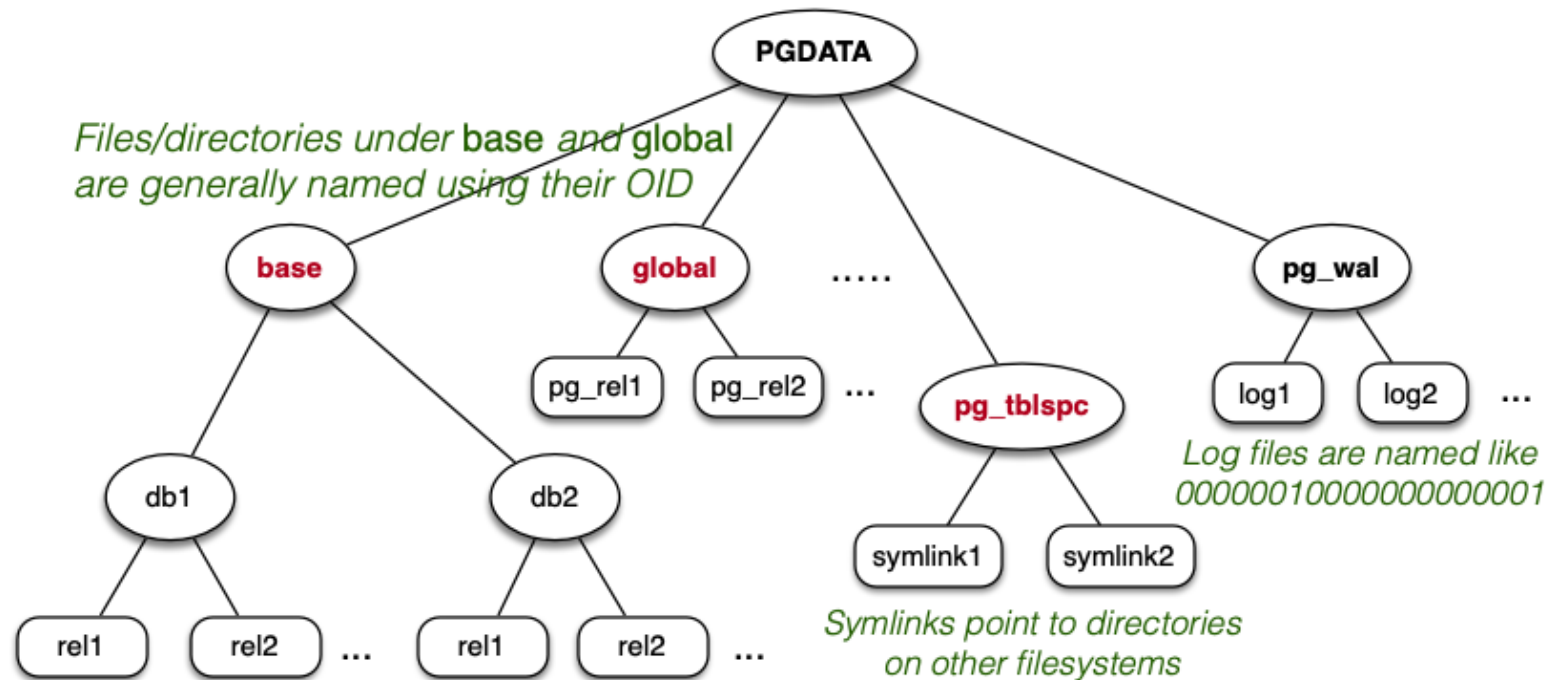


PostgreSQL File Manager

- PostgreSQL File Manager
- Relations as Files
- File Descriptor Pool
- File Manager

❖ PostgreSQL File Manager

PostgreSQL uses the following file organisation ...



❖ PostgreSQL File Manager (cont)

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 disk handler provided

❖ Relations as Files

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 (cont)

