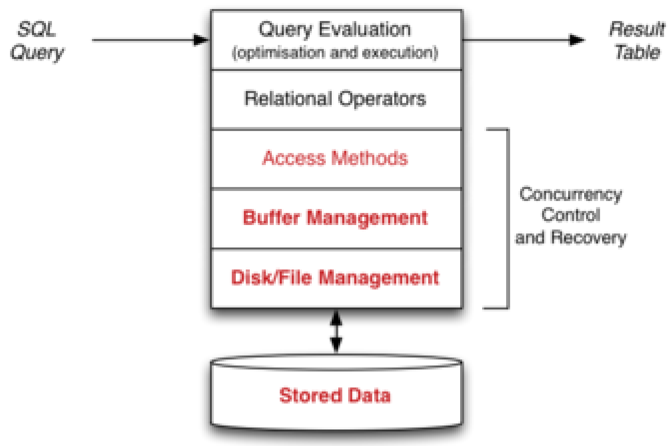
Week 02 Lectures

Storage Manager

Storage Management

2/100

Levels of DBMS related to storage management:



... Storage Management

3/100

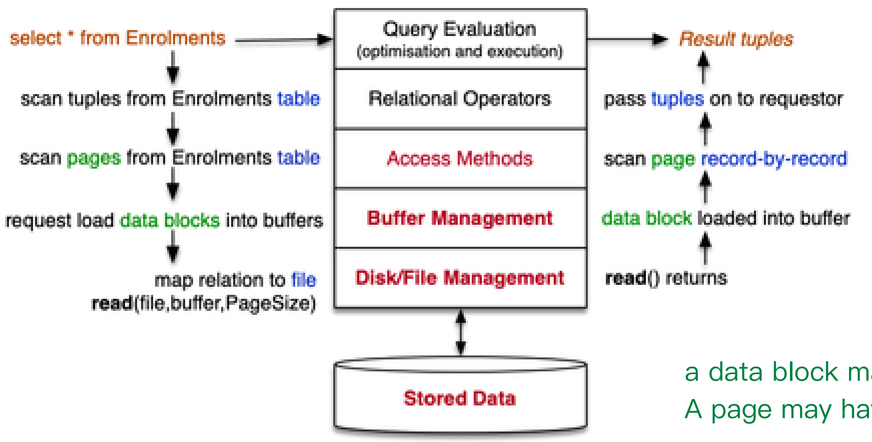
Aims of storage management in DBMS:

a sequence of bytes

- map from database objects (e.g. tables) to disk files
- manage transfer of data to/from disk storage
- use buffers to minimise disk/memory transfers
- interpret loaded data as tuples/records
- provide view of data as collection of pages/tuples
- basis for "file structures" used by access methods

Views of Data in Query Evaluation

4/100



a data block may have multiple page
A page may have multiple tuple

... Views of Data in Query Evaluation

5/100

Representing database objects during query execution:

- `DB` (handle on an authorised/opened database)
- `Rel` (handle on an opened relation)
- `Page` (memory buffer to hold contents of disk block)
- `Tuple` (memory holding data values from one tuple)

Addressing in DBMSs:

- `PageID = FileID+Offset` ... identifies a block of data
  - where `Offset` gives location of block within file
- `TupleID = PageID+Index` ... identifies a single tuple
  - where `Index` gives access to location of tuple within page

Storage Management

6/100

Topics in storage management ...

From a page ID, we can get the tuple ID and further get a file ID

- Disks and Files
  - performance issues and organisation of disk files
- Buffer Management
  - using caching to improve DBMS system throughput
- Tuple/Page Management
  - how tuples are represented within disk pages
- DB Object Management (Catalog)
  - how tables/views/functions/types, etc. are represented

Storage Technology

Storage Technology

8/100

A db has multiple table(relations)
A table(relation) has multiple pages
A pages has multiple tuple

Persistent storage is

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

Computational storage is

Data has to be stored in memory`

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

Access cost HDD:RAM ≅ 100000:1, e.g.

Hard Disk Drive Random Access Memory

- 10ms to read block containing two tuples
- 1μs to compare fields in two tuples

... Storage Technology

9/100

Hard disks are well-established, cheap, high-volume, ...

Alternative bulk storage: SSD

Solid-state disk

- faster than HDDs, no latency
- can read single items
- update requires block erase then write
- over time, writes "wear out" blocks
- require controllers that spread write load

Feasible for long-term, high-update environments?

... Storage Technology

10/100

Comparison of HDD and SSD properties:

	HDD	SDD
Cost/byte	~ 2c / GB	~ 13c / GB
Read latency	~ 10ms	~ 50μs

Write latency	~ 10ms	~ 900μs
Read unit	block (e.g. 1KB)	byte
Writing	write a block	write on empty block

Will SSDs ever replace HDDs for large-scale database storage?

