>>

COMP9315 Week 02

COMP9315 25T1
DBMS Implementation

(Data structures and algorithms inside relational DBMSs)

Lecturer: **Xiaoyang Wang**

Web Site: http://www.cse.unsw.edu.au/~cs9315/

(If WebCMS unavailable, use
http://www.cse.unsw.edu.au/~cs9315/25T1/)

❖ Buffer Pool

<< ∧ >>

❖ Buffer Pool

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

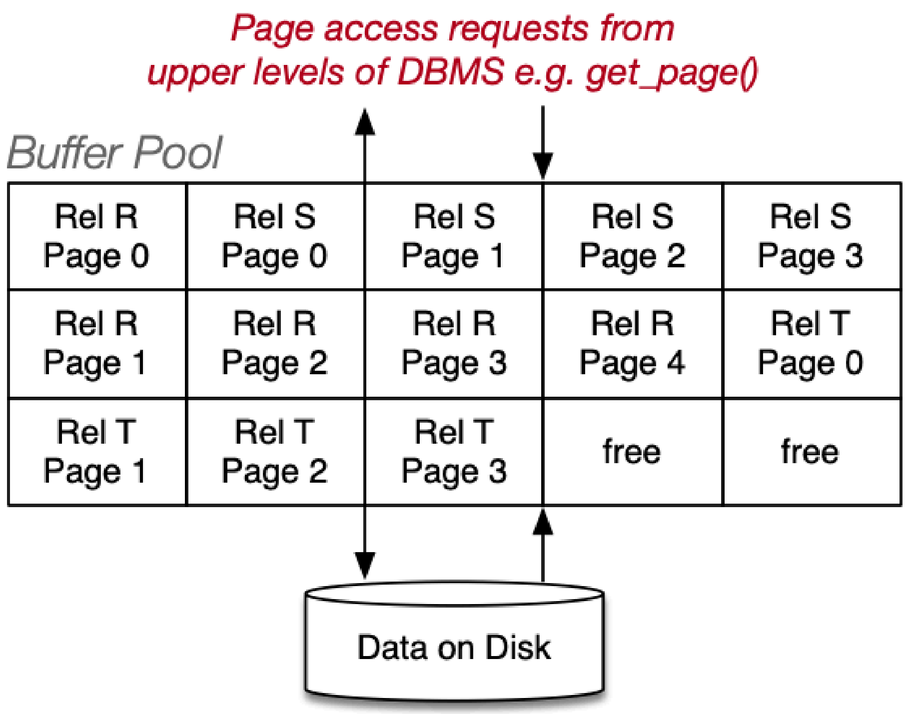
Uses:

- 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)`, ...

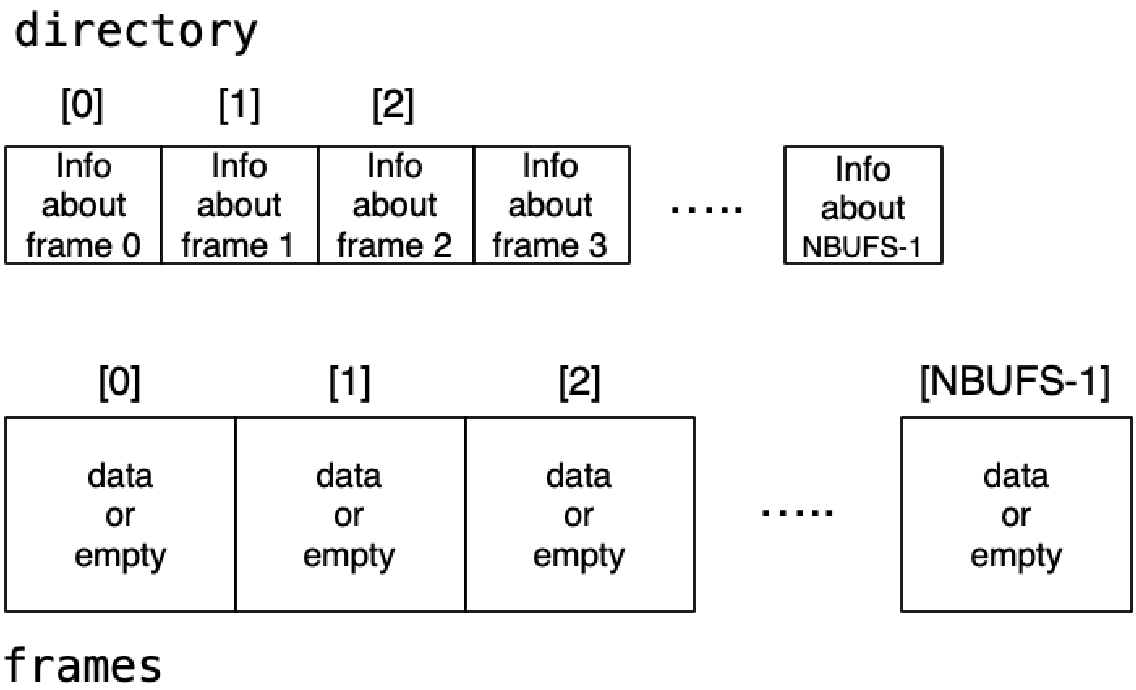
To some extent ...