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;
}
```

COMP9315 21T1 ◇ PG File Manager ◇ [4/15]

❖ 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 (cont)

Interface to file descriptor (pool):

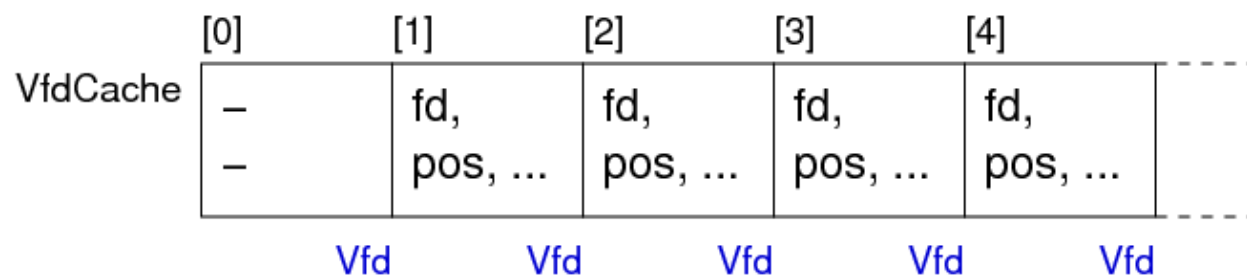
```
File FileNameOpenFile(FileName fileName,
                      int fileFlags, int fileMode);
    // open a file in the database directory ($PGDATA/base/...)
File OpenTemporaryFile(bool interXact);
    // open temp file; flag: close at end of transaction?
void FileClose(File file);
void FileUnlink(File 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()**, ...

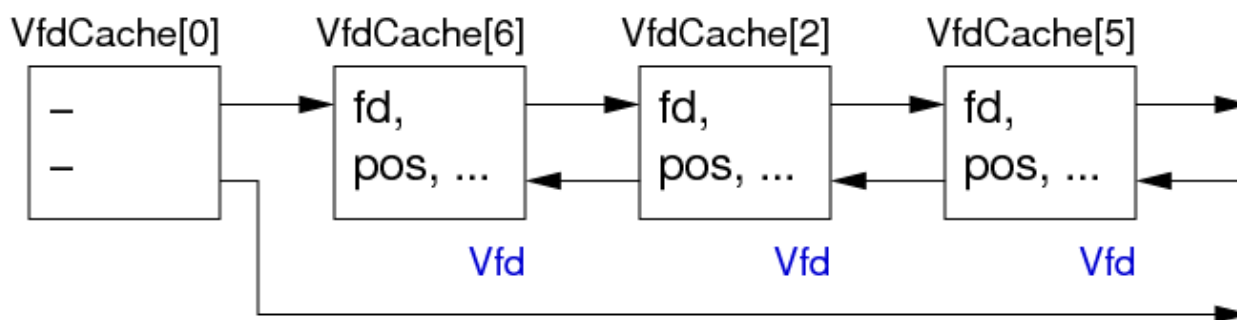
❖ File Descriptor Pool (cont)

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.

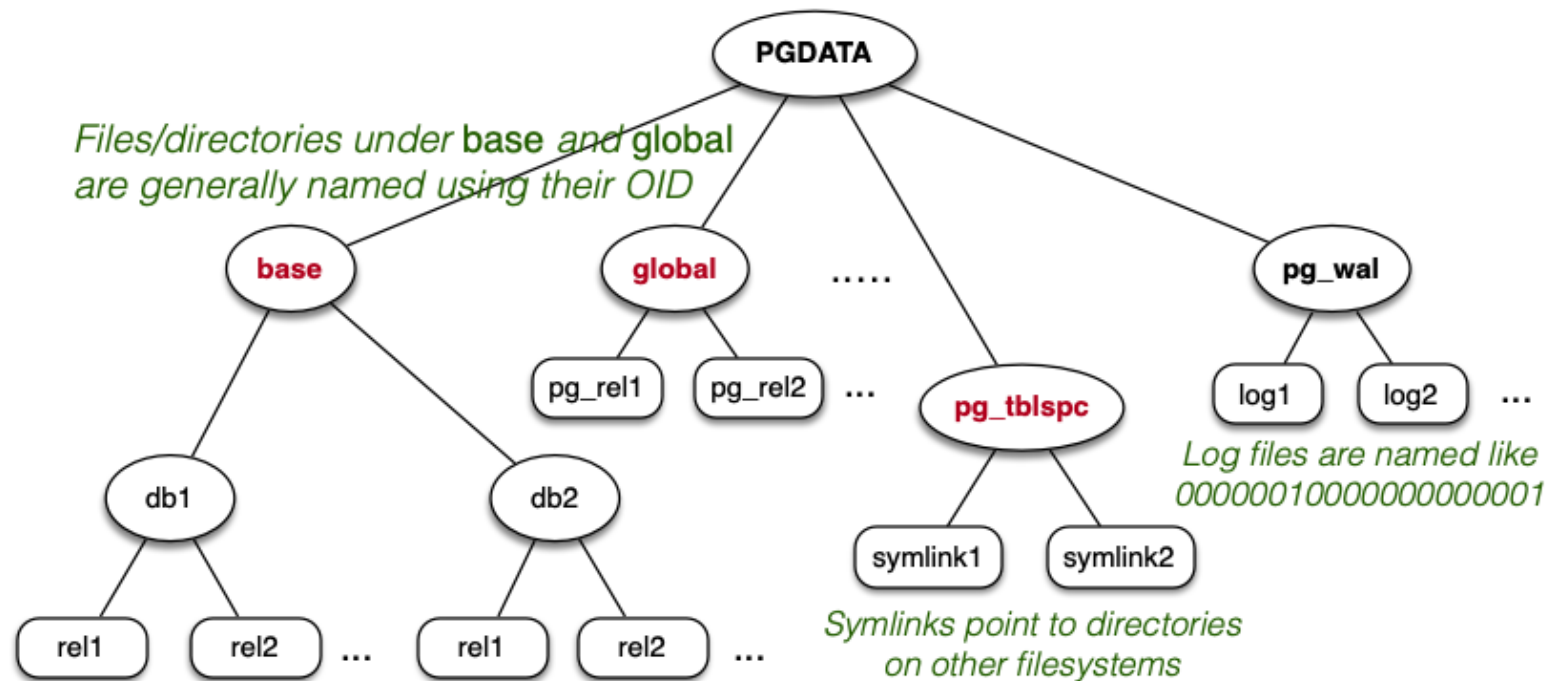
❖ File Descriptor Pool (cont)