Cost Models

11/100

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

12/100

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

13/100

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 the file descriptor to a certain point of a file
  // move file pointer to relative/absolute file offset
fsync(fd)
  // flush contents of file buffers to disk
```

synchronise the data from buffer to disk

DBMS File Organisation

14/100

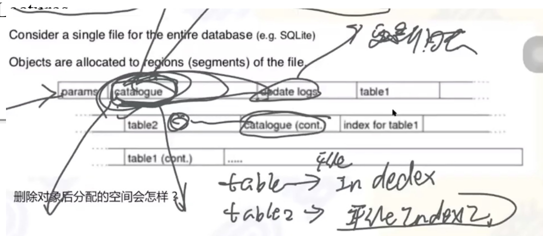
How is data for DB objects arranged in the file system?

Different DBMSs make different choices, e.g.

When we need to manage a large data base, it is unsuitable to put it in a single file. Then the first step is to create multiple file and index them. Like the '001 —> the 1st ~ 500 milions th record' Then when we get a tupleID, we can go for the corresponding record

- 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

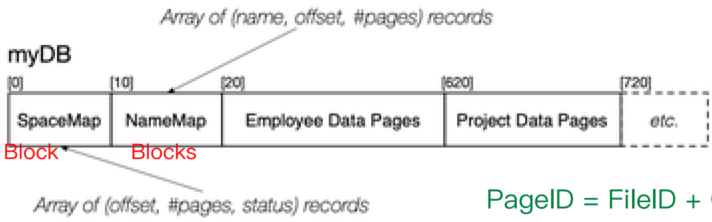
- have multiple files for each table
- etc.



Single-file Storage Manager

15/100

Consider the following simple single-file DBMS layout:



PageID = FileID + Offset

TupleID = PageID + Index

E.g.

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

NameMap = [("employee",20,350), ("project",620,40)]

... Single-file Storage Manager

16/100

Each file segment consists of a number fixed-size blocks

The following data/constant definitions are useful

```
#define PAGESIZE 4096   // 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

17/100

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
   NameMap 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

18/100

With the above disk manager, our example:

```
select name from Employee
```

might be implemented as something like

```
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 {
```

Put the page into the buffer and modify it in the 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 = readSpaceMap(db->fd);
   db->names = readNameMap(db->fd);
   return db;
}
// 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;
}
```

... Single-File Storage Manager


```
// 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);
}
```

Exercise 1: 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 $T_r = 10msec$

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.

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

23/100

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	-	$\cong 0$

Multiple-file Disk Manager

24/100

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 meta-data)

Precise file structure varies between individual DBMSs.

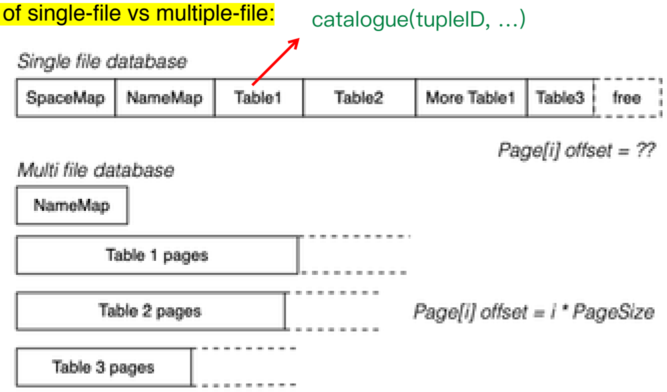
Using multiple files (one file per relation) can be easier, e.g.

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

... Multiple-file Disk Manager

25/100

Example of single-file vs multiple-file:



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

... Multiple-file Disk Manager

26/100

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

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

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

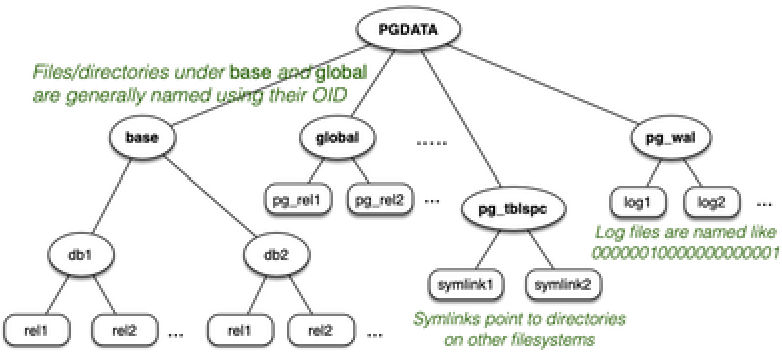
If system uses several files per table, `PageId` contains:

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

PostgreSQL Storage Manager

27/100

PostgreSQL uses the following file organisation ...



... PostgreSQL Storage Manager

28/100

Components of storage subsystem:

Actually, there are more than two kinds of files(data file, index file)

- 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

29/100

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

30/100

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;
}
```

Exercise 2: PostgreSQL Files

31/100

In your PostgreSQL server

- examine the content of the `$PGDATA` directory
- find the directory containing the `pizza` database
- 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 databases?

