File Management

❖ 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 (cont)

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?

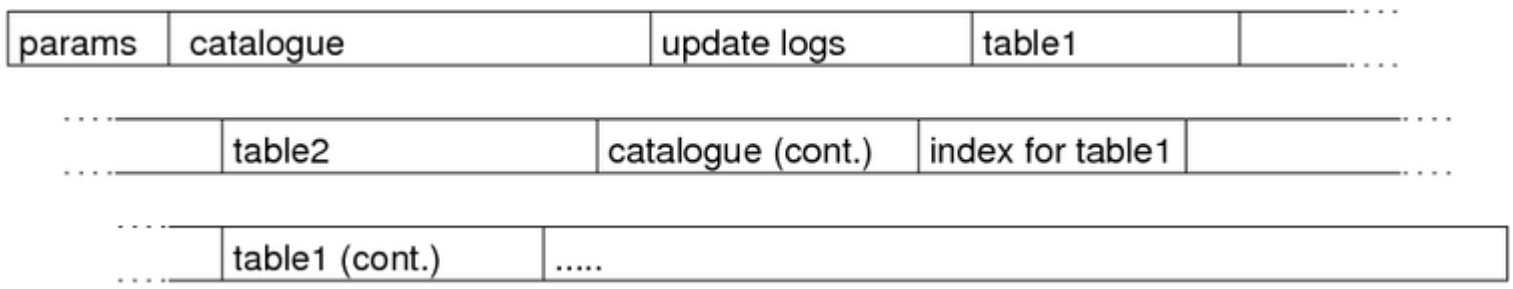
Different DBMSs make different choices, e.g.

- 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 (cont)

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

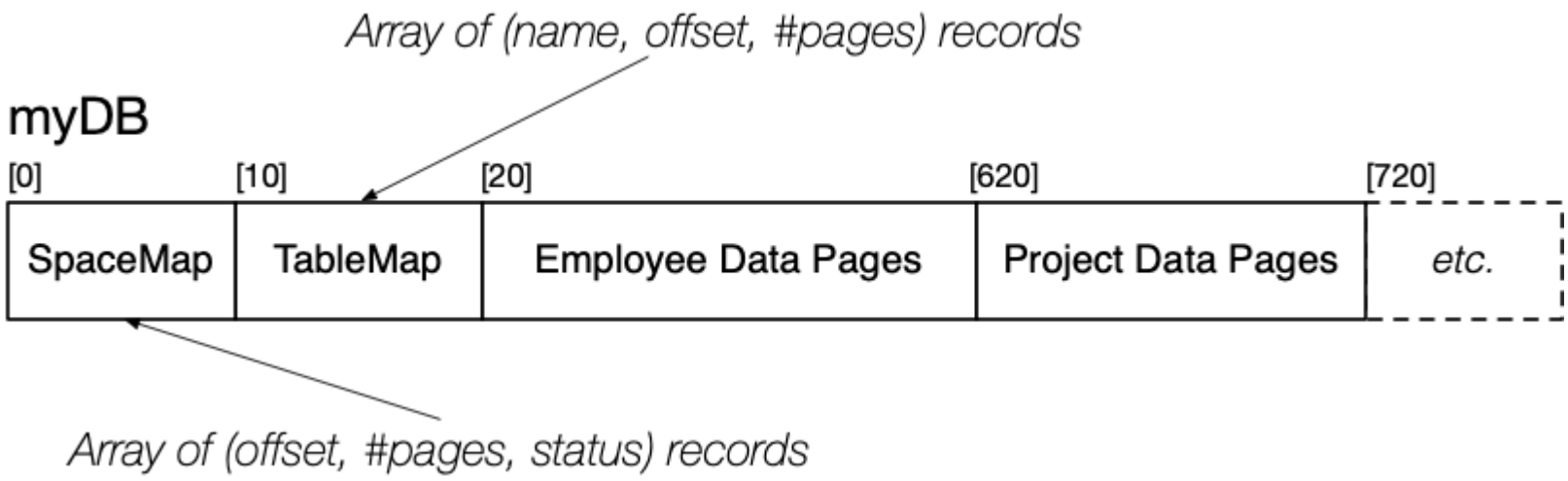
If the seek goes way beyond the end of the file:

- 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 Storage Manager

Consider the following simple single-file DBMS layout:



E.g.

SpaceMap = [(0,10,U), (10,10,U), (20,600,U), (620,100,U), (720,20,F)]

TableMap = [("employee",20,500), ("project",620,40)]

❖ Single-file Storage Manager (cont)

Each file segment consists of a number fixed-size blocks

The following data/constant definitions are useful

```
#define PAGESIZE 2048    // bytes per page

typedef long PageId;     // PageId is block index
                         // pageOffset=PageId*PAGESIZE

typedef char *Page;      // pointer to page/block buffer
```

Typical **`PAGESIZE`** values: 1024, 2048, 4096, 8192

❖ Single-file Storage Manager (cont)

Possible storage manager data structures for opened DBs & Tables

```
typedef struct DBrec {
    char *dbname;      // copy of database name
    int fd;            // the database file
    SpaceMap map;      // map of free/used areas
    TableMap names;    // map names to areas + sizes
} *DB;

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

❖ Example: Scanning a Relation

With the above disk manager, a query like

```
select name from Employee
```

might be implemented as

```
DB db = openDatabase("myDB");
Rel r = openRelation(db,"Employee");
Page buffer = malloc(PAGESIZE*sizeof(char));
for (int i = 0; i < r->npages; i++) {
   PageId pid = r->start+i;
   get_page(db, pid, buffer);
   for each tuple in buffer {
      get tuple data and extract name
      add (name) to result tuples
   }
}
```

❖ Single-File Storage Manager

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

❖ Single-File Storage Manager (cont)

```
// set up struct describing relation
Rel openRelation(DB db, char *rname) {
    Rel r = new(struct Relrec);
    r->relname = strdup(rname);
    // get relation data from map tables
    r->start = ...;
    r->npages = ...;
    return r;
}

// stop using a relation
void closeRelation(Rel r) {
    free(r->relname);
    free(r);
}
```

❖ Single-File Storage Manager (cont)

```
// assume that Page = byte[PageSize]
// 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, p*PAGESIZE, 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, p*PAGESIZE, SEEK_SET);
    write(db->fd, buf, PAGESIZE);
}
```

❖ Single-File Storage Manager (cont)

Managing contents of space mapping table can be complex:

```
// assume an array of (offset,length,status) records

// allocate n new pages
PageId allocate_pages(int n) {
    if (no existing free chunks are large enough) {
        int endfile = lseek(db->fd, 0, SEEK_END);
        addNewEntry(db->map, endfile, n);
    } else {
        grab "worst fit" chunk
        split off unused section as new chunk
    }
    // note that file itself is not changed
}
```

❖ Single-File Storage Manager (cont)

Similar complexity for freeing chunks

