping from database ID to file address
- transfer blocks of data between buffer pool and filesystem
- also attempts to handle file access error problems (retry)

Builds higher-level operations on top of OS file operations.

... File Management

Typical file operations provided by the operating system:

```
fd = open(fileName,mode)
  // open a named file for reading/writing/appending
close(fd)
  // close an open file, via its descriptor
nread = read(fd, buf, nbytes)
  // attempt to read data from file into buffer
nwritten = write(fd, buf, nbytes)
  // attempt to write data from buffer to file
lseek(fd, offset, seek_type)
  // move file pointer to relative/absolute file offset
fsync(fd)
  // flush contents of file buffers to disk
```

DBMS File Organisation

How is data for DB objects arranged in the file system?

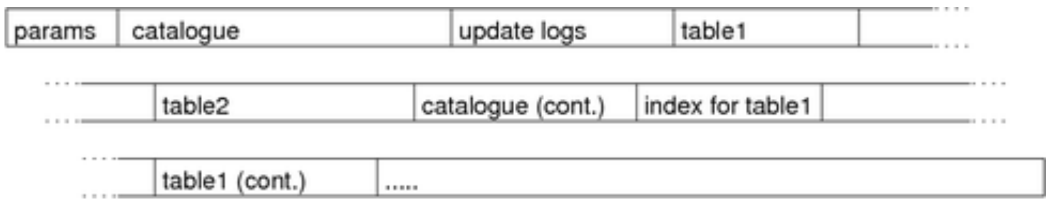
Different DBMSs make different choices, e.g.

- by-pass the file system and use a raw disk partition
- have a single very large file containing all DB data
- have several large files, with tables spread across them
- have multiple data files, one for each table
- have multiple files for each table
- etc.

Single-file DBMS

Consider a single file for the entire database (e.g. SQLite)

Objects are allocated to regions (segments) of the file.



If an object grows too large for allocated segment, allocate an extension.

What happens to allocated space when objects are removed?

... Single-file DBMS

Allocating space in Unix files is easy:

- simply seek to the place you want and write the data
- if nothing there already, data is appended to the file
- if something there already, it gets overwritten

If the seek goes way beyond the end of the file:

- Unix does not (yet) allocate disk space for the "hole"
- allocates disk storage only when data is written there

With the above, a disk/file manager is easy to implement.

Single-file Disk Manager

Simple disk manager for a single-file database:

```
// Disk Manager data/functions
#define PAGESIZE 2048   // bytes per page
typedef int PageID;     // PageID is block index

typedef struct DBdescriptor {
   char *dbname;     // copy of database name
   int   fd;         // the database file
   SpaceTable map;   // map of free/used areas
   NameTable names;  // map names to areas + sizes
   ...
} *DB;

typedef struct RelDescriptor {
   char *relname;    // copy of table name
   int   start;      // page index of start of table data
   int   npages;     // number of pages of table data
   ...
} *Reln;
```

... Single-file Disk Manager

```
// start using DB
DB openDatabase(char *name) {
   DB db = new(DBdescriptor);
   db->dbname = strdup(name);
   db->fd = open(name,O_RDWR);
   db->map = readSpaceTable(db);
   db->names = readNameTable(db);
   return db;
}
// stop using DB and update all meta-data
void closeDatabase(DB db) {
   writeSpaceTable(db,db->map);
   writeNameTable(db,db->names);
   fsync(db->fd);  // ensure that changes reach disk
   close(db->fd);
   free(db);
}
```

... Single-file Disk Manager

```
// set up struct describing relation
Reln openRelation(DB db, char *rname) {
   Reln r = new(RelDescriptor);
   r->relname = strdup(rname);
   // get relation data from map tables
   r->start = ...;
   r->npages = ...;
   return r;
}
// stop using a relation
void closeRelation(Reln r) {
   free(r);
}
#define nPages(r)  (r->npages)
#define makePageID(r,i)  (r->first + i)
```