File Descriptor Pool

32/100

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".

Defs: **`include/storage/fd.h`**
Code: **`backend/storage/file/fd.c`**

... File Descriptor Pool

33/100

Interface to file descriptor (pool):

```
File PathNameOpenFile(char *fileName, int flags)
      // open a file with default pg.conf mode
File PathNameOpenFilePerm(char *fName, int flags, int mode)
     // open a file in the DB 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()`, ...

... File Descriptor Pool

34/100

Virtual file descriptor records (simplified):

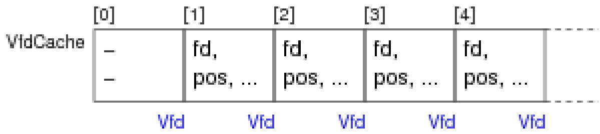
```
typedef struct vfd
{
    s_short  fd;              // current FD, or VFD_CLOSED if none
    u_short  fdstate;         // bitflags for Vfd's state
    File     nextFree;        // link to next free Vfd, if in freelist
    File     lruMoreRecently; // doubly linked recency-of-use list
    File     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 Descriptor Pool

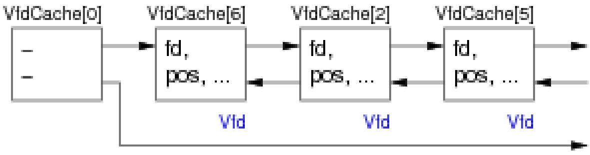
35/100

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.

Exercise 3: Opening a Vfd

36/100

Consider the following call to open a file

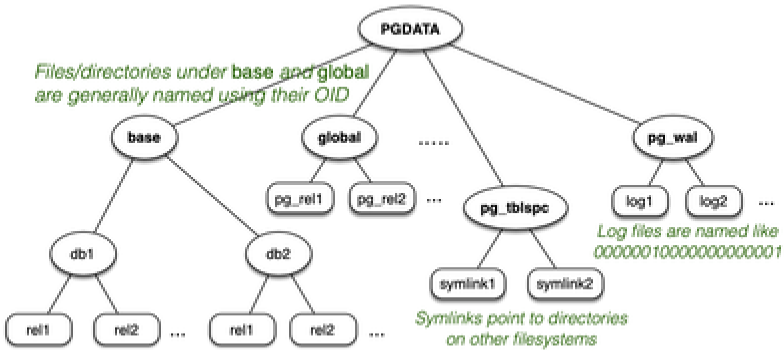
```
f = PathNameOpenFilePerm(
   "/srvr/jas/pgsql/data/base/13645/12348",
   O_RDWR | O_CREAT | O_EXCL | PG_BINARY,   all bitwise
   0600
)
```

Sketch implementation of `PathNameOpenFilePerm()`

File Manager

37/100

Reminder: PostgreSQL file organisation

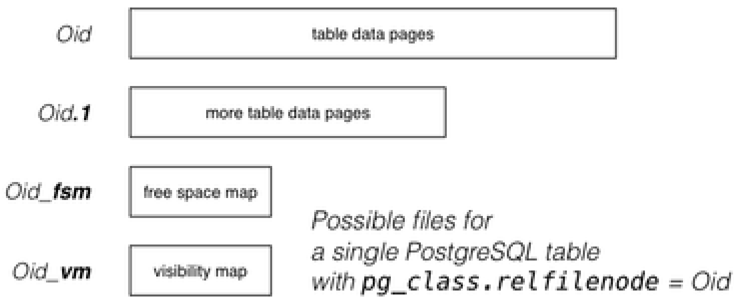


... File Manager

38/100

PostgreSQL stores each table

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



... File Manager

39/100

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 40/100

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 41/100

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 42/100

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)
{
   File fid;  off_t offset;  int fd;
   (fid, offset) = findBlock(pageID)
   fd = VfdCache[fid].fd;
   lseek(fd, offset, SEEK_SET)
   VfdCache[fid].seekPos = offset;
   nread = read(fd, buf, BLOCKSIZE)
   if (nread < BLOCKSIZE) ... we have a problem
}
```

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

... File Manager 43/100