```
// drop n pages starting from p
void deallocate_pages(PageId p, int n) {
    if (no adjacent free chunks) {
        markUnused(db->map, p, n);
    } else {
        merge adjacent free chunks
        compress mapping table
    }
    // note that file itself is not changed
}
```

Changes take effect when **`closeDatabase()`** executed.

❖ Multiple-file Disk Manager

Most DBMSs don't use a single large file for all data.

They typically provide:

- multiple files partitioned physically or logically
- mapping from DB-level objects to files (e.g. via catalog meta-data)

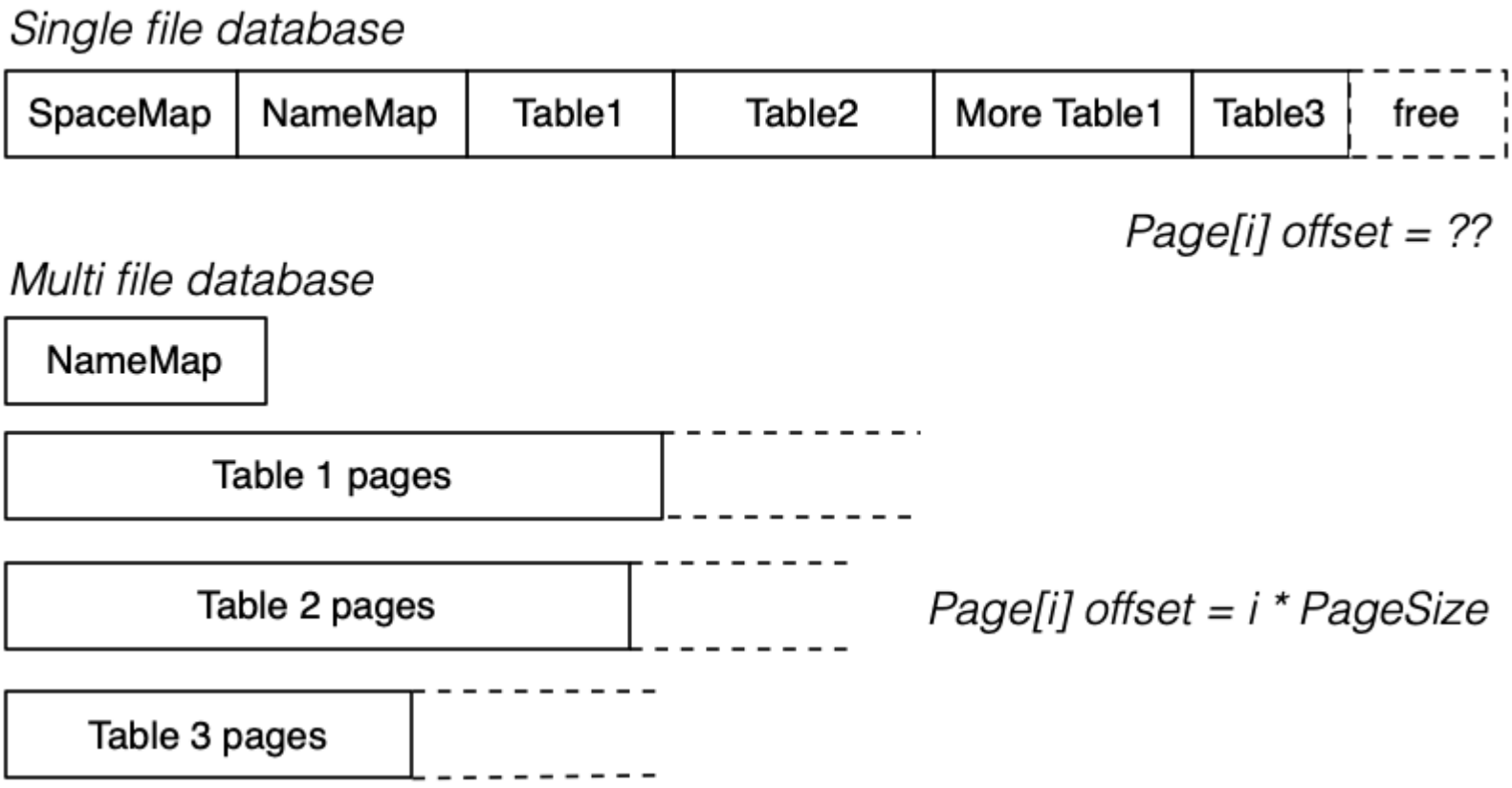
Precise file structure varies between individual DBMSs.

Using multiple files (one file per relation) can be easier, e.g.

- adding a new relation
- extending the size of a relation
- computing page offsets within a relation

❖ Multiple-file Disk Manager (cont)

Example of single-file vs multiple-file:



Consider how you would compute file offset of page[i] in table[1] ...

❖ Multiple-file Disk Manager (cont)

Structure of **`PageId`** for data pages in such systems ...

If system uses one file per table, **`PageId`** contains:

- relation indentifier (which can be mapped to filename)
- page number (to identify page within the file)

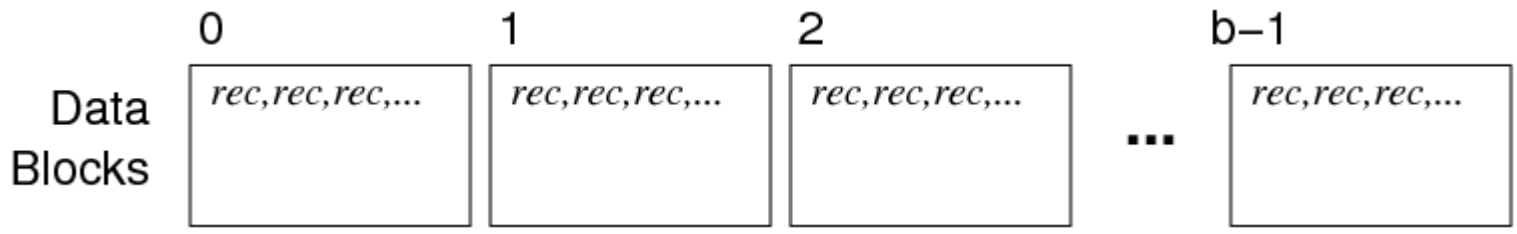
If system uses several files per table, **`PageId`** contains:

- relation identifier
- file identifier (combined with relid, gives filename)
- page number (to identify page within the file)

❖ DBMS File 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 File Parameters (cont)

Typical DBMS/table parameter values:

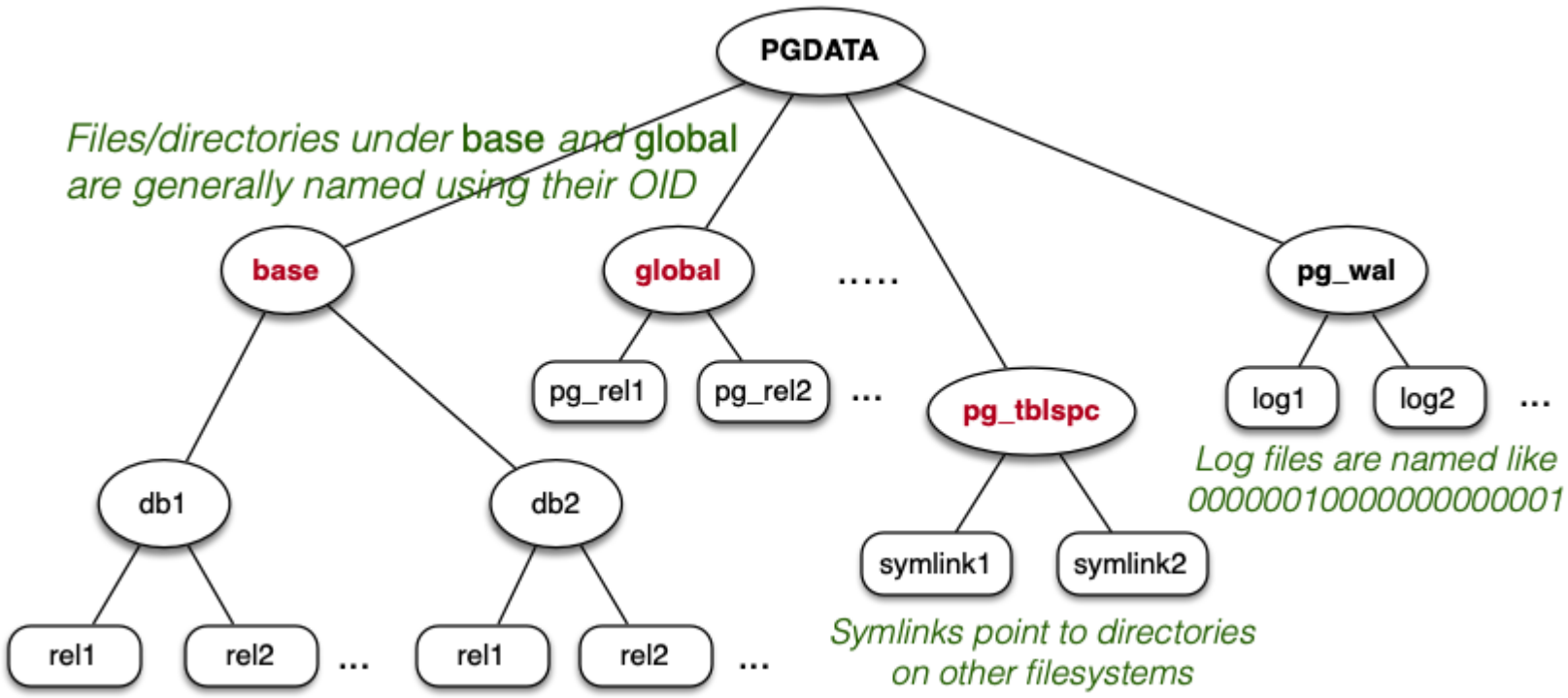
Quantity	Symbol	E.g. Value
total # tuples	r	10^6
record size	R	128 bytes
total # pages	b	10^5
page size	B	8192 bytes
# tuples per page	c	60
page read/write time	T_r, T_w	10 msec
cost to process one page in memory	-	$\cong 0$

Produced: 31 Jan 2023

PostgreSQL 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`**:

```
// include/storage/relfilenode.h
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:

```
// include/common/relpath.h
// common/relpath.c
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 (cont)

Interface to file descriptor (pool):

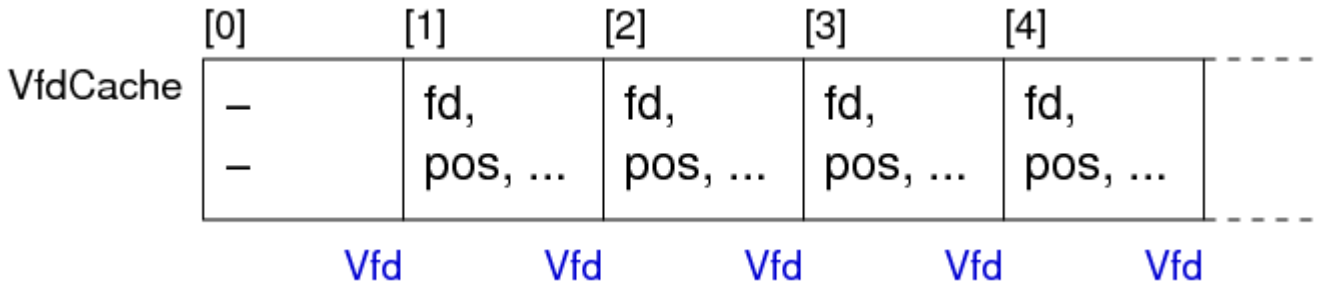
```
backend/storage/file/fd.c
File FileNameOpenFile(FileName fileName,
                      int fileFlags, int fileMode);
     // open a file in the database directory ($PGDATA/base/...)
File OpenTemporaryFile(bool interXact);
     // open temp file; flag: close at end of transaction?
void FileClose(File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

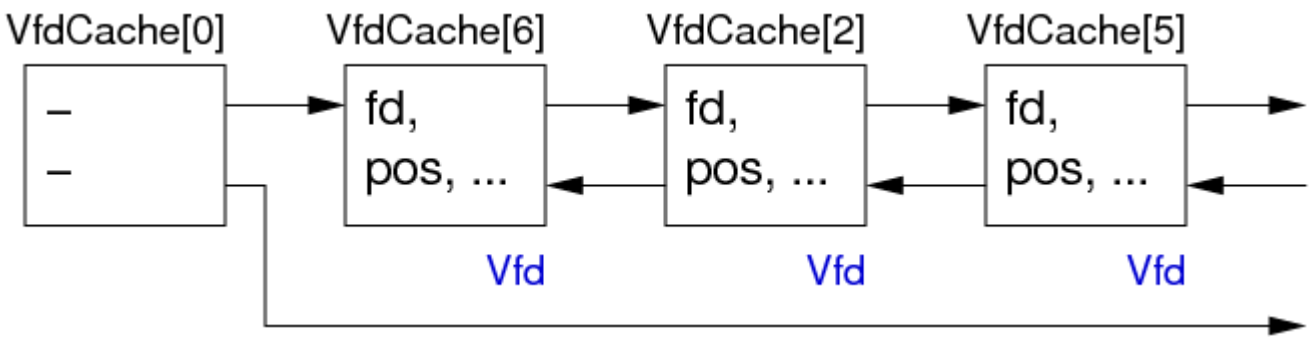
❖ File Descriptor Pool (cont)

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.

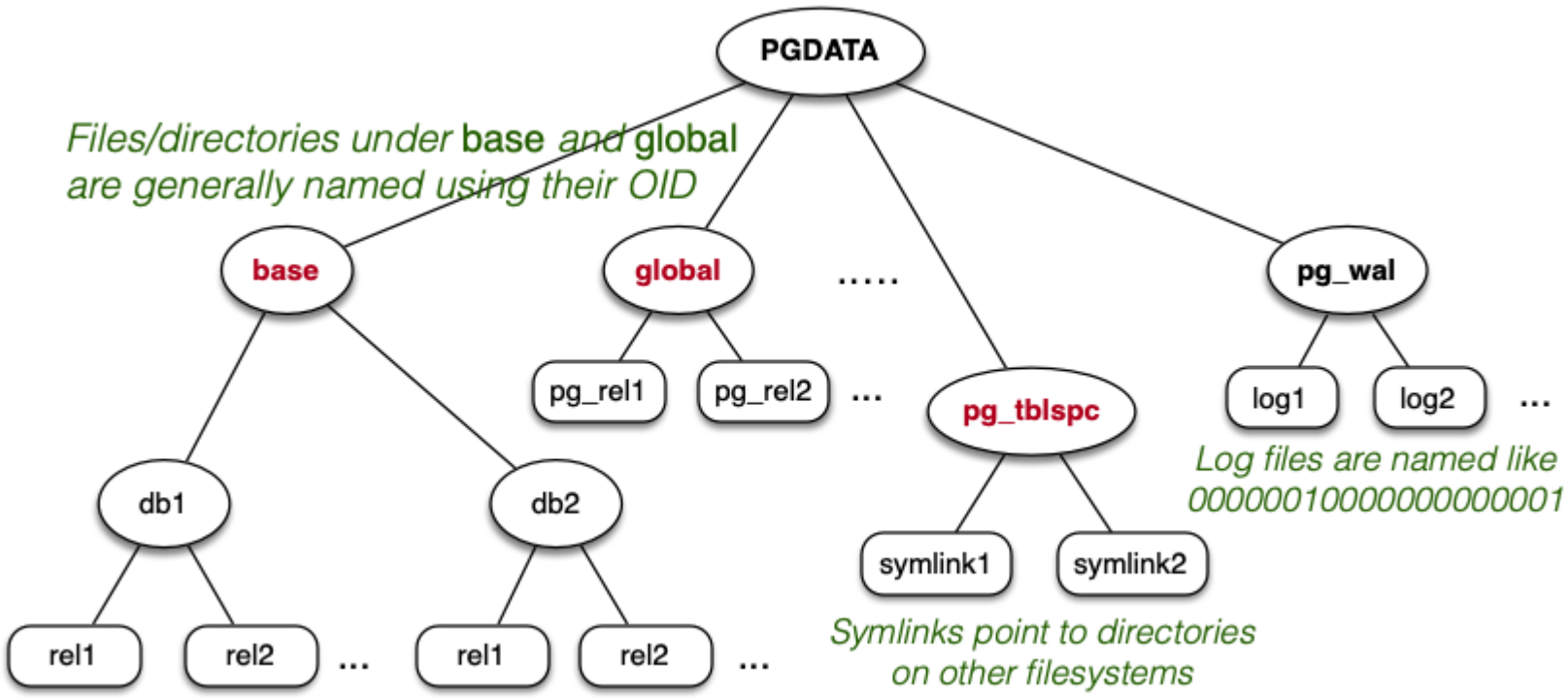
❖ File Descriptor Pool (cont)