... Single-file Disk Manager

```
// assume that Page = buffer of PageSize bytes
// assume that PageID = block number in file
// read page from file into memory buffer
void get_page(DB db, PageID p, Page buf) {
   lseek(db->fd, pageOffset(p), SEEK_SET);
   read(db->fd, buf, PAGESIZE);
}

// write page from memory buffer to file
void put_page(Db db, PageID p, Page buf) {
   lseek(db->fd, pageOffset(p), SEEK_SET);
   write(db->fd, buf, PAGESIZE);
}
```

... Single-file Disk Manager

The `pageOffset()` function uses the DB map

- takes a `PageID` value
- uses the DB space map
- returns an absolute file offset

E.g. each table is allocated large contiguous segment of file

- get start address of `relation(PageID)` from map
- add `pageNumber(PageID)*PAGESIZE` to give offset

... Single-file Disk Manager

```
// managing contents of mapping table is complex
// assume a list of (offset,length,status) tuples

// allocate n new pages at end of file
PageID allocate_pages(int n) {
   int endfile = lseek(db->fd, 0, SEEK_END);
   addNewEntry(db->map, endfile, n);
   // note that file itself is not changed
}
// drop n pages starting from p
void deallocate_pages(PageID p, int n) {
   markUnused(db->map, p, n);
   // note that file itself is not changed
}
```

Example: Scanning a Relation

With the above disk manager, the query:

```
select name from Employee
```

might be implemented as something like

```
DB db = openDatabase("myDB");
Reln r = openRelation(db,"Employee");
Page buffer = malloc(PAGESIZE*sizeof(char));
for (int i = 0; i < nPages(r); i++) {
   PageID pid = makePageID(r,i);
   get_page(db, pid, buffer);
   foreach tuple in buffer {
      get tuple data and extract name
```

```
    }
}
```

Exercise 4: Relation Scan Cost

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

- number of tuples *r = 100,000*
- 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_r *= 10msec*

Calculate the total time-cost for answering the query:

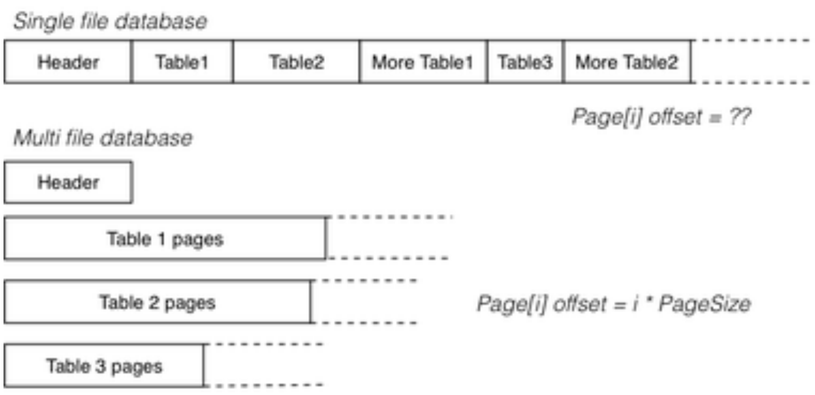
```
select * from R where x > 10;
```

if 50% of the tuples satisfy the condition.

Multi-file Disk Manager

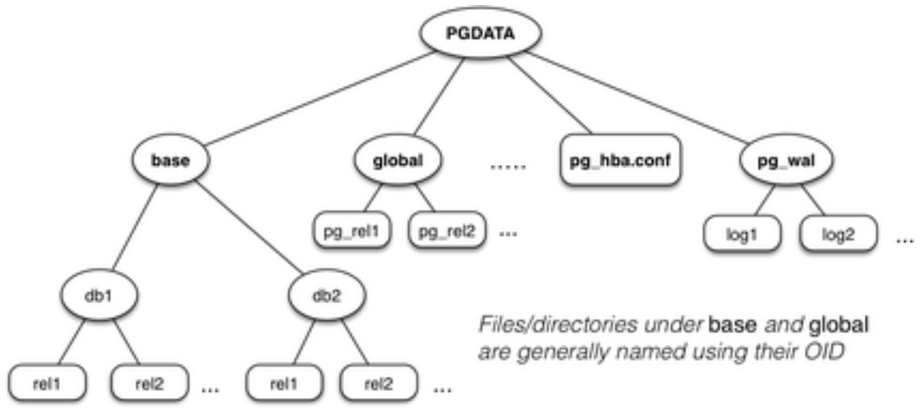
Using multiple files (one file per relation) can be easier

E.g. extending the size of a relation



PostgreSQL Storage Manager

PostgreSQL uses the following file organisation ...