```
findBlock(BufferTag pageID) returns (Vfd, off_t)
{
   offset = pageID.blockNum * BLOCKSIZE
   fileName = relpath(pageID.rnode)
   if (pageID.forkNum > 0)
      fileName = fileName+"."+pageID.forkNum
   fid = PathNameOpenFIle(fileName, O_READ);
   fSize = VfdCache[fid].fileSize;
   if (offset > fSize) {
      fid = allocate new Vfd for next fork
      offset = offset - fd.fileSize
   }
   return (fd, offset)
}
```

Buffer Pool

To process multiple query, decrease the communication with db, boost the efficiency

Buffer Pool

45/100

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

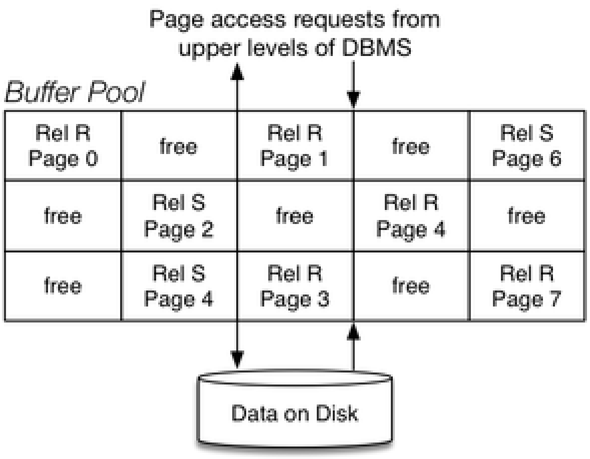
Uses:

- file manager functions to access data files

Note: we use the terms *page* and *block* interchangably

... Buffer Pool

46/100



... Buffer Pool

47/100

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

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

To some extent ...

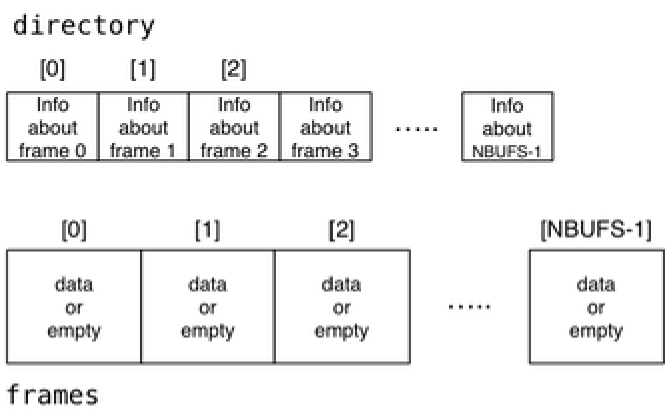
- `request_page()` replaces `getBlock()` or `get_page()`
- `release_page()` replaces `putBlock()` or `put_page()`

Buffer pool data structures:

- frames ... array of `NBUFS` Page buffers
- directory ... array of `NBUFS` FrameData items

... Buffer Pool

48/100



... 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)

Pages are referenced by PageID ...

Simply means init a new data file

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

Basically means a new data file

... 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.

Read data from buffer is much faster than disk

... 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

Buffer pool data structures:

```
typedef char Page[PAGESIZE];
typedef ... PageID;  // defined earlier

typedef struct _FrameData {
   PageID  pid;        // which page is in frame
   int     pin_count;  // how many processes using page
   int     dirty;      // page modified since loaded?
   Time    last_used;  // when page was last accessed
} FrameData;

Page frames[NBUFS];     // actual buffers
FrameData directory[NBUFS];
```

... Buffer Pool

Implementation of `request_page()`