Virtual file descriptor records (simplified):

```
backend/storage/file/fd.c
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

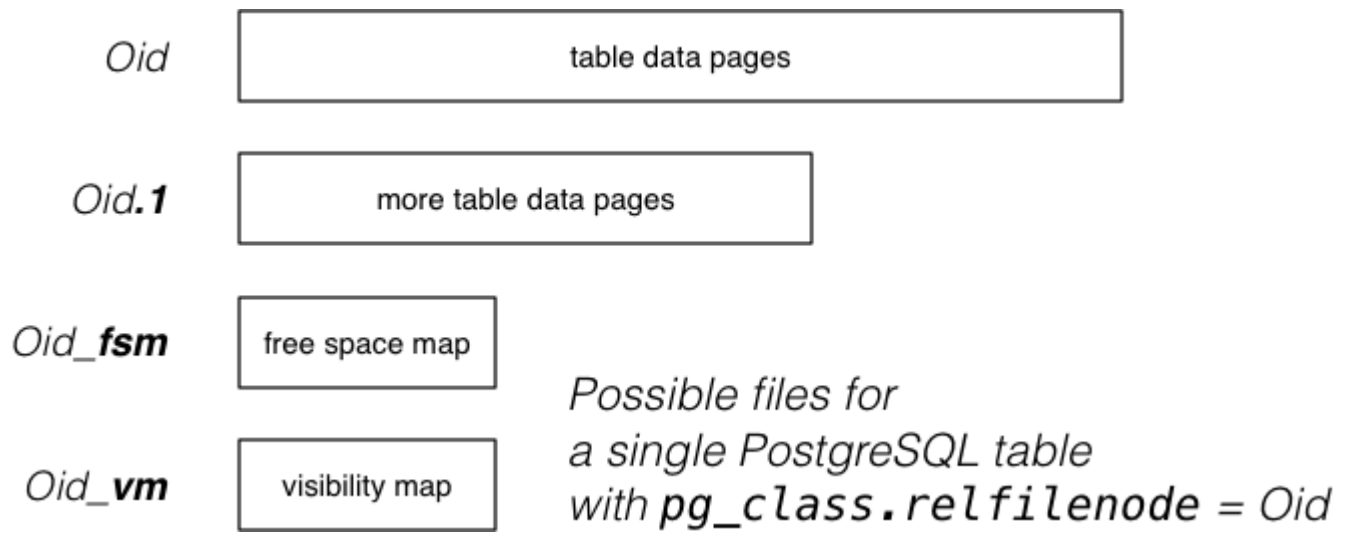
Reminder: PostgreSQL file organisation



❖ File Manager (cont)

PostgreSQL stores each table

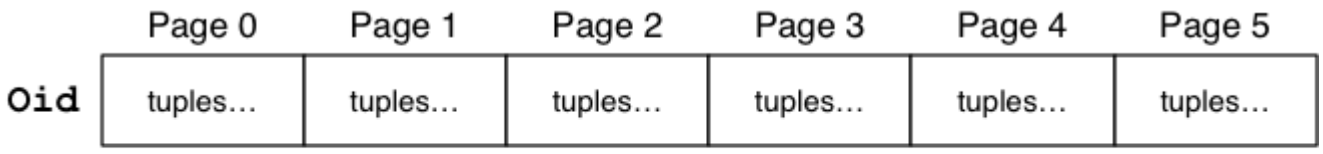
- in the directory ***PGDATA*/pg_database.oid**
- often in multiple files (aka *forks*)



❖ 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*_`vm`):

- 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:

```
include/storage/buf_internals.h
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)

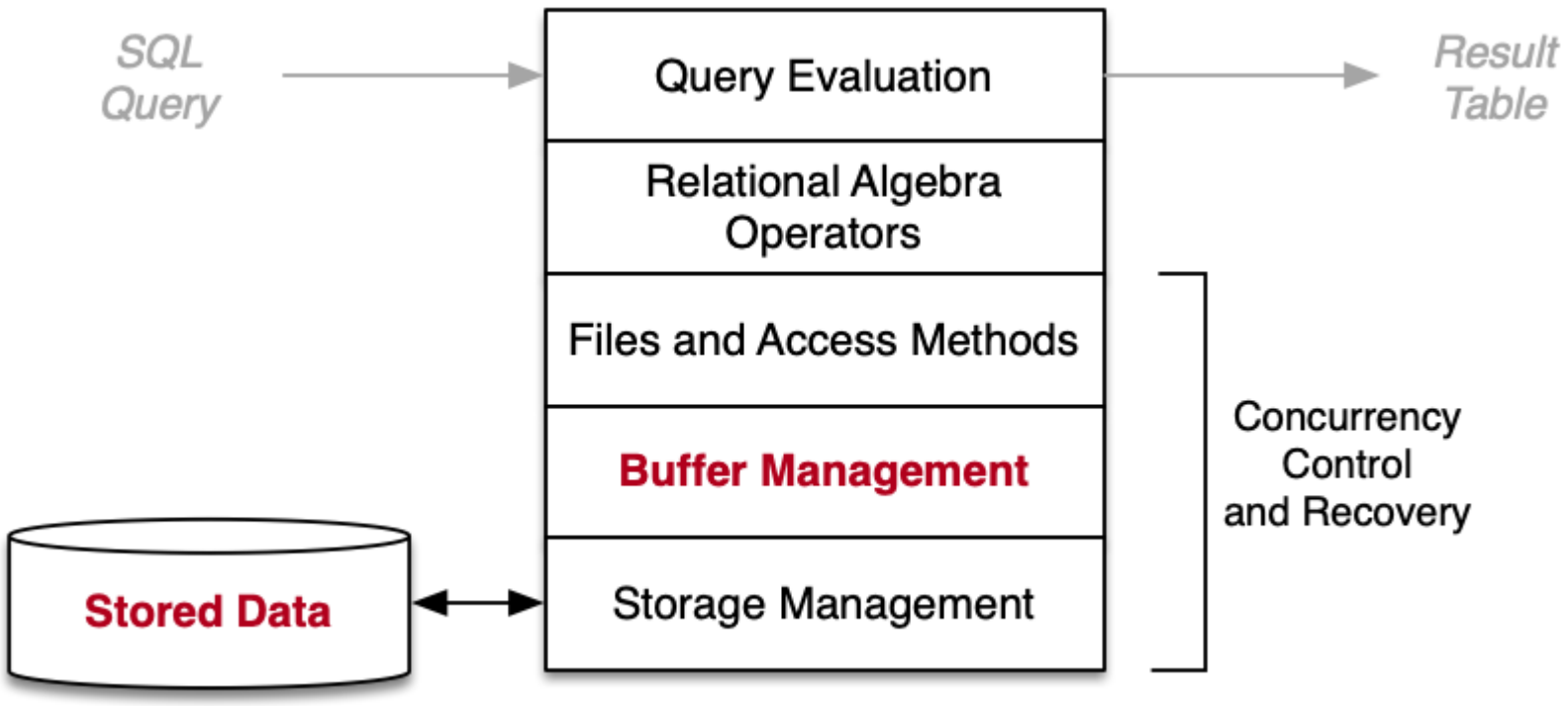
❖ File Manager (cont)

```
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 (pageID.forkNum > 0) {
      offset = offset - (pageID.forkNum*MAXFILESIZE)
   }
   return (fd, offset)
}
```

Produced: 20 Feb 2023

Buffer Pool

❖ Buffer Pool


SQL Query
Query Evaluation
Result Table
Relational Algebra Operators
Files and Access Methods
Buffer Management
Storage Management
Concurrency Control and Recovery
Stored Data

❖ Buffer Pool (cont)

Aim of buffer pool:

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

Used by:

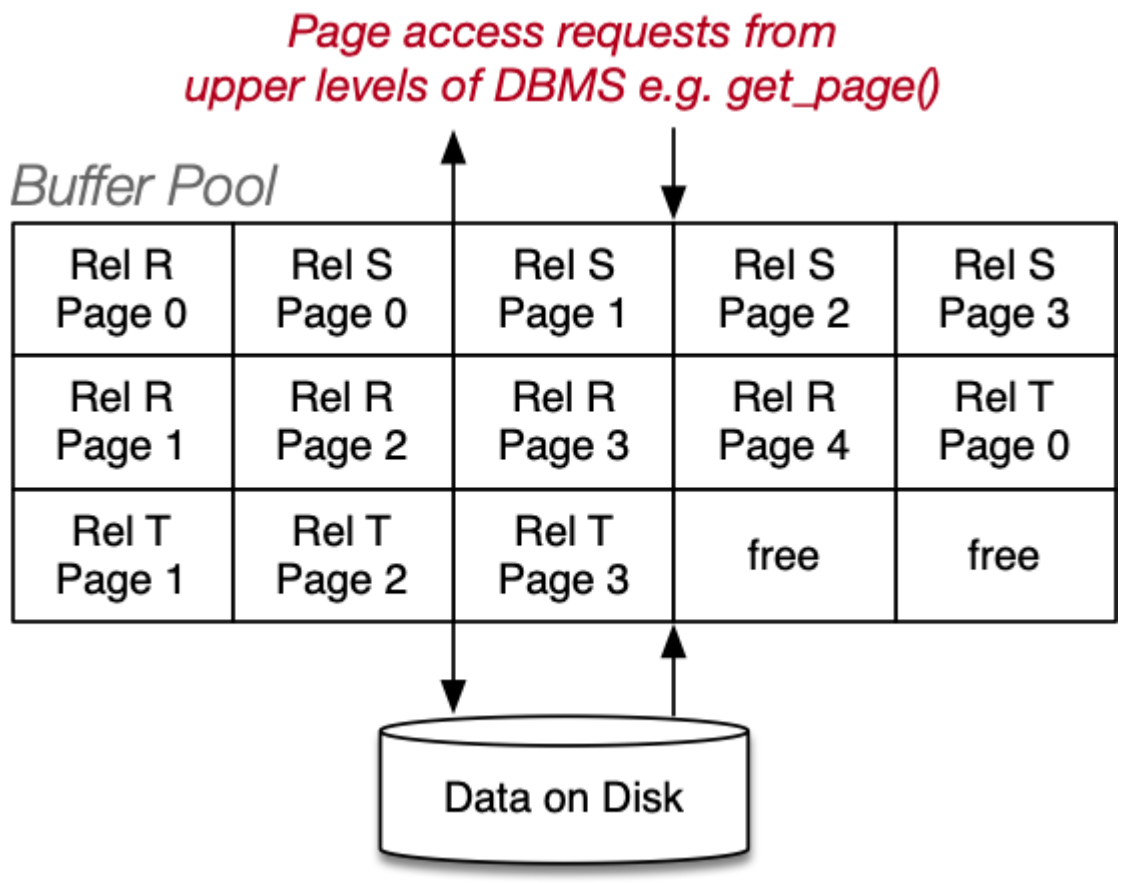
- access methods which read/write data pages
- e.g. sequential scan, indexed retrieval, hashing

Uses:

- file manager functions to access data files

Note: we use the terms page and block interchangably

❖ Buffer Pool (cont)



❖ Buffer Pool (cont)