... PostgreSQL Storage Manager

25/60

Components of storage subsystem:

- mapping from relations to files (**`RelFileNode`**)
- abstraction for open relation pool (**`storage/smgr`**)
- functions for managing files (**`storage/smgr/md.c`**)
- file-descriptor pool (**`storage/file`**)

PostgreSQL has two basic kinds of files:

- heap files containing data (tuples)
- index files containing index entries

Note: `smgr` designed for many storage devices; only mag disk handler used

---

Relations as Files

26/60

PostgreSQL identifies relation files via their OIDs.

The core data structure for this is **`RelFileNode`**:

```
typedef struct RelFileNode {
    Oid  spcNode;  // tablespace
    Oid  dbNode;   // database
    Oid  relNode;  // relation
} RelFileNode;
```

Global (shared) tables (e.g. `pg_database`) have

- `spcNode == GLOBALTABLESPACE_OID`
- `dbNode == 0`

... Relations as Files

27/60

The **`relpath`** function maps **`RelFileNode`** to file:

```
char *relpath(RelFileNode r)  // simplified
{
   char *path = malloc(ENOUGH_SPACE);

   if (r.spcNode == GLOBALTABLESPACE_OID) {
      /* Shared system relations live in PGDATA/global */
      Assert(r.dbNode == 0);
      sprintf(path, "%s/global/%u",
              DataDir, r.relNode);
   }
   else if (r.spcNode == DEFAULTTABLESPACE_OID) {
      /* The default tablespace is PGDATA/base */
      sprintf(path, "%s/base/%u/%u",
              DataDir, r.dbNode, r.relNode);
   }
   else {
      /* All other tablespaces accessed via symlinks */
      sprintf(path, "%s/pg_tblspc/%u/%u/%u", DataDir
              r.spcNode, r.dbNode, r.relNode);
   }
   return path;
}
```

File Descriptor Pool

Unix has limits on the number of concurrently open files.

PostgreSQL maintains a pool of open file descriptors:

- to hide this limitation from higher level functions
- to minimise expensive `open()` operations

File names are simply strings: **`typedef char *FileName`**

Open files are referenced via: **`typedef int File`**

A **`File`** is an index into a table of "virtual file descriptors".

... File Descriptor Pool

Interface to file descriptor (pool):

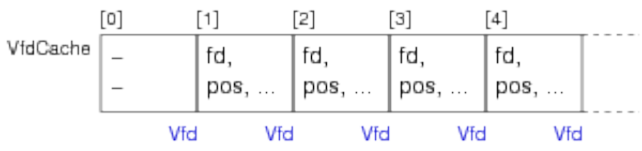
```
File FileNameOpenFile(FileName fileName,
                      int fileFlags, int fileMode);
     // open a file in the database directory ($PGDATA/base/...)
File OpenTemporaryFile(bool interXact);
     // open temp file; flag: close at end of transaction?
void FileClose(File file);
void FileUnlink(File file);
int  FileRead(File file, char *buffer, int amount);
int  FileWrite(File file, char *buffer, int amount);
int  FileSync(File file);
long FileSeek(File file, long offset, int whence);
int  FileTruncate(File file, long offset);
```

Analogous to Unix syscalls `open()`, `close()`, `read()`, `write()`, `lseek()`, ...