```
int request_page(PageID pid)
{
   bufID = findInPool(pid)
```

```
    if (pid == NOT_FOUND) {
       if (no free frames in Pool) {
          bufID = findFrameToReplace()
          if (directory[bufID].dirty)
             old = directory[bufID].page
             put_page(old, frames[bufID])
       }
       bufID = index of freed frame
       directory[bufID].page = pid
       directory[bufID].pin_count = 0
       directory[bufID].dirty = 0
       get_page(pid, frames[bufID])
    }
    directory[bufID].pin_count++
    return bufID
}
```

if a buffer pool is filled up, then just simply enlarge the size

A directory correspond to `

... Buffer Pool

54/100

All of the displayed query go through the buffer pool

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()`)

Note: not generally used by higher levels of DBMS

... Buffer Pool

55/100

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

56/100

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

57/100

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

58/100

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

59/100

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 4: Buffer Cost Benefit (i)

60/100

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 ...

- if we have only 2 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 $n < b_E$)

Exercise 5: Buffer Cost Benefit (ii)

61/100

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)

default

PostgreSQL Buffer Manager

62/100

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`**

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

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

... PostgreSQL Buffer Manager

63/100

Buffer pool consists of:

`BufferDescriptors`

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

`BufferBlocks`

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

NBuffers * BufferDesc

`Buffer` = index in above arrays

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

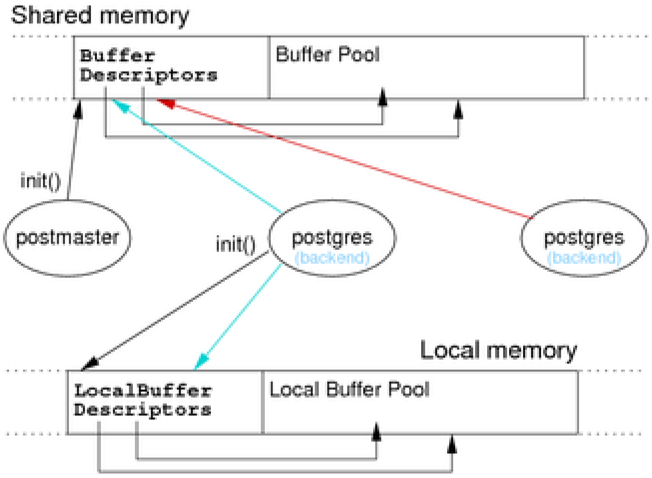
Size of buffer pool is set in *postgresql.conf*, e.g.

Processes have their own buffers

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

... PostgreSQL Buffer Manager

64/100



Buffer Pool Data Types

65/100

```
typedef struct buftag {
   RelFileNode rnode;     /* physical relation identifier */
   ForkNumber  forkNum;
   BlockNumber blockNum;  /* relative to start of reln */
} BufferTag;

typedef struct BufferDesc { (simplified)
   BufferTag tag;        // ID of page contained in buffer
   int       buf_id;     // buffer's index number (from 0)
   Bits32    state;      // dirty, refcount, usage
   int       freeNext;   // link in freelist chain
   ...                   // others related to concurrency
} BufferDesc;
```

the ID is Pointed to a certain page that is proccessed

Buffer Pool Functions

66/100

Buffer manager interface:

`Buffer ReadBuffer(Relation r, BlockNumber n)`

- ensures page *n* of file for relation `r` is loaded
- increments reference (pin) count and usage count for buffer
- returns index of loaded page in buffer pool (`Buffer` value)

```
BufferDesc *BufferAlloc(
           Relation r, ForkNumber f,
           BlockNumber n, bool *found)
```

- used by **`ReadBuffer`** to find a buffer for *(r,f,n)*
- if no available buffers, select buffer to be replaced

... Buffer Pool Functions

67/100

Buffer manager interface (cont):

`void ReleaseBuffer(Buffer buf)`

- decrement pin count on buffer
- if pin count falls to zero,
  ensures all activity on buffer is completed before returning

`void MarkBufferDirty(Buffer buf)`

- marks a buffer as modified
- requires that buffer is pinned and locked
- actual write is done later (e.g. when buffer replaced)

... Buffer Pool Functions

68/100

Additional buffer manager functions:

`Page BufferGetPage(Buffer buf)`

- finds actual data associated with buffer in pool
- returns reference to memory where data is located

`BufferIsPinned(Buffer buf)`

- check whether this backend holds a pin on buffer

`CheckPointBuffers`

- write data in checkpoint logs (for recovery)
- flush all dirty blocks in buffer pool to disk

etc. etc. etc.

Clock-sweep Replacement Strategy

69/100

PostgreSQL page replacement strategy: *clock-sweep*

- treat buffer pool as circular list of buffer slots 缓冲槽
- `NextVictimBuffer` holds index of next possible evictee
- if this 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
- increment `NextVictimBuffer` and try again (wrap at end)

Exercise 6: PostgreSQL Buffer Pool

70/100

Consider an initally empty buffer pool with only 3 slots.

Show the state of the pool after each of the following:

```
Req R0, Req S0, Rel S0, Req S1, Rel S1, Req S2,
Rel S2, Rel R0, Req R1, Req S0, Rel S0, Req S1,
Rel S1, Req S2, Rel S2, Rel R1, Req R2, Req S0,
Rel S0, Req S1, Rel S1, Req S2, Rel S2, Rel R2
```

Treat `BufferDesc` entries as

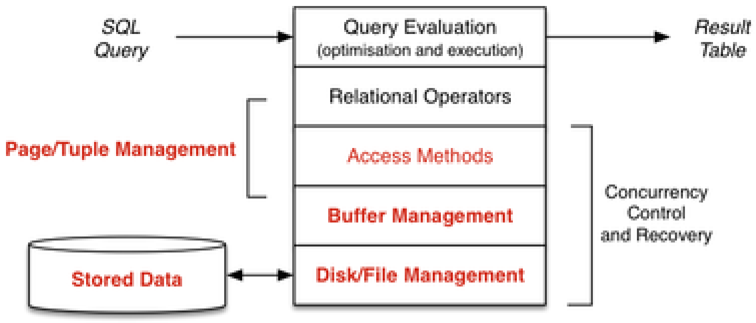
`(tag, usage_count, refcount, freeNext)`

Assume `freeList` and `nextVictim` global variables.

Pages

Page/Tuple Management

72/100



Pages

73/100

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

Page Formats

74/100

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

We want to interpret/manipulate it as a collection of `Record`s.

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

Note: `rid` contains `(PageId,TupIndex)`, so no explicit `pid` needed

... Page Formats

75/100

Factors affecting `Page` formats:

- determined by record size flexibility (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?

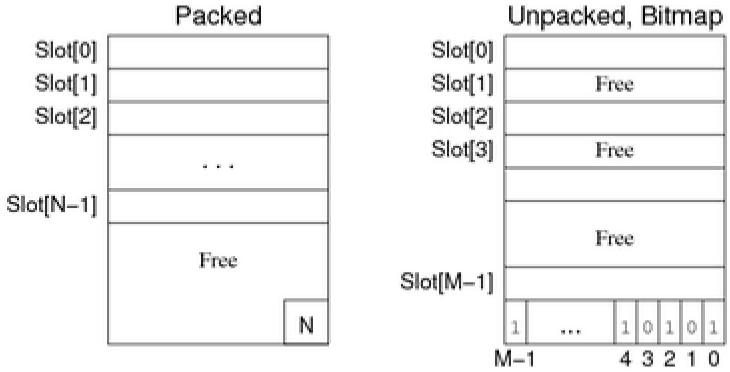
Implementation of `Page` operations critically depends on format.

... Page Formats

76/100

For fixed-length records, use *record slots*.

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



Exercise 7: Fixed-length Records

77/100

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 8: Inserting/Deleting Fixed-length Records

78/100

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

Page Formats

79/100

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)