- `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; just 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 operations 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.

<< ^ >>

❖ Exercise:

Consider the following scenarios:

- initially empty buffer pool with n=4 frames
- file with b=3 pages
- file with b=5 pages
- LRU page replacement strategy
- MRU page replacement strategy

Show state of buffer pool during two sequential scans of file.

Solution

❖ 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);
}
```

<< ^ >>

❖ Exercise: Buffer Cost Benefit (i)

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 $n < b_E$)

Solution

<< ^ >>

❖ Exercise: Buffer Cost Benefit (ii)

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

Write a C program to simulate buffer pool usage

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

Solution

<< ^ >>

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

<< ^ >>

❖ PostgreSQL Buffer Manager (cont)

PostgreSQL buffer descriptors:

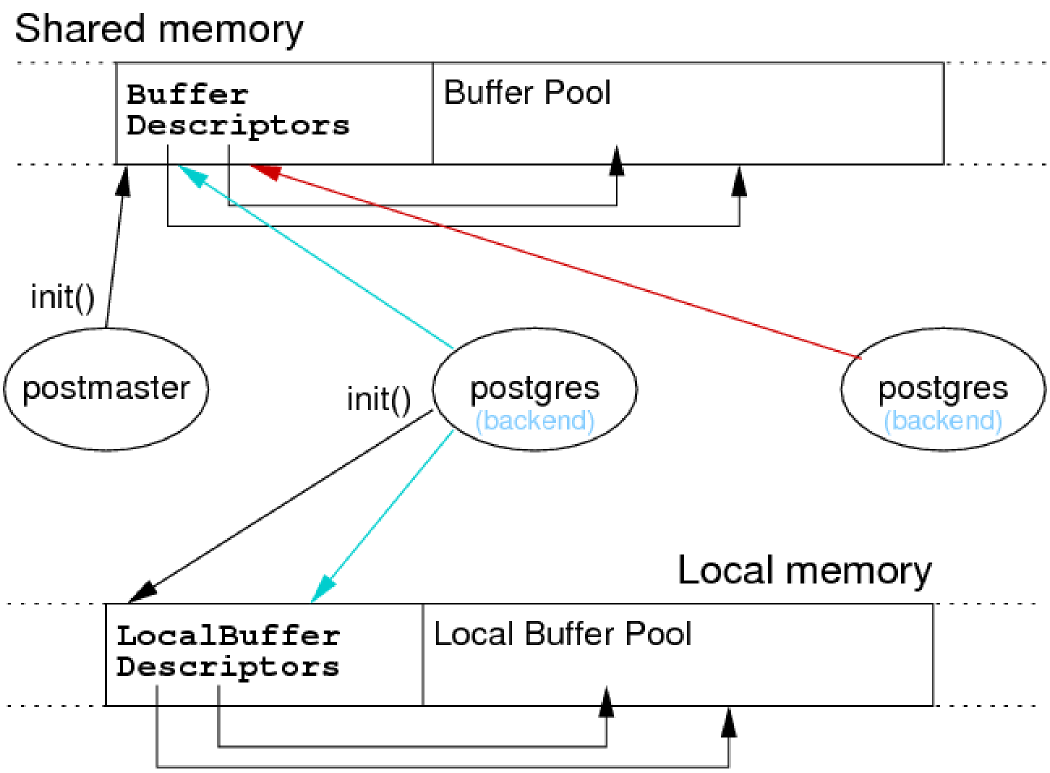
```
typedef struct BufferDesc
{
   BufferTag  tag;           /* ID of page contained in buffer */
   int        buf_id;        /* buffer's index number (from 0) */

   /* state of the tag, containing:
             flags, refcount and usagecount */
   pg_atomic_uint32 state;