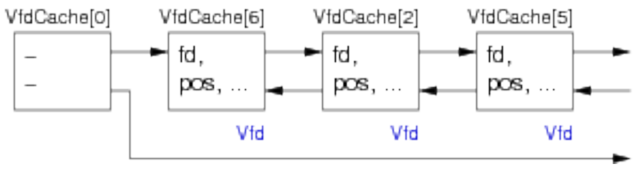
... File Descriptor Pool

Virtual file descriptors (**`Vfd`**)

- physically stored in dynamically-allocated array



- also arranged into list by recency-of-use



`VfdCache[0]` holds list head/tail pointers.

... File Descriptor Pool

31/60

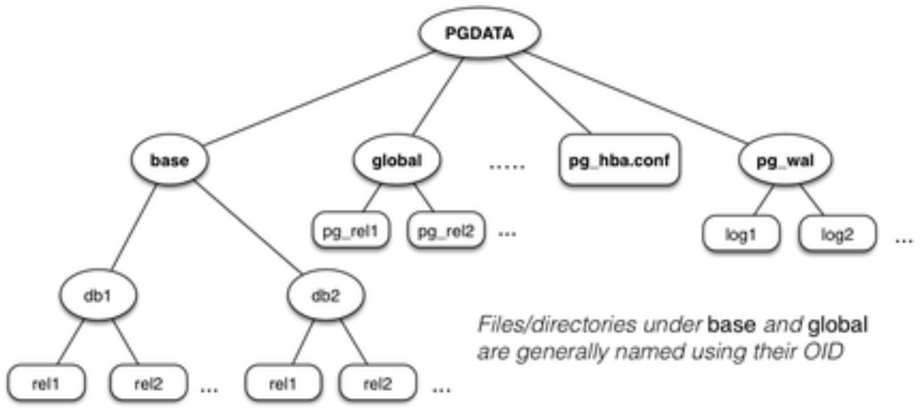
Virtual file descriptor records (simplified):

```
typedef struct vfd
{
    s_short  fd;              // current FD, or VFD_CLOSED if none
    u_short  fdstate;         // bitflags for VFD's state
    File     nextFree;        // link to next free VFD, if in freelist
    File     lruMoreRecently; // doubly linked recency-of-use list
    File     lruLessRecently;
    long     seekPos;         // current logical file position
    char     *fileName;       // name of file, or NULL for unused VFD
    // NB: fileName is malloc'd, and must be free'd when closing the VFD
    int      fileFlags;       // open(2) flags for (re)opening the file
    int      fileMode;        // mode to pass to open(2)
} Vfd;
```

File Manager

32/60

Reminder: PostgreSQL file organisation

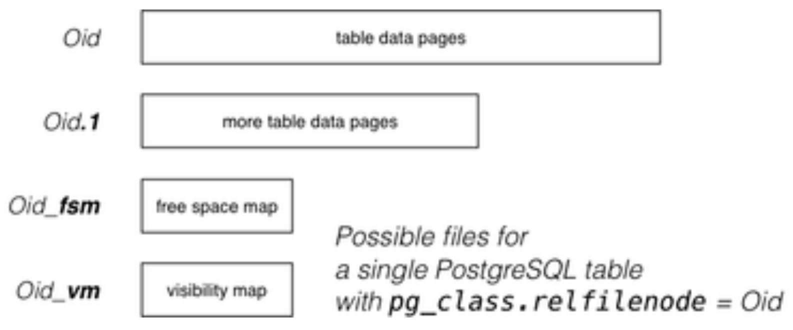


... File Manager

33/60

PostgreSQL stores each table

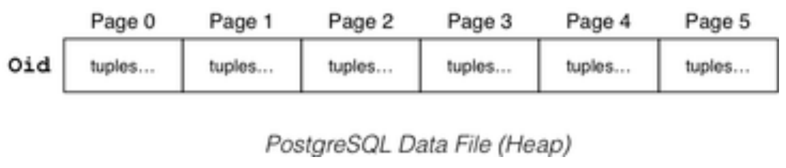
- in the directory *PGDATA*/`pg_database.oid`
- often in multiple files (aka *forks*)



... File Manager

Data files (*Oid*, *Oid*`.1`, ...):