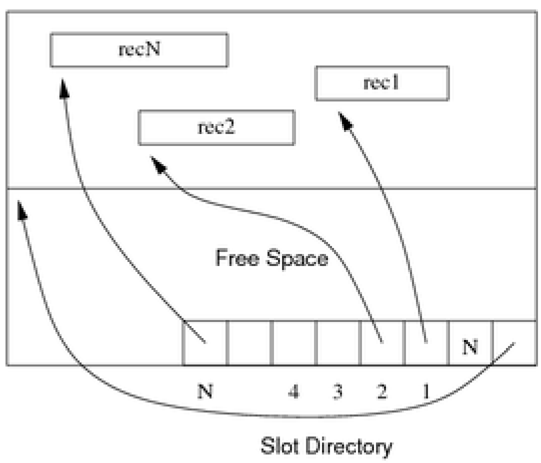
Important aspect of using slot directory

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

... Page Formats

80/100

Compacted free space:

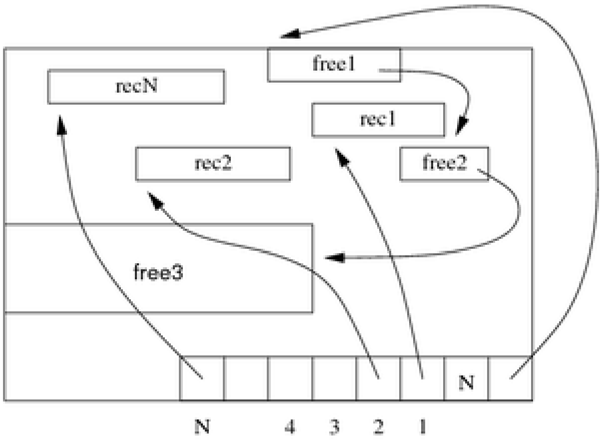


Note: "pointers" are implemented as word offsets within block.

... Page Formats

81/100

Fragmented free space:

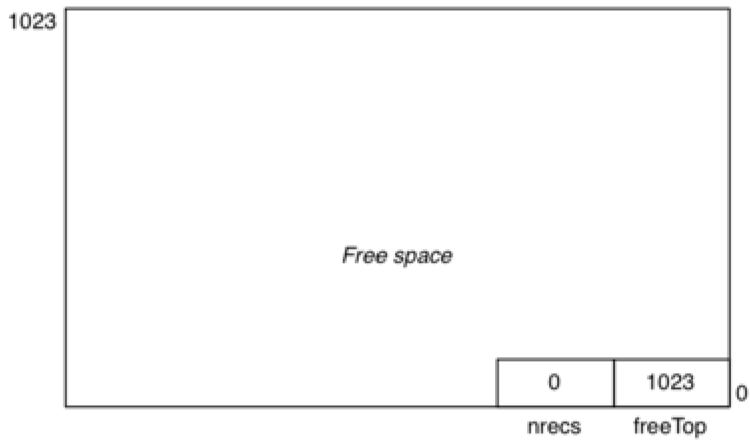


Slot Directory

... Page Formats

82/100

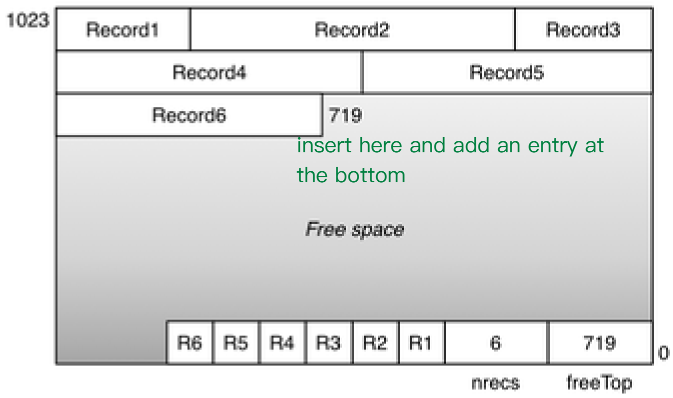
Initial page state (compacted free space) ...

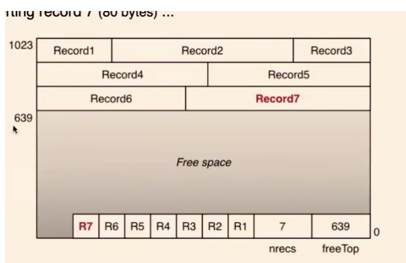


... Page Formats

83/100

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

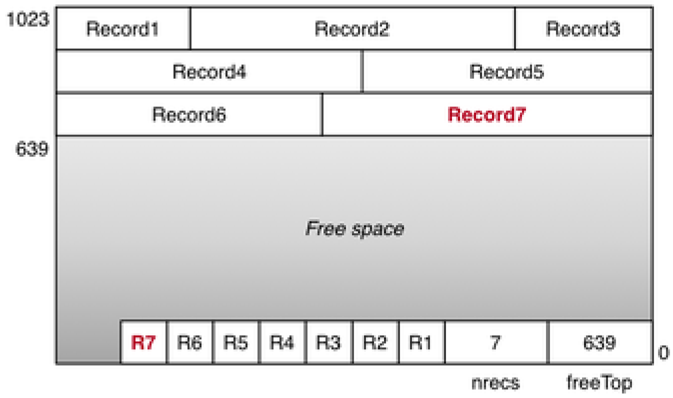




... Page Formats

84/100

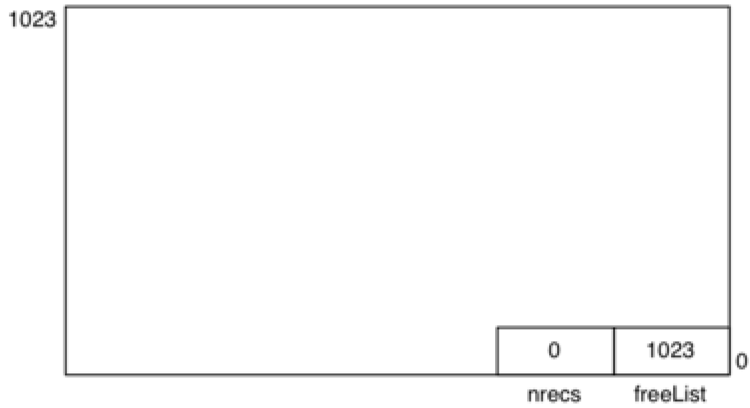
After inserting record 7 (80 bytes) ...



... Page Formats

85/100

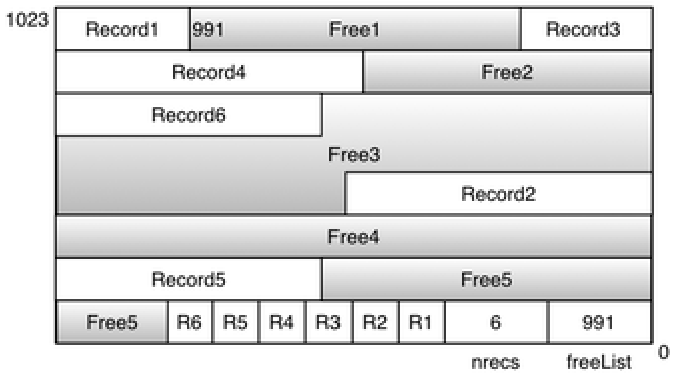
Initial page state (fragmented free space) ...



... Page Formats

86/100

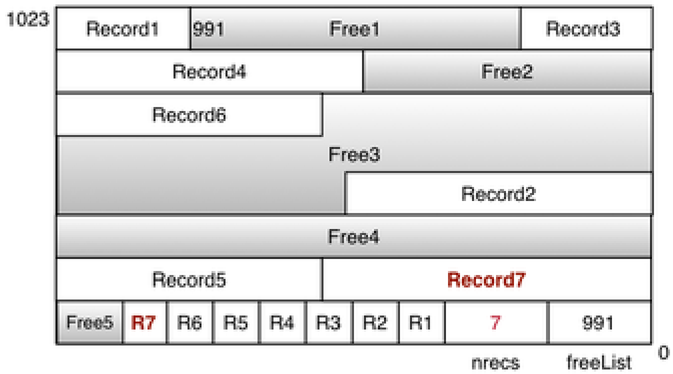
Before inserting record 7 (fragmented free space) ...



... Page Formats

87/100

After inserting record 7 (80 bytes) ...



Exercise 9: Inserting Variable-length Records

88/100

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

Storage Utilisation

89/100

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 ≤ PageSize*

tuple

Exercise 10: Space Utilisation

90/100

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 records 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.