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

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

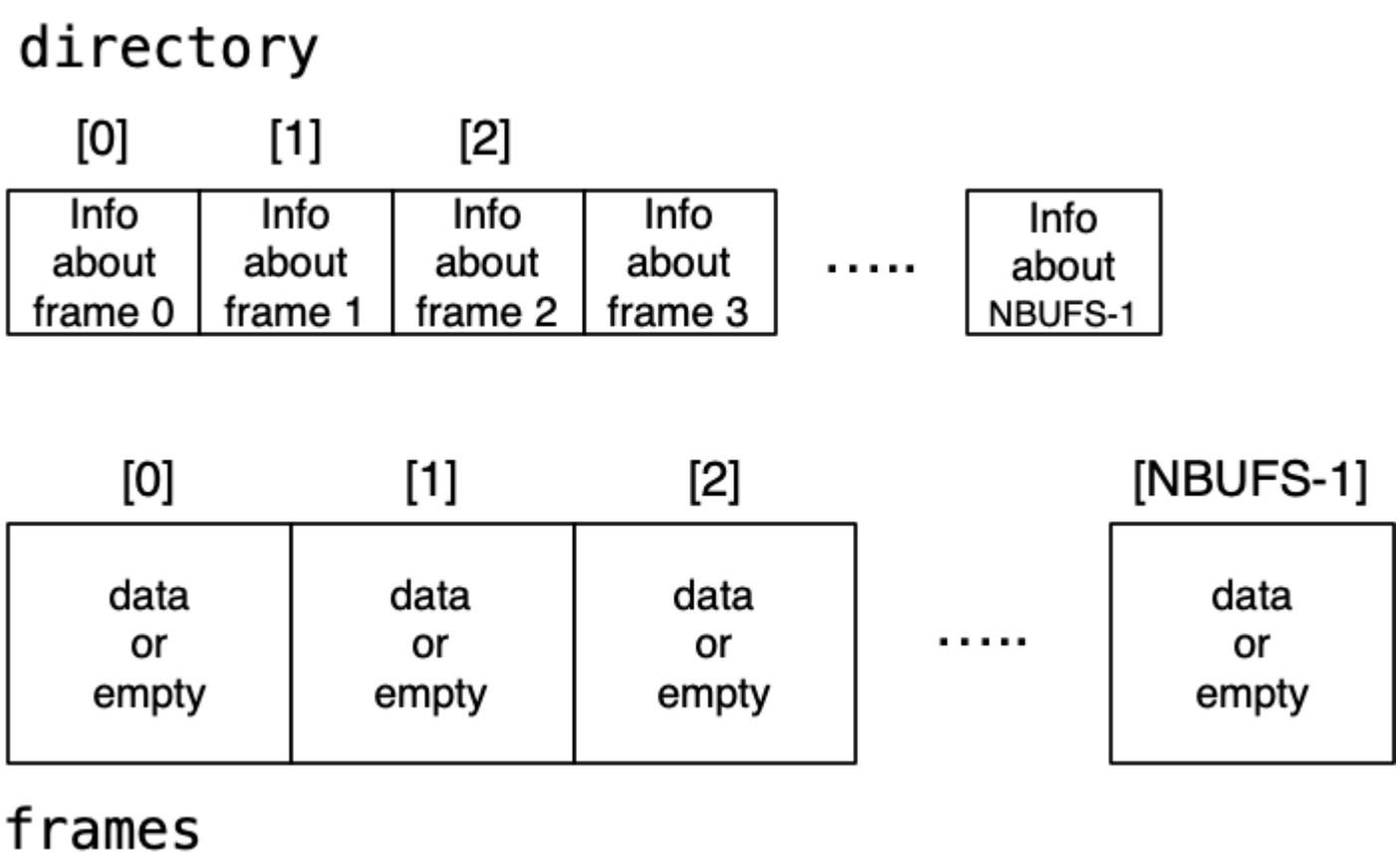
To some extent ...

- **`request_page()`** replaces **`getBlock()`**
- **`release_page()`** replaces **`putBlock()`**

Buffer pool data structures:

- **`Page frames[NBUFS]`**
- **`FrameData directory[NBUFS]`**
- **`Page`** is **`byte[BUFSIZE]`**

❖ Buffer Pool (cont)



❖ Buffer Pool (cont)

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)

Pages are referenced by PageID ...

- PageID = BufferTag = (rnode, forkNum, blockNum)

❖ Buffer Pool (cont)

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 (cont)

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 (cont)

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 (cont)

The **`release_page(pid)`** operation:

- Decrement pin count for specified page

Note: no effect on disk or buffer contents until replacement required

The **`mark_page(pid)`** operation:

- Set dirty bit on for specified page

Note: doesn't actually write to disk; indicates that page changed

The **`flush_page(pid)`** operation:

- Write the specified page to disk (using **`write_page`**)

Note: not generally used by higher levels of DBMS

❖ Buffer Pool (cont)

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 (cont)

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 (cont)

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);
}
```

❖ Effect of Buffer Management (cont)

Costs depend on relative size of tables, #buffers (n), replacement strategy

Requests: each rC page requested once, each rE page requested rC times

If nPages(rC)+nPages(rE) ≤ n

- read each page exactly once, holding all pages in buffer pool

If nPages(rE) ≤ n-1, and LRU replacement

- sequential flooding (see earlier slide)

If n == 2 (worst case)

- read each page every time it's requested

Produced: 31 Jan 2023

PostgreSQL Buffer Manager

❖ 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 (cont)

Buffer pool consists of:

`BufferDescriptors`

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

`BufferBlocks`

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

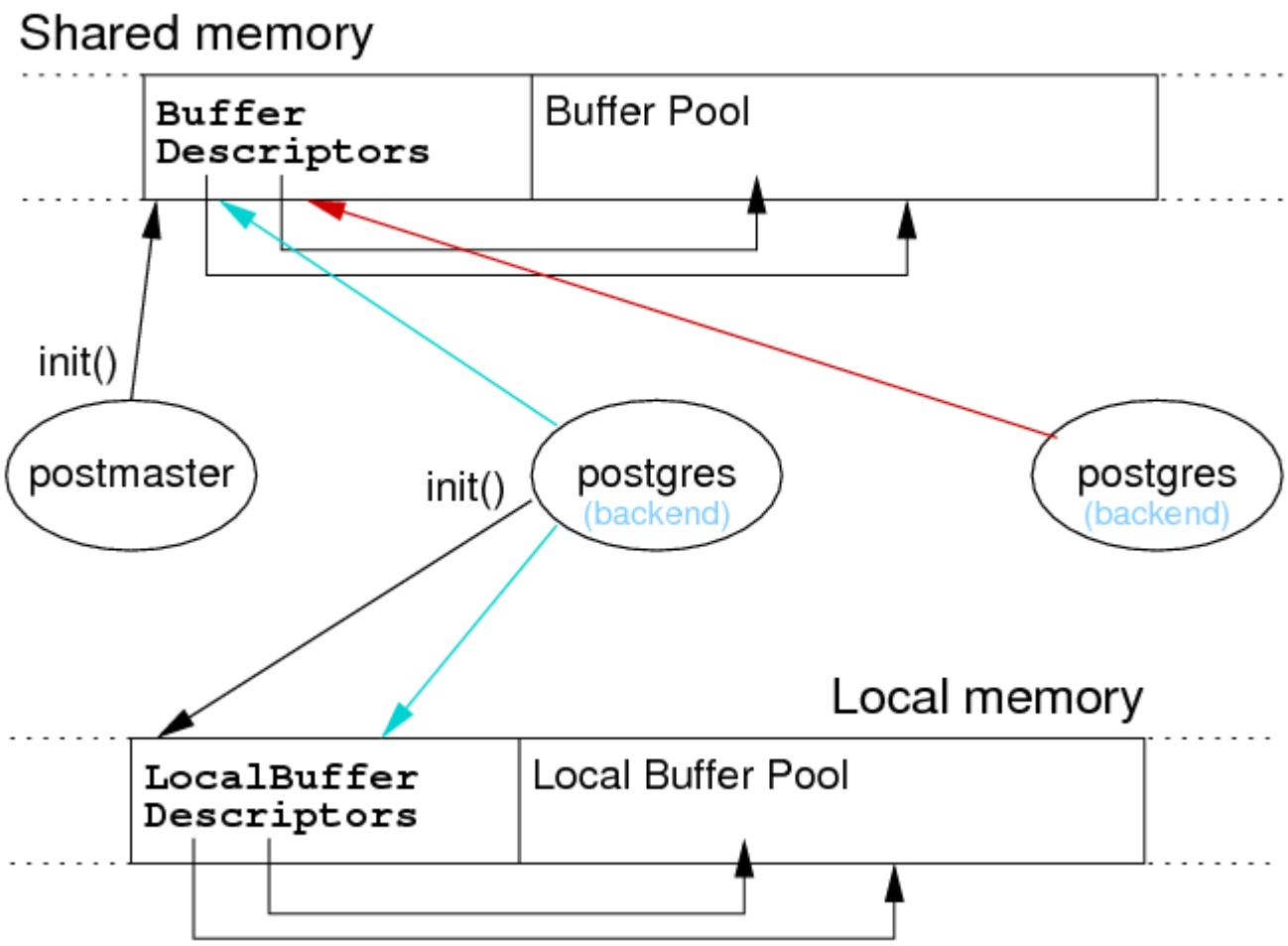
`Buffer` = index values in above arrays

- 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 (cont)



❖ PostgreSQL Buffer Manager (cont)

`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`**

❖ PostgreSQL Buffer Manager (cont)

Definition of buffer descriptors (simplified):

```
include/storage/buf_internals.h
typedef struct BufferDesc
{
    BufferTag  tag;      // ID of page contained in buffer
    int        buf_id;   // buffer's index number (from 0)

    // state, containing flags, refcount and usagecount
    pg_atomic_uint32 state;