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

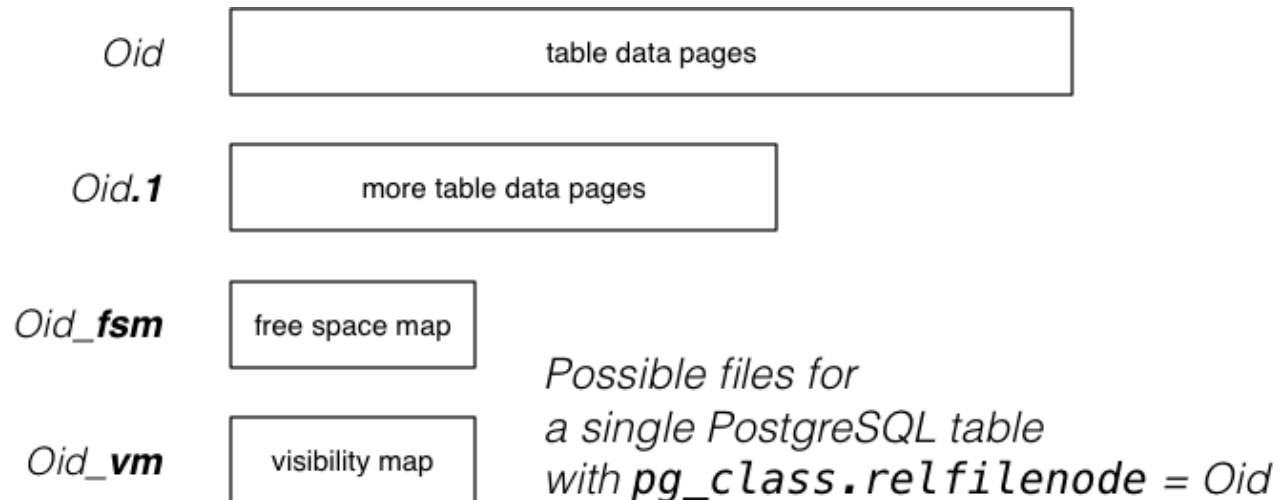
Reminder: PostgreSQL file organisation



❖ File Manager (cont)

PostgreSQL stores each table

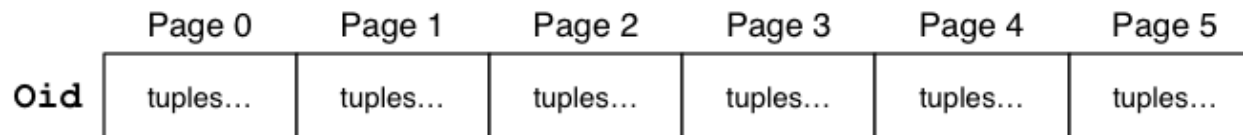
- in the directory ***PGDATA/pg_database.oid***
- often in multiple files (aka *forks*)



❖ File Manager (cont)

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)



PostgreSQL Data File (Heap)

❖ File Manager (cont)

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_vn*):

- indicates pages where all tuples are "visible"
(*visible* = accessible to all currently active transactions)
- such pages can be ignored by **VACUUM**

❖ File Manager (cont)

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 (cont)

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 (cont)

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

Produced: 21 Feb 2021