   int         wait_backend_pgprocno;   /* pin-count waiter */
   int         freeNext;      /* link in freelist chain */
   LWLock      content_lock;  /* to lock access to buffer contents */
} BufferDesc;
```

<< ^ >>

❖ 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`)

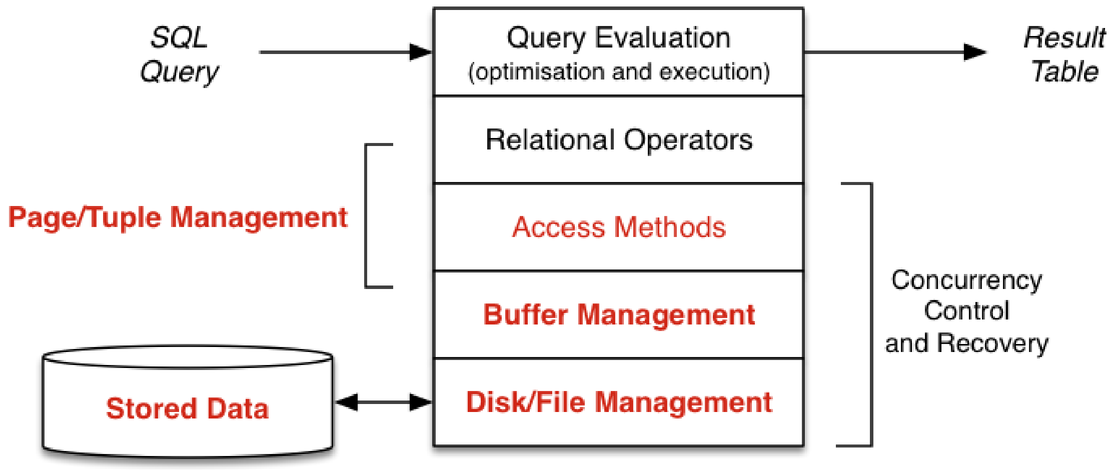
Code: `backend/storage/buffer/*.c`

Commentary: `backend/storage/buffer/README`

❖ Pages

<< ∧ >>

❖ Page/Tuple Management

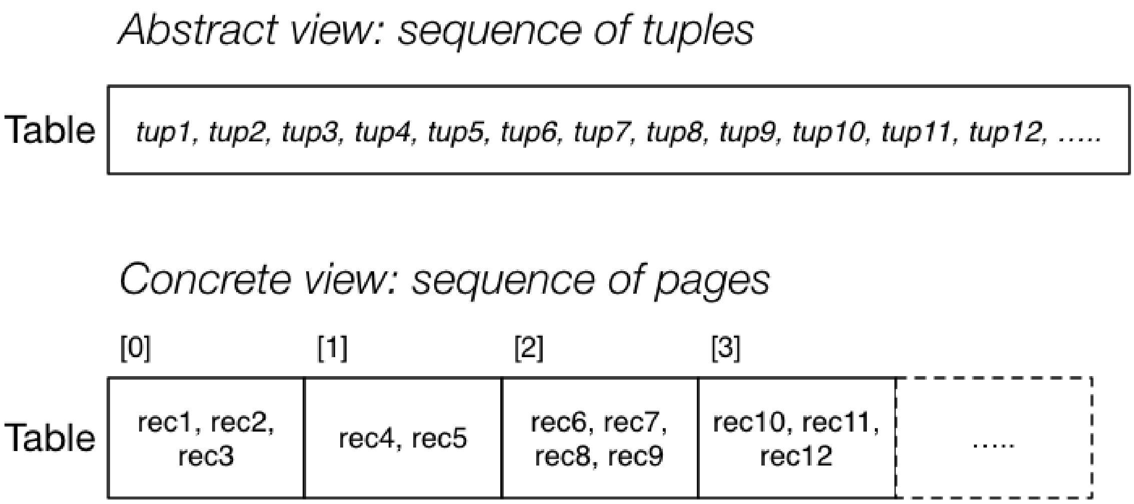


❖ Some terminology

Terminology used in these slides ...

- `Record` = sequence of bytes stored on disk (data for one tuple)
- `Tuple` = "interpretable" version of a `Record` in memory
- `Page` = copy of page from file on disk
- `PageId` = index of Page within file = `pid`
- `pageOffsetInFile` = `pid * PAGESIZE`
- `TupleId` = index of record within page = `tid`
- `RecordId` = (`PageId`, `TupleId`) = `rid`
- `recOffsetInPage` = `page.directory[tid].offset`
- `Relation` = descriptor for open relation

❖ Reminder: Views of Data



Each *tuple* is represented by a *record* in some *page*

<< ^ >>

❖ Page Formats

A `Page` is simply an array of bytes (`byte[]`).

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

Typical operations on pages and records:

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

Note: `rid` = `(PageId,TupleId)`, `rel` = open relation

<< ∧ >>

❖ Exercise: get_record(rel,rid)

Give an implementation of a function