    int        freeNext;  // link in freelist chain
    ...
} BufferDesc;
```

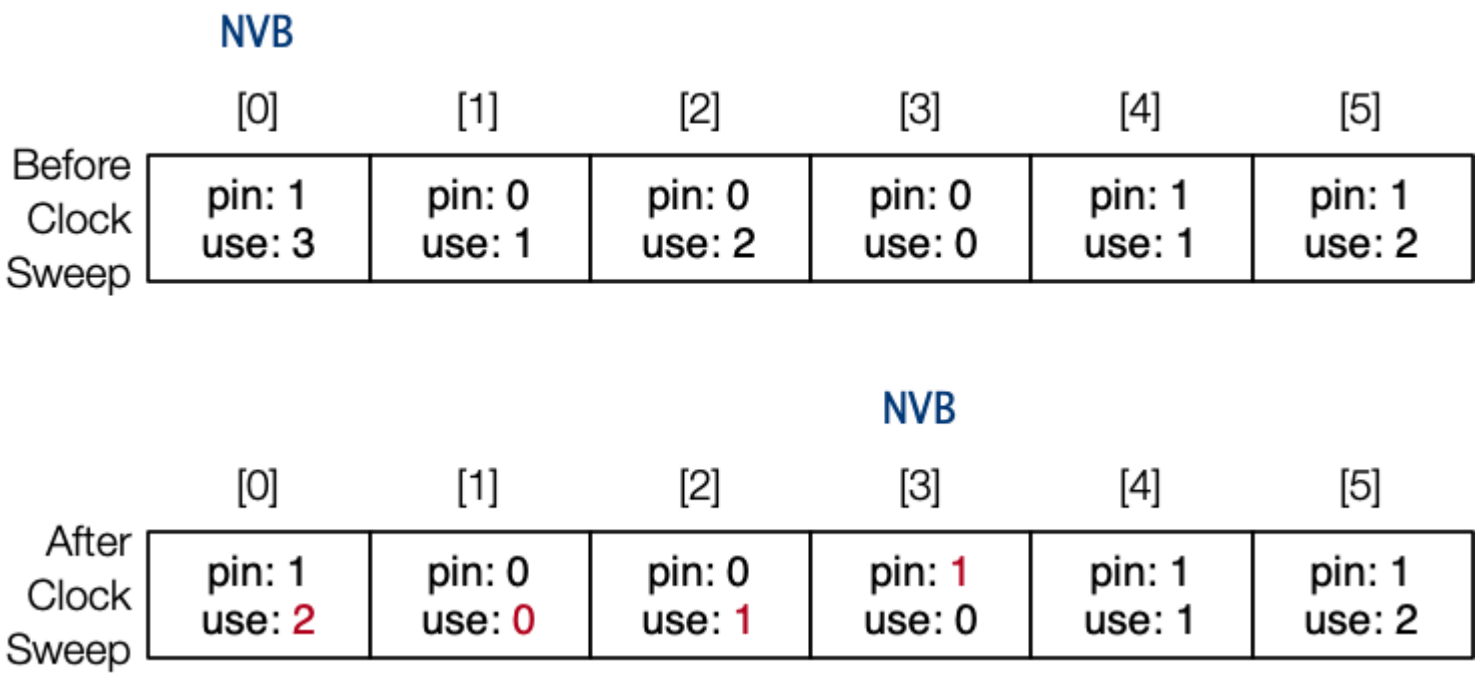
❖ Clock-sweep Replacement Strategy

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)

❖ Clock-sweep Replacement Strategy (cont)

Action of clock-sweep:



❖ Clock-sweep Replacement Strategy (cont)

For specialised kinds of access (e.g. sequential scan),

- clock-sweep is not the best replacement strategy
- can allocate a private "buffer ring"
- use this buffer ring with alternative replacement strategy

Produced: 15 Feb 2023

Page Internals

- Pages
- Page Formats
- Page Formats
- Storage Utilisation
- Overflows

❖ Pages

Database applications view data as:

- a collection of records (tuples)
- records can be accessed via a **`TupleId`**/**`RecordId`**/**`RID`**
- **`TupleId`** = (**`PageID`** + **`TupIndex`**)

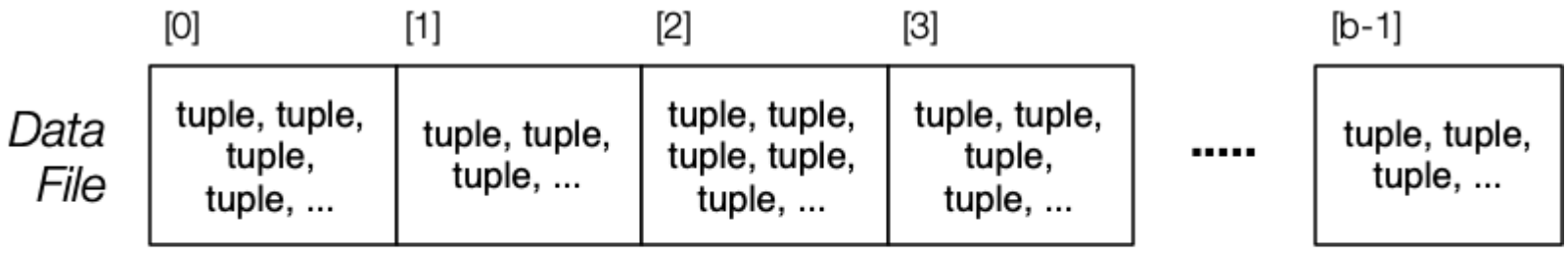
The disk and buffer manager provide the following view:

- data is a sequence of fixed-size pages (aka "blocks")
- pages can be (random) accessed via a **`PageID`**
- each page contains zero or more tuple values

Page format = how space/tuples are organised within a page

❖ Pages (cont)

Data files consist of pages containing tuples:



__r__ tuples contained in __b__ pages

each page can hold up to __c__ tuples

Each data file (in PostgreSQL) is related to one table.

❖ Page Formats

Ultimately, a **`Page`** is simply an array of bytes (**`byte[]`**).

We want to interpret/manipulate it as a collection of **`Record`**s (tuples).

Tuples are addressed by a record ID (**`rid`** = **`(PageId,TupIndex)`**)

Typical operations on **`Page`**s:

- **`request_page(pid)`** ... get page via its **`PageId`**
- **`get_record(rid)`** ... get record via its **`TupleId`**
- **`rid = insert_record(pid,rec)`** ... add new record
- **`update_record(rid,rec)`** ... update value of record
- **`delete_record(rid)`** ... remove record from page

❖ Page Formats (cont)

Page format = tuples + data structures allowing tuples to be found

Characteristics of **`Page`** formats:

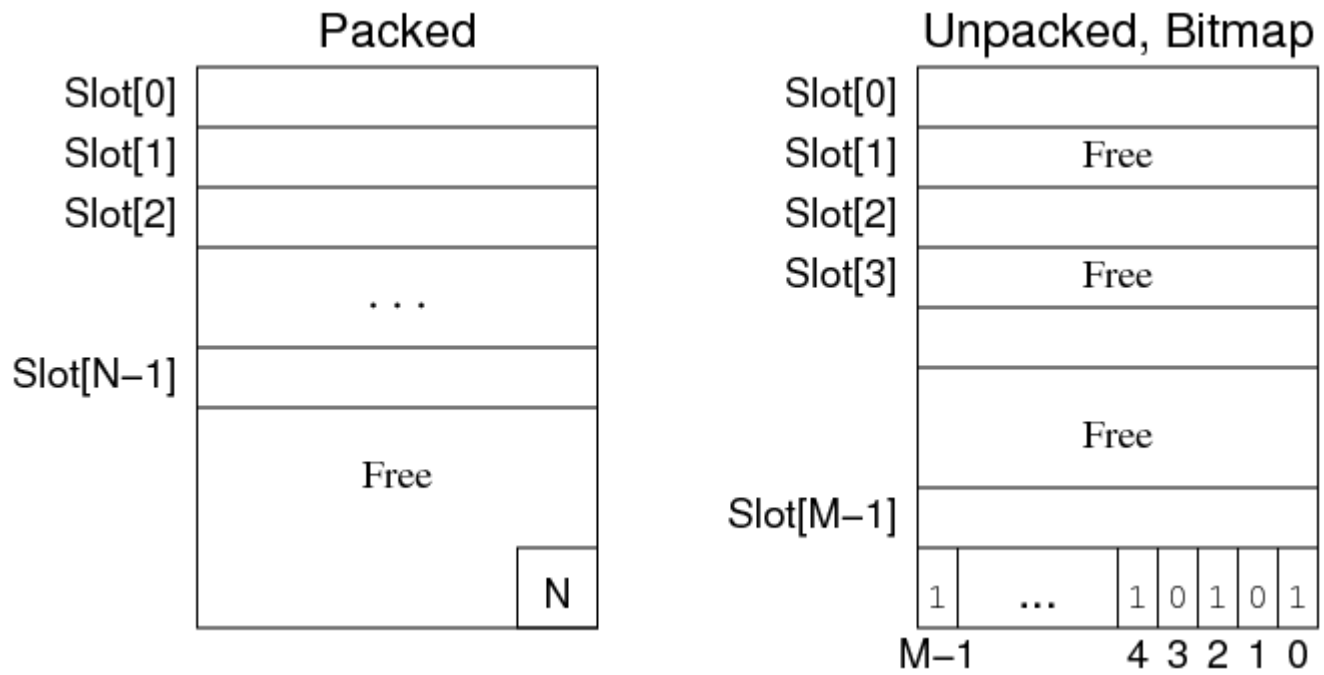
- record size variability (fixed, variable)
- how free space within **`Page`** is managed
- whether some data is stored outside **`Page`**
  - does **`Page`** have an associated overflow chain?
  - are large data values stored elsewhere? (e.g. TOAST)
  - can one tuple span multiple **`Page`**s?

Implementation of **`Page`** operations critically depends on format.

❖ Page Formats (cont)

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.

Possibilities for handling free-space within block:

- compacted (one region of free space)
- fragmented (distributed free space)

In practice, a combination is useful:

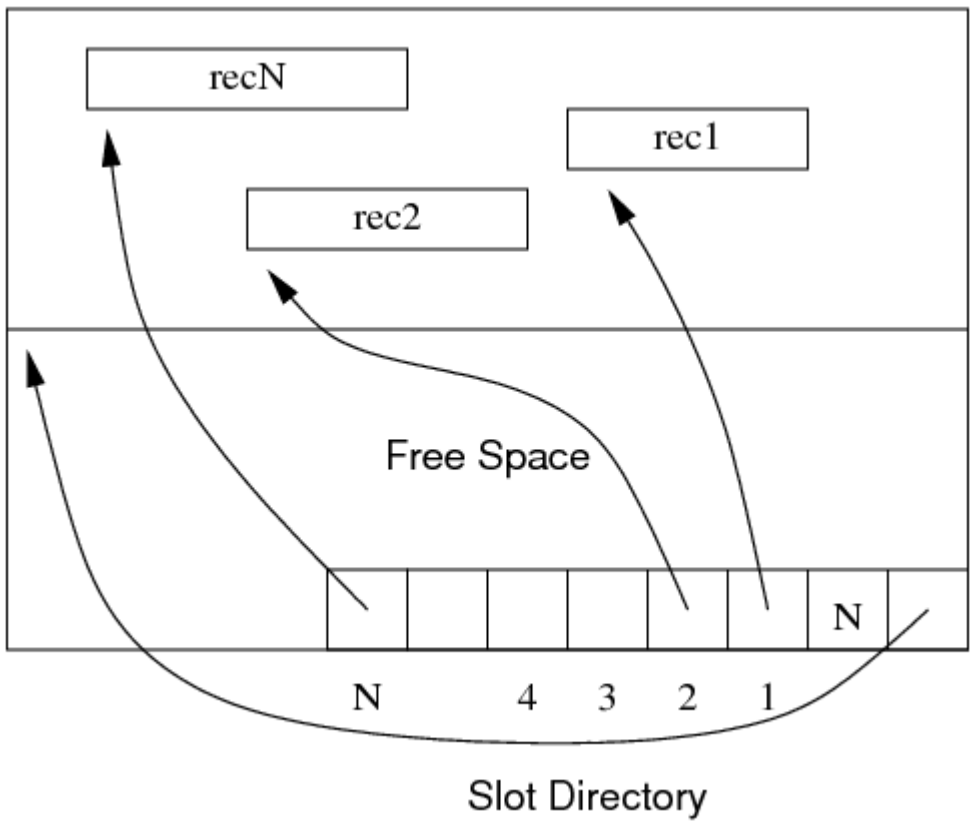
- normally fragmented (cheap to maintain)
- compacted when needed (e.g. record won't fit)

Important aspect of using slot directory

- location of tuple within page can change, tuple index does not change

❖ Page Formats (cont)

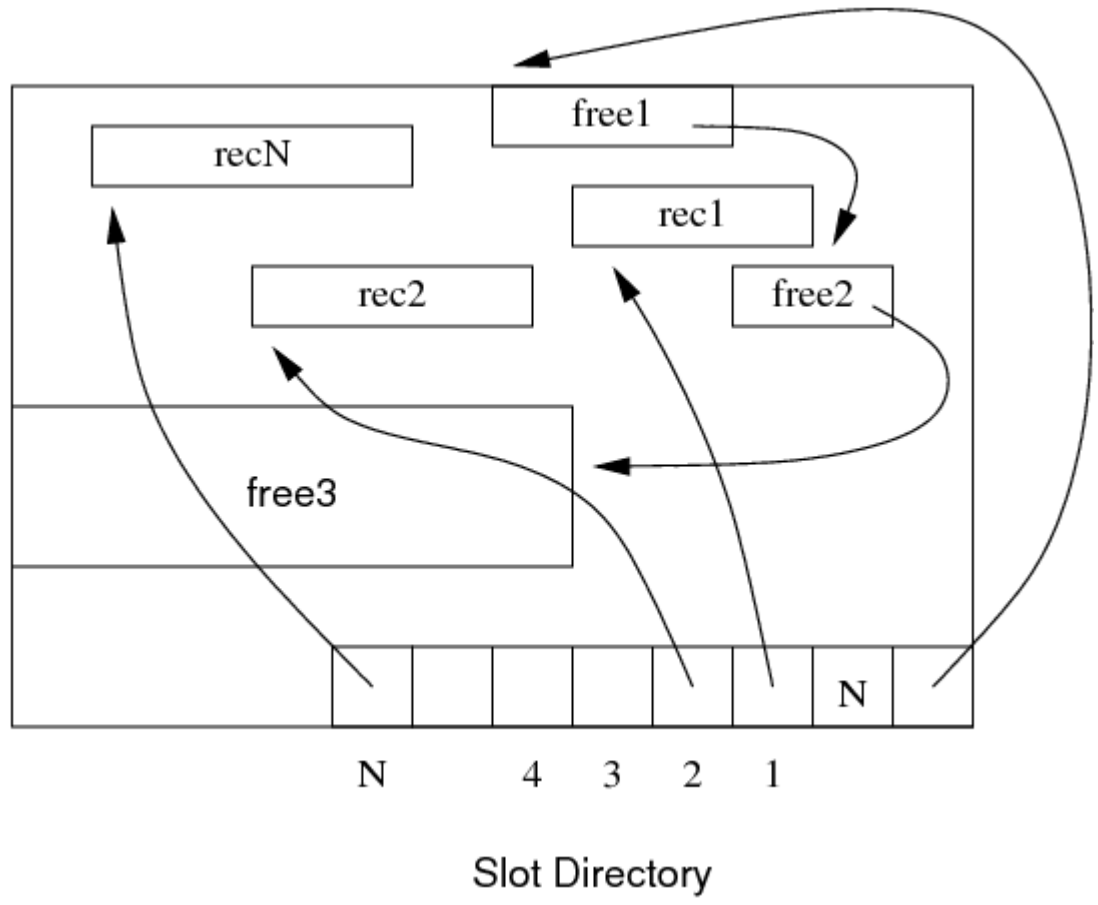
Compacted free space:



Note: "pointers" are implemented as word offsets within block.

❖ Page Formats (cont)

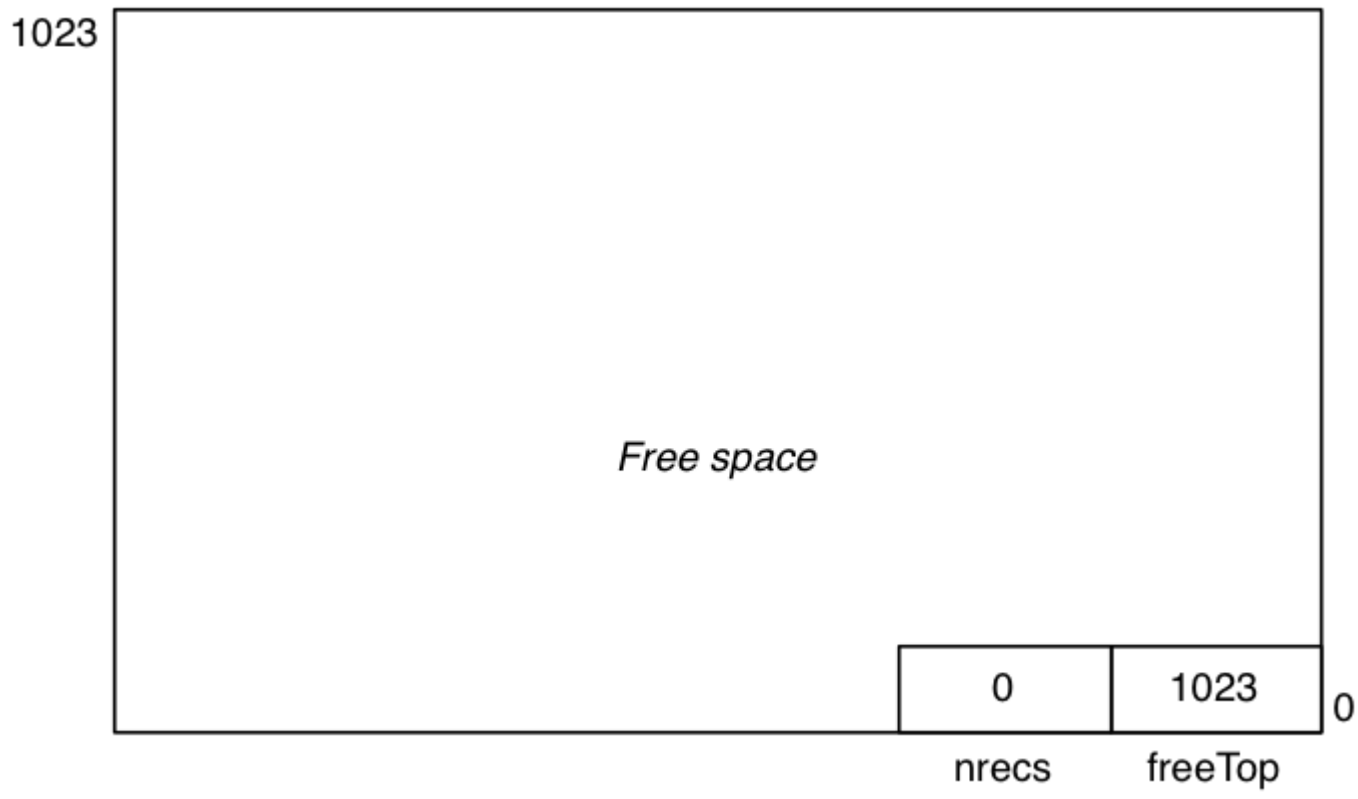
Fragmented free space:



Slot Directory

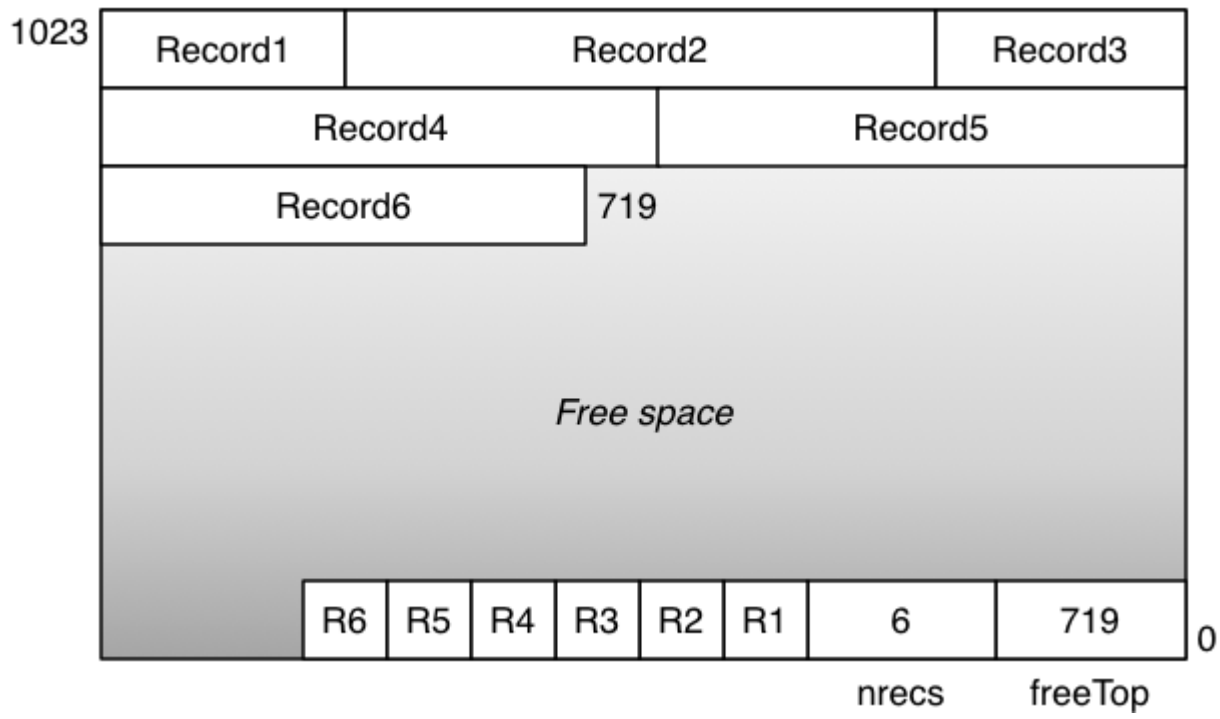
❖ Page Formats (cont)

Initial page state (compacted free space) ...



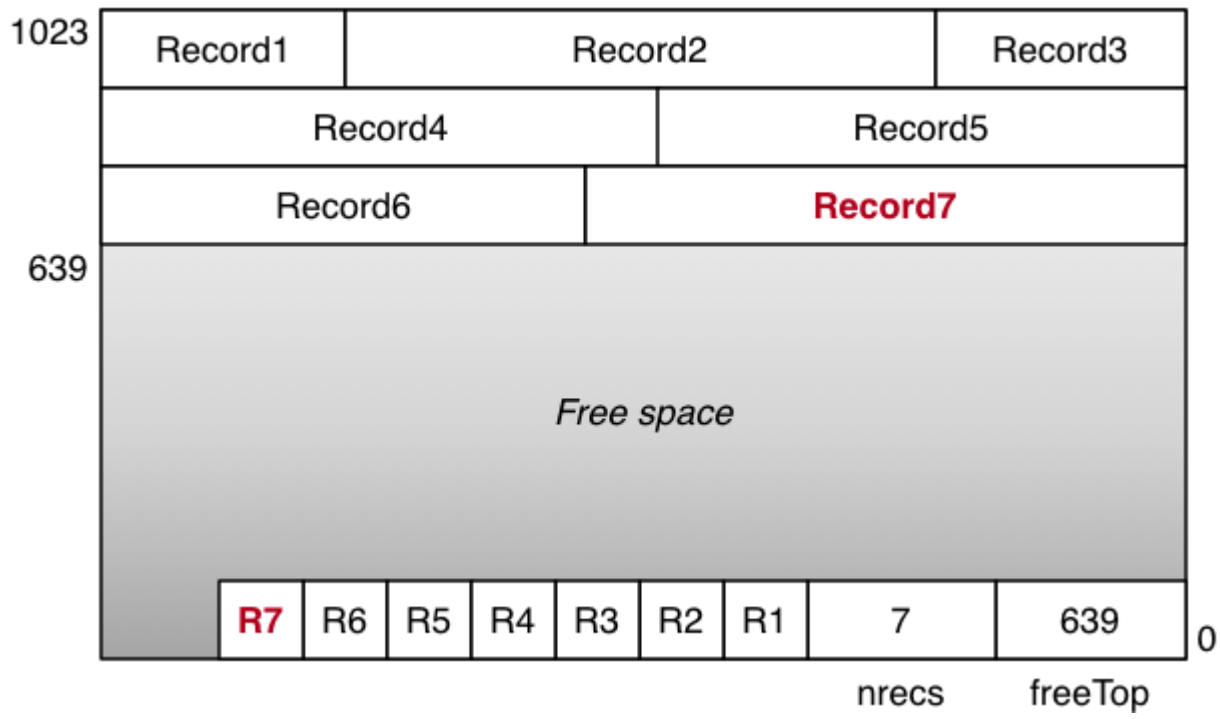
❖ Page Formats (cont)

Before inserting record 7 (compacted free space) ...



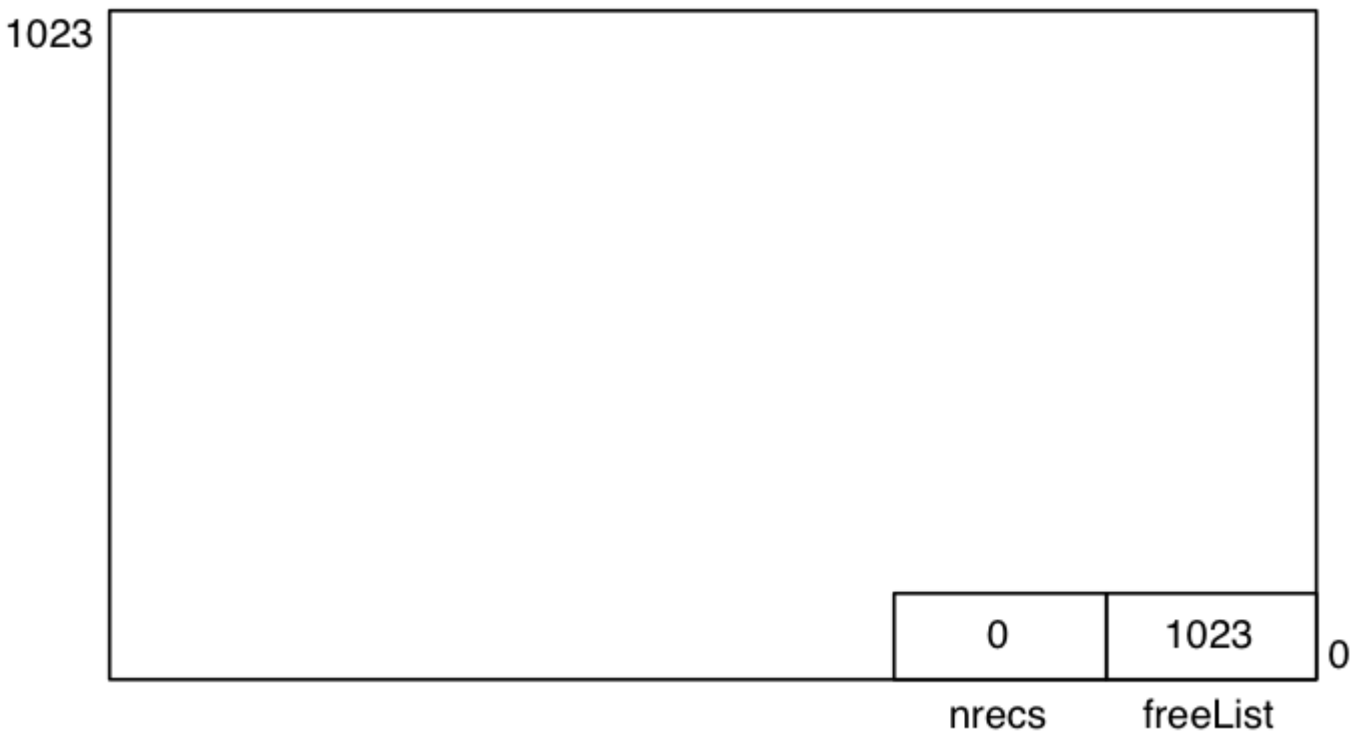
❖ Page Formats (cont)

After inserting record 7 (80 bytes) ...



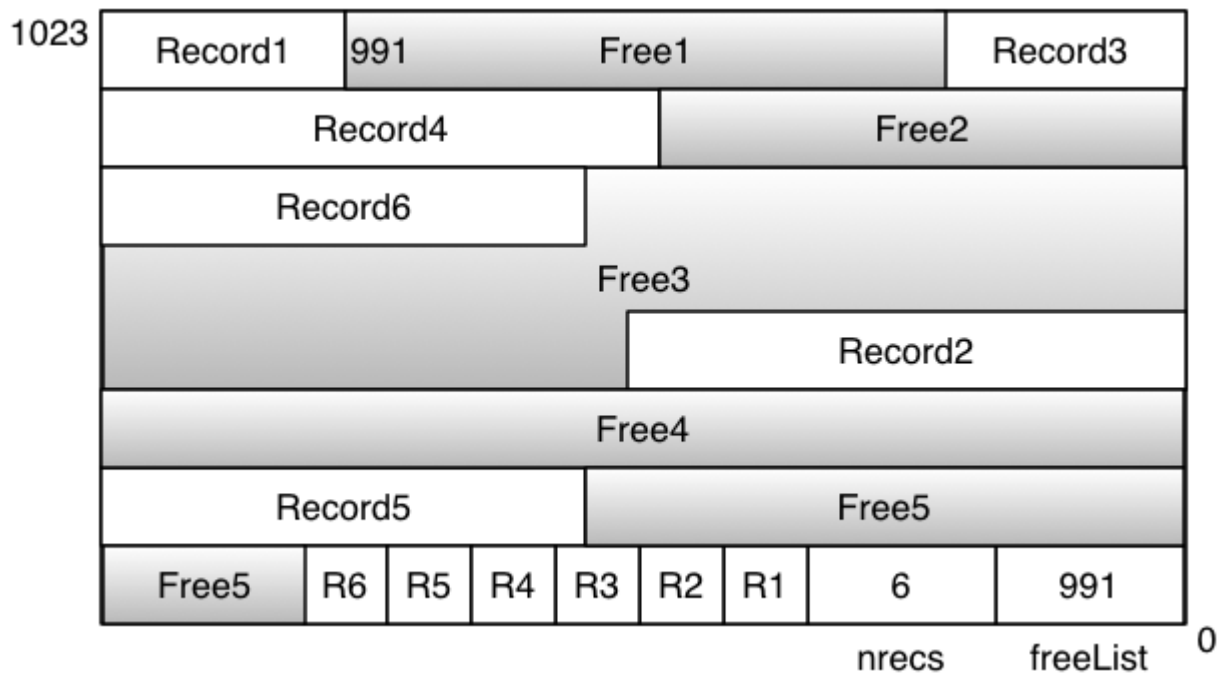
❖ Page Formats (cont)

Initial page state (fragmented free space) ...



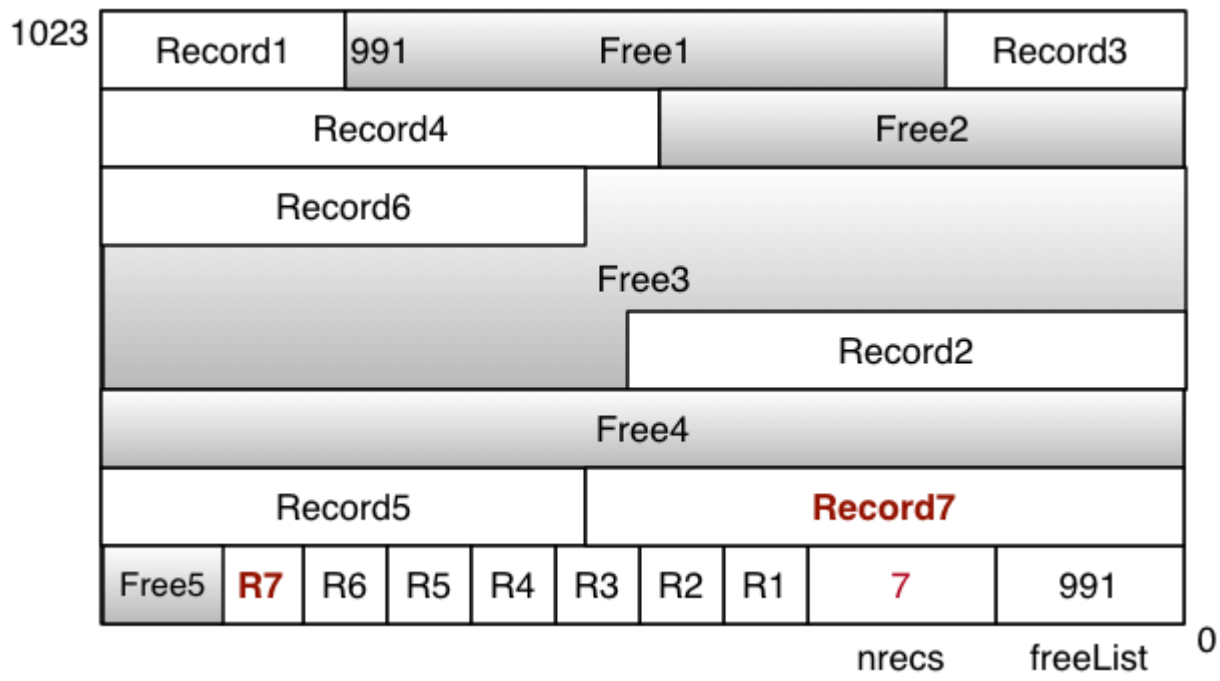
❖ Page Formats (cont)

Before inserting record 7 (fragmented free space) ...



❖ Page Formats (cont)

After inserting record 7 (80 bytes) ...



❖ Storage Utilisation

How many records can fit in a page? (denoted *C* = capacity)

Depends on:

- page size ... typical values: 1KB, 2KB, 4KB, 8KB
- record size ... typical values: 64B, 200B, app-dependent
- page header data ... typically: 4B - 32B
- slot directory ... depends on how many records