- sequence of fixed-size blocks/pages (typically 8KB)
- each page contains tuple data and admin data (see later)
- max size of data files 1GB (Unix limitation)



... File Manager

Free space map (*Oid*`_fsm`):

- indicates where free space is in data pages
- "free" space is only free after `VACUUM`
  (`DELETE` simply marks tuples as no longer in use `xmax`)

Visibility map (*Oid*`_vm`):

- indicates pages where all tuples are "visible"
  (*visible* = accessible to all currently active transactions)
- such pages can be ignored by `VACUUM`
- also used for index pages, to indicate all index entries visible
  (allows *index-only scans* to be done more efficiently)

... File Manager

Relation files are identified via `pg_class.relfilenode`

- most of the time, this is the same as `pg_class.oid`
- and with `forkNum` appended, unless it is zero

The core data structure for this is **`RelFileNode`**:

```
typedef struct RelFileNode {
    Oid  spcNode;  // tablespace
    Oid  dbNode;   // database
    Oid  relNode;  // relation
} RelFileNode;
```

Some relations have no `relfilenode`; use `pg_filenode.map` file

... File Manager

The "magnetic disk storage manager" (**`storage/smgr/md.c`**)

- manages its own pool of open file descriptors (Vfd's)
- may use several Vfd's to access data, if several forks
- manages mapping from **`PageID`** to file+offset.

PostgreSQL `PageID` values are structured:

```
typedef struct
{
    RelFileNode rnode;    // which relation/file
    ForkNumber  forkNum;  // which fork (of reln)
    BlockNumber blockNum; // which page/block
} BufferTag;
```

... File Manager

Access to a block of data proceeds (roughly) as follows:

```
// pageID set from pg_catalog tables
// buffer obtained from Buffer pool
getBlock(BufferTag pageID, Buffer buf)
{
   Vfd vf;  off_t offset;
   (vf, offset) = findBlock(pageID)
   lseek(vf.fd, offset, SEEK_SET)
   vf.seekPos = offset;
   nread = read(vf.fd, buf, BLOCKSIZE)
   if (nread < BLOCKSIZE) ... we have a problem
}
```

`BLOCKSIZE` is a global configurable constant (default: 8192)

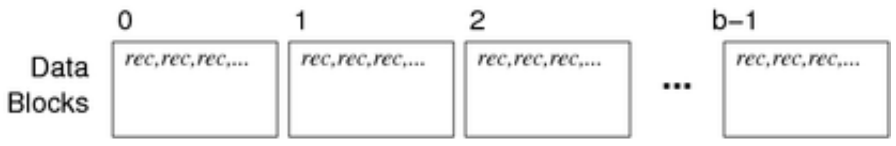
... File Manager

```
findBlock(BufferTag pageID) returns (Vfd, off_t)
{
   offset = pageID.BlockNumber * BLOCKSIZE
   fileID = relpath(pageID.rnode)
   if (pageID.forkNum > 0)
      fileID = fileID+"."+pageID.ForkNum
   if (fileID is not in Vfd pool)
      fd = allocate new Vfd for this fileID
   else
      fd = use Vfd from pool
   if (offset > fd.fileSize) {
      fd = allocate new Vfd for next fork
      offset = offset - fd.fileSize
   }
   return (fd, offset)
}
```

DBMS Parameters

Our view of relations 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



... DBMS Parameters

Typical DBMS/table parameter values:

Quantity	Symbol	E.g. Value
total # tuples	r	10^6
record size	R	128 bytes
total # pages	b	10^5
page size	B	8192 bytes
# tuples per page	c	60
page read/write time	T_r ,T_w	10 msec
cost to process one page in memory	-	$\cong 0$

Buffer Pool

Buffer Pool

Aim of buffer pool:

- hold pages read from database files, for possible re-use

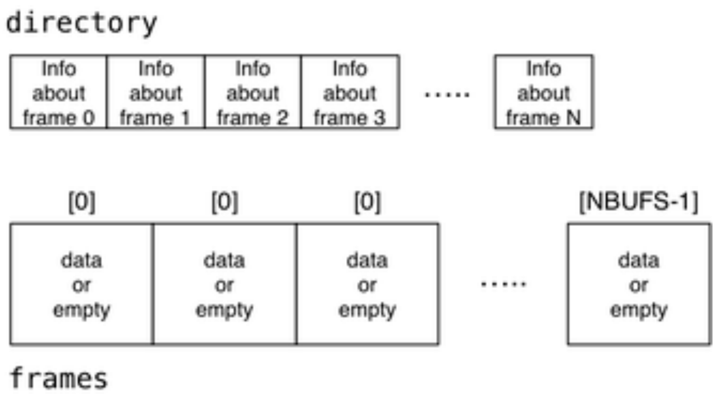
Buffer pool operations: (both take single `PageID` argument)

- `request_page(pid)`, `release_page(pid)`, ...

Buffer pool data structures:

- `Page frames[NBUFS]; FrameData directory[NBUFS];`
- `Page` is `byte[BUFSIZE]`, `FrameData` is `struct {...}`

... Buffer Pool



... Buffer Pool

For each frame, we need to know: (`FrameData`)

- which Page it contains, or whether empty/free
- whether it has been modified since loading (*dirty bit*)
- how many transactions are currently using it (*pin count*)
- time-stamp for most recent access (assists with replacement)

which page = PageID = (RelationID,PageNum) (note: Page = Block)

... Buffer Pool

How scans are performed without Buffer Pool:

```
Buffer buf;
int N = numberOfBlocks(Rel);
for (i = 0; i < N; i++) {
   pageID = makePageID(db,Rel,i);
   getBlock(pageID, buf);
   for (j = 0; j < nTuples(buf); j++)
      process(buf, j)
}
```

Requires `N` page reads.

If we read it again, `N` page reads.

... Buffer Pool

How scans are performed with Buffer Pool:

```
Buffer buf;
int N = numberOfBlocks(Rel);
for (i = 0; i < N; i++) {
   pageID = makePageID(db,Rel,i);
   bufID = request_page(pageID);
   buf = frames[bufID]
   for (j = 0; j < nTuples(buf); j++)
      process(buf, j)
   release_page(pageID);
}
```

Requires `N` page reads on the first pass.

If we read it again, 0 ≤ page reads ≤ `N`

... Buffer Pool

Implementation of `request_page()`

```
int request_page(PageID pid)
{
   if (pid in Pool)
      bufID = index for pid in Pool
   else {
      if (no free frames in Pool)
         evict a page (free a frame)
      bufID = allocate free frame
      directory[bufID].page = pid
      directory[bufID].pin_count = 0
      directory[bufID].dirty_bit = 0
   }
   directory[bufID].pin_count++
   return bufID
}
```

... Buffer Pool

Implementation of `release_page()`

```
int release_page(PageID pid)
{
   bufID = index for pid in Pool
   directory[bufID].pin_count--
}
```

... Buffer Pool

Evicting a page ...

- find frame(s) preferably satisfying
    - pin count = 0   (i.e. nobody using it)
    - dirty bit = 0   (not modified)
- if selected frame was modified, flush frame to disk
- flag directory entry as "frame empty"

If multiple frames can potentially be released

- need a policy to decide which is best choice

Page Replacement Policies

Several schemes are commonly in use:

- Least Recently Used (LRU)
- Most Recently Used (MRU)

- First in First Out (FIFO)
- Random

LRU / MRU require knowledge of when pages were last accessed

- how to keep track of "last access" time?
- base on request/release ops or on *real* page usage?

... Page Replacement Policies

Cost benefit from buffer pool (with *n* frames) is determined by:

- number of available frames (more ⇒ better)
- replacement strategy vs page access pattern

Example (a): sequential scan, LRU or MRU, $n \geq b$

First scan costs *b* reads; subsequent scans are "free".

Example (b): sequential scan, MRU, $n < b$

First scan costs *b* reads; subsequent scans cost $b - n$ reads.

Example (c): sequential scan, LRU, $n < b$

All scans cost *b* reads; known as *sequential flooding*.