Overflows

91/100

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

92/100

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)

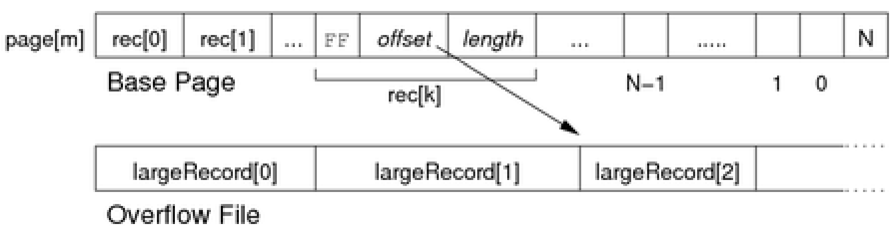
- cases (2) and (4) require an "overflow page"
- case (3) requires an "overflow file"

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

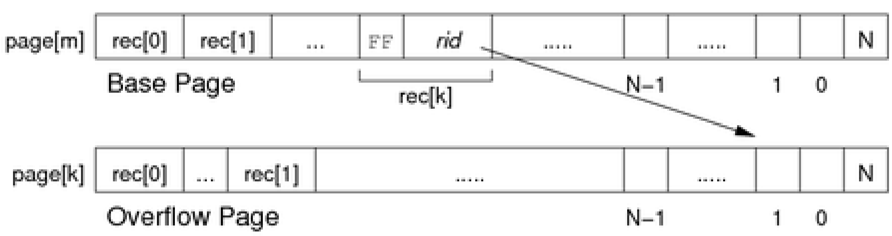
... Overflows

93/100

Overflow files for very large records and BLOBs:



Record-based handling of overflows:



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

94/100

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)

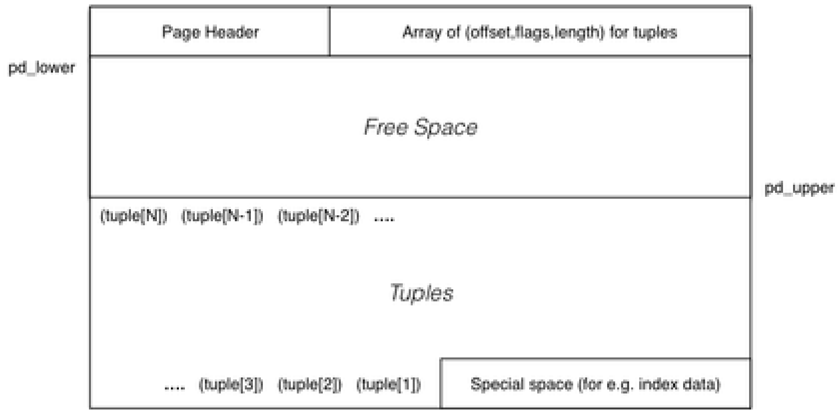
Large data items are stored in separate (TOAST) files (implicit)

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

... PostgreSQL Page Representation

95/100

PostgreSQL page layout:



... PostgreSQL Page Representation

96/100

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

97/100

Page-related data types: (cont)

```
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[1];  // beginning of line pointer array
} PageHeaderData;

typedef PageHeaderData *PageHeader;
```

... PostgreSQL Page Representation

98/100

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

99/100

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

Exercise 11: PostgreSQL Pages

100/100

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: 2 Mar 2020