We typically consider *average* record size (*R*)

Given *C*, $HeaderSize + C*SlotSize + C*R \leq PageSize$

❖ Overflows

Sometimes, it may not be possible to insert a record into a page:

1. no free-space fragment large enough
2. overall free-space is not large enough
3. the record is larger than the page
4. no more free directory slots in page

For case (1), can first try to compact free-space within the page.

If still insufficient space, we need an alternative solution ...

❖ Overflows (cont)

File organisation determines how cases (2)..(4) are handled.

If records may be inserted anywhere that there is free space

- cases (2) and (4) can be handled by making a new page
- case (3) requires either spanned records or "overflow file"

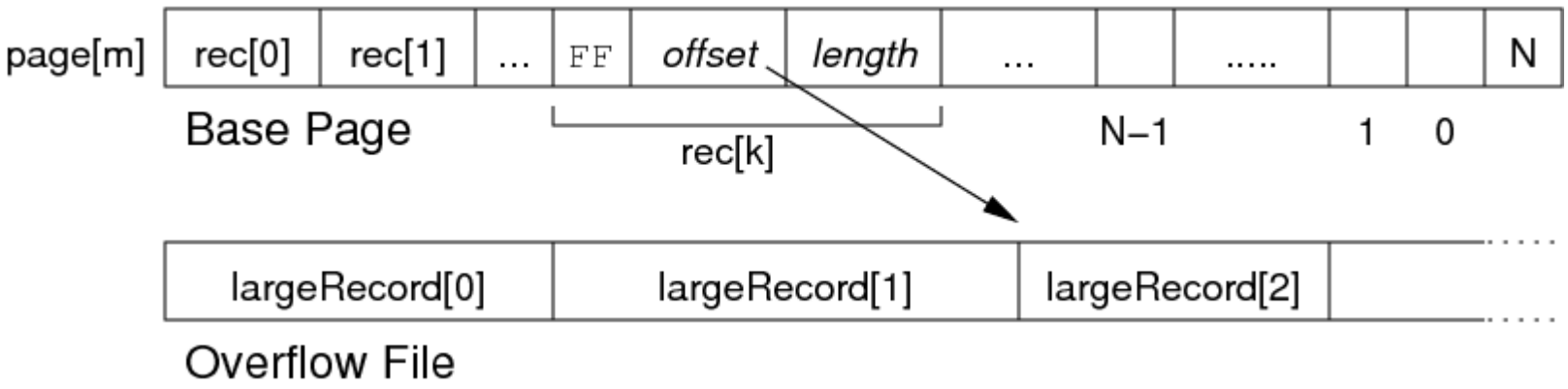
If file organisation determines record placement (e.g. hashed file)

- cases (2) and (4) require an "overflow page"
- case (3) requires an "overflow file"

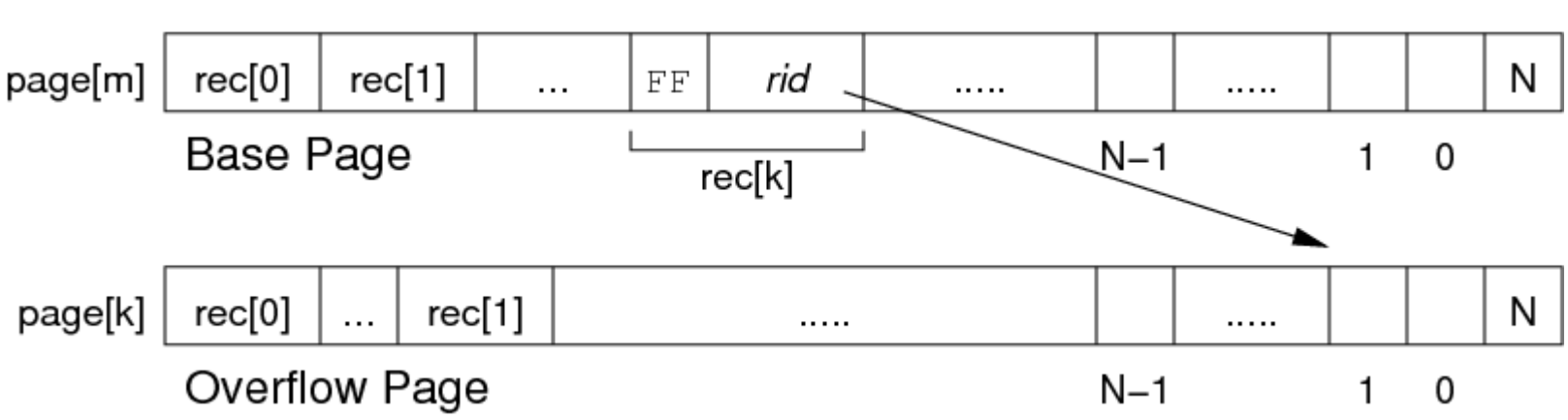
With overflow pages, *rid* structure may need modifying *(rel,page,ovfl,rec)*

❖ Overflows (cont)

Overflow files for very large records and BLOBs:



Record-based handling of overflows:



We discuss overflow pages in more detail when covering Hash Files.

Produced: 31 Jan 2023

PostgreSQL Page Internals

❖ PostgreSQL Page Representation

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

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

Each page is 8KB (default **`BLCKSZ`**) and contains:

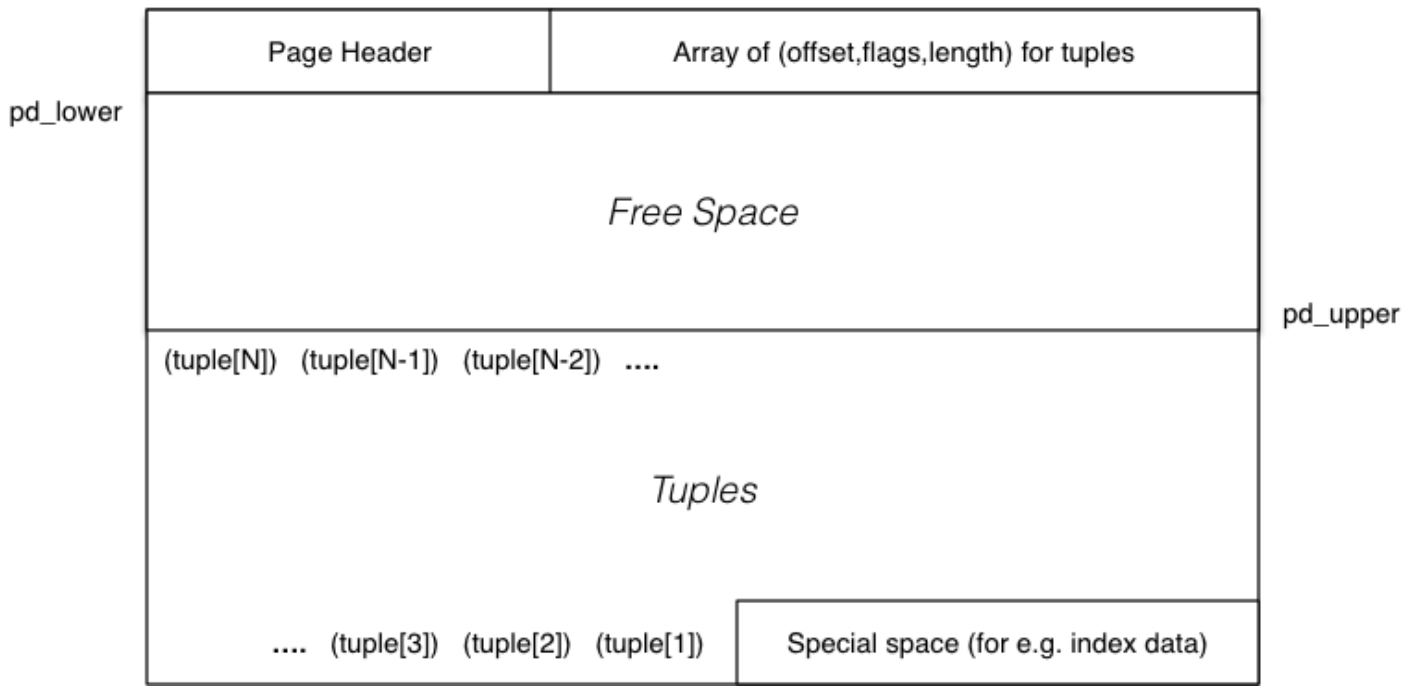
- header (free space pointers, flags, xact data)
- array of (offset,length) pairs for tuples in page
- free space region (between array and tuple data)
- actual tuples themselves (inserted from end towards start)
- (optionally) region for special data (e.g. index data)

Large data items are stored in separate (TOAST) files (implicit)

Also supports ~SQL-standard BLOBs (explicit large data items)

❖ PostgreSQL Page Representation (cont)

PostgreSQL page layout:



❖ PostgreSQL Page Representation (cont)

Page-related data types:

```
// a Page is simply a pointer to start of buffer
typedef Pointer Page;

// indexes into the tuple directory
typedef uint16  LocationIndex;

// entries in tuple directory (line pointer array)
typedef struct ItemIdData
{
   unsigned   lp_off:15,    // tuple offset from start of page
              lp_flags:2,   // unused,normal,redirect,dead
              lp_len:15;    // length of tuple (bytes)
} ItemIdData;
```

❖ PostgreSQL Page Representation (cont)

Page-related data types: (cont)

```
include/storage/bufpage.h
typedef struct PageHeaderData
{
    XLogRecPtr    pd_lsn;      // xact log record for last change
    uint16        pd_tli;      // xact log reference information
    uint16        pd_flags;    // flag bits (e.g. free, full, ...
    LocationIndex pd_lower;    // offset to start of free space
    LocationIndex pd_upper;    // offset to end of free space
    LocationIndex pd_special;  // offset to start of special space
    uint16        pd_pagesize_version;
    TransactionId pd_prune_xid;// is pruning useful in data page?
    ItemIdData    pd_linp[];  // beginning of line pointer array, FLEXIBLE_ARRAY_MEMBER
} PageHeaderData;

typedef PageHeaderData *PageHeader;
```

❖ PostgreSQL Page Representation (cont)

Operations on **`Page`**s:

```
void PageInit(Page page, Size pageSize, ...)
```

- initialize a **`Page`** buffer to empty page
- in particular, sets **`pd_lower`** and **`pd_upper`**

```
OffsetNumber PageAddItem(Page page,
                         Item item, Size size, ...)
```

- insert one tuple (or index entry) into a **`Page`**
- fails if: not enough free space, too many tuples

```
void PageRepairFragmentation(Page page)
```

- compact tuple storage to give one large free space region

❖ PostgreSQL Page Representation (cont)

PostgreSQL has two kinds of pages:

- heap pages which contain tuples
- index pages which contain index entries

Both kinds of page have the same page layout.

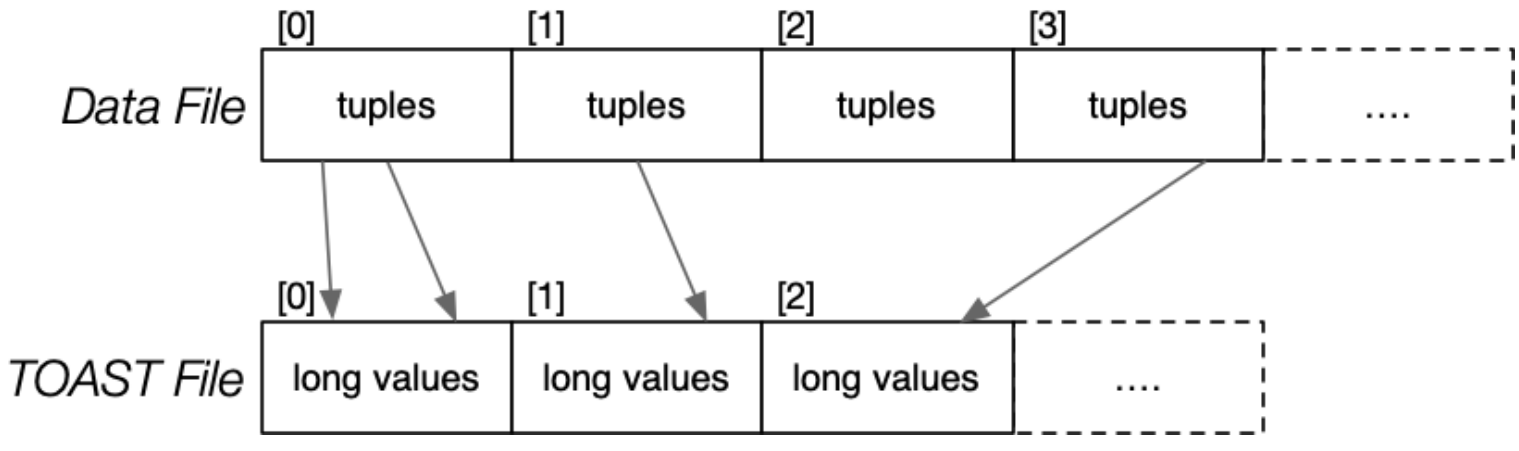
One important difference:

- index entries tend be a smaller than tuples
- can typically fit more index entries per page

❖ TOAST'ing

TOAST = The Oversized-Attribute Storage Technique

- handles storage of large attribute values (> 2KB) (e.g. long **`text`**)



❖ TOAST'ing (cont)

Large attribute values are stored out-of-line (i.e. in separate file)

- "value" of attribute in tuple is a reference to TOAST data
- TOAST'd values may be compressed
- TOAST'd values are stored in 2K chunks

Strategies for storing TOAST-able columns ...

- **`PLAIN`** ... allows no compression or out-of-line storage
- **`EXTENDED`** ... allows both compression and out-of-line storage
- **`EXTERNAL`** ... allows out-of-line storage but not compression
- **`MAIN`** ... allows compression but not out-of-line storage

Produced: 20 Feb 2023

Week 2 Exercises

❖ Exercise 1: 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?

❖ Exercise 2: Relation Scan Cost

Consider a table *R(x,y,z)* with 10^4 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.

❖ Exercise 3: 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 3: 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

❖ Exercise 4 : 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)

❖ Exercise 5: 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

❖ 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 6: 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?

❖ Exercise 7: 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 8: 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 9: 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.

Produced: 19 Feb 2023