Effect of Buffer Management

Consider a query to find customers who are also employees:

```
select c.name
from   Customer c, Employee e
where  c.ssn = e.ssn;
```

This might be implemented inside the DBMS via nested loops:

```
for each tuple t1 in Customer {
    for each tuple t2 in Employee {
        if (t1.ssn == t2.ssn)
            append (t1.name) to result set
    }
}
```

... Effect of Buffer Management

In terms of page-level operations, the algorithm looks like:

```
Rel rC = openRelation("Customer");
Rel rE = openRelation("Employee");
for (int i = 0; i < nPages(rC); i++) {
    PageID pid1 = makePageID(db,rC,i);
    Page p1 = request_page(pid1);
    for (int j = 0; j < nPages(rE); j++) {
        PageID pid2 = makePageID(db,rE,j);
        Page p2 = request_page(pid2);
        // compare all pairs of tuples from p1,p2
        // construct solution set from matching pairs
        release_page(pid2);
    }
    release_page(pid1);
}
```

Exercise 5: Buffer Management Cost Benefit (i)

Assume that:

- the `Customer` relation has b_C pages (e.g. 5)
- the `Employee` relation has b_E pages (e.g. 4)

Compute how many page reads occur ...

- if we have only 2 buffers (i.e. effectively no buffer pool)
- 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 $n < b_E$)

Exercise 6: Buffer Management 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)

PostgreSQL Buffer Manager

PostgreSQL buffer manager:

- provides a shared pool of memory buffers for all backends
- all access methods get data from disk via buffer manager

Buffers are located in a large region of shared memory.

Definitions: **`src/include/storage/buf*.h`**

Functions: **`src/backend/storage/buffer/*.c`**

Buffer code is also used by backends who want a private buffer pool

... PostgreSQL Buffer Manager

Buffer pool consists of:

`BufferDescriptors`

- shared fixed array (size `NBuffers`) of **`BufferDesc`**

`BufferBlocks`

- shared fixed array (size `NBuffers`) of 8KB frames

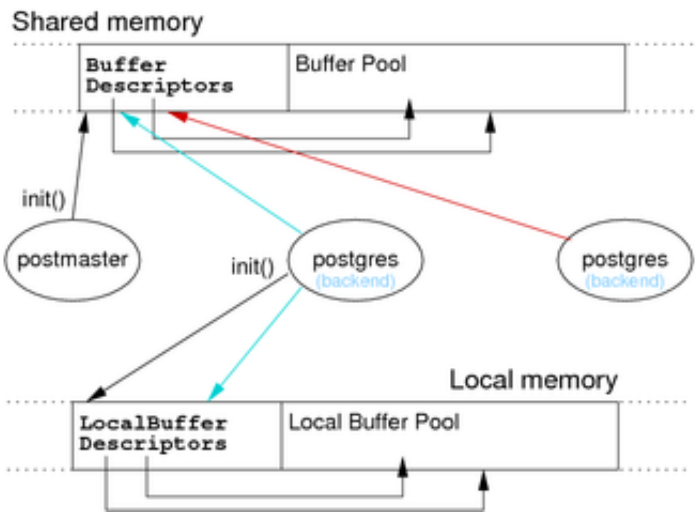
`Buffer` = index values in above arrays

- indexes: global buffers `1..NBuffers`; local buffers negative

Size of buffer pool is set in *postgresql.conf*, e.g.

```
shared_buffers = 16MB   # min 128KB, 16*8KB buffers
```

... PostgreSQL Buffer Manager



... PostgreSQL Buffer Manager

`include/storage/buf.h`

- basic buffer manager data types (e.g. **`Buffer`**)

`include/storage/bufmgr.h`

- definitions for buffer manager function interface
  (i.e. functions that other parts of the system call to use buffer manager)

`include/storage/buf_internals.h`

- definitions for buffer manager internals (e.g. **`BufferDesc`**)

Code: **`backend/storage/buffer/*.c`**

Commentary: **`backend/storage/buffer/README`**

Produced: 2 Aug 2018