```
Record get_record(Relation rel, RecordId rid)
```

which takes two parameters

- an open relation descriptor (`rel`)
- a record id (`rid`)

and returns the record corresponding to that `rid`

Solution

❖ Page Format

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.

<< ^ >>

❖ Exercise: Fixed-length Records (i)

How records are managed in Pages ...

- depends on whether records are fixed-length or variable-length

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?

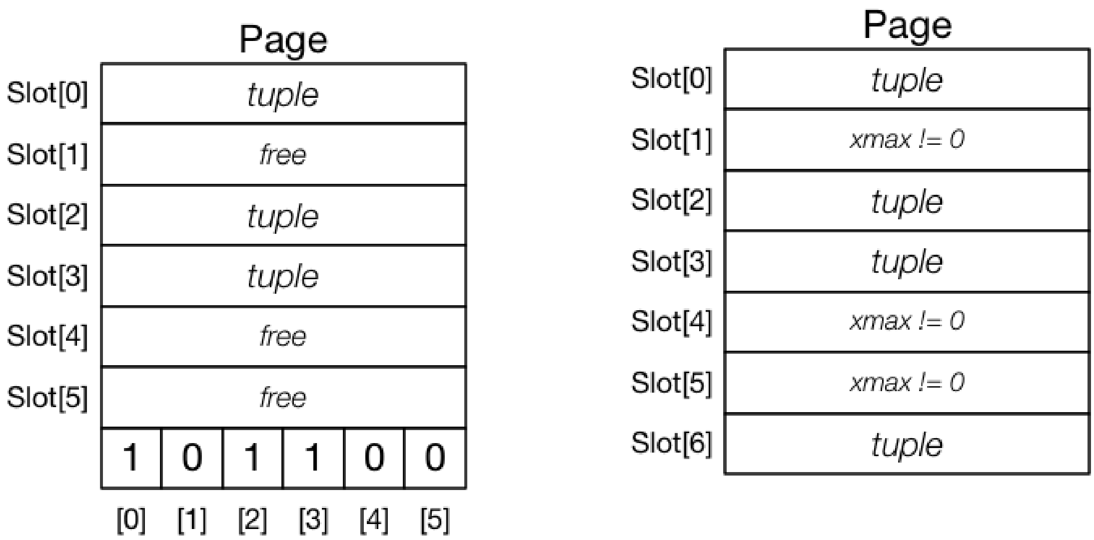
Solution

<< ∧ >>

❖ Exercise: Fixed-length Records (i) (cont)

For fixed-length records, use record slots.

- insert: place new record in first available slot
- delete: mark slot as free, or set *xmax*



<< ^ >>

❖ Exercise: Fixed-length Records (ii)

For the two fixed-length record page formats ...

Implement

- a suitable data structure to represent a `Page`
- insertion ... `rid = insert_record(rel,pid,rec)`
- deletion ... `delete_record(rel,rid)`

Ignore buffer pool (i.e. use `get_page()` and `put_page()`)

Solution

<< ^ >>

❖ Page Formats

For variable-length records, must use record directory

- `directory[i]` gives location within page of i^{th} record

An important aspect of using record directory

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

Issue with variable-length records

- managing space withing the page (esp. after deletions)
- recording used and unused regions of the page

We refer to tuple index within directory as `TupleId` `tid`

<< ∧ >>

❖ Page Formats (cont)

Possibilities for handling free-space within block:

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

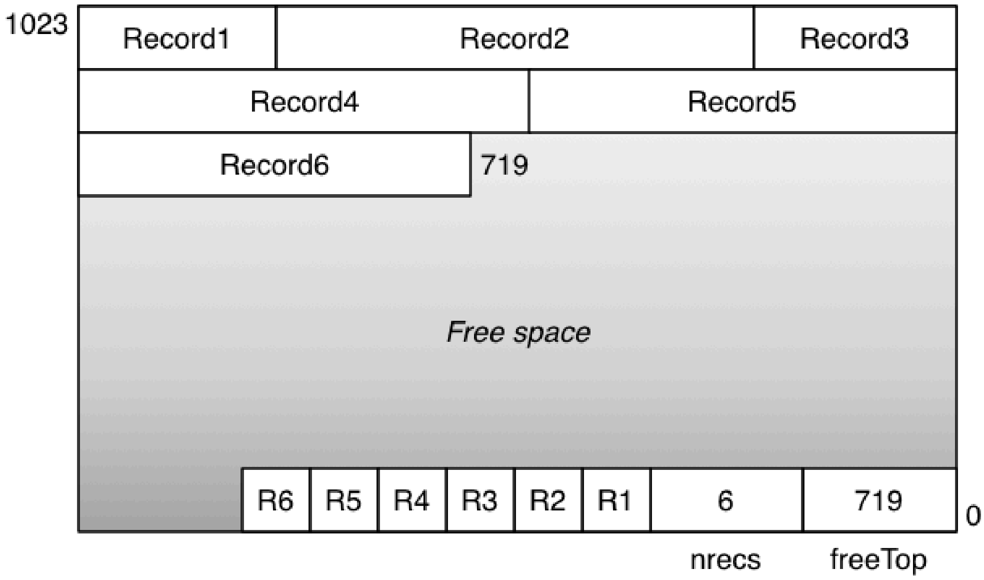
In practice, a combination is useful:

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

<< ∧ >>

❖ Page Formats (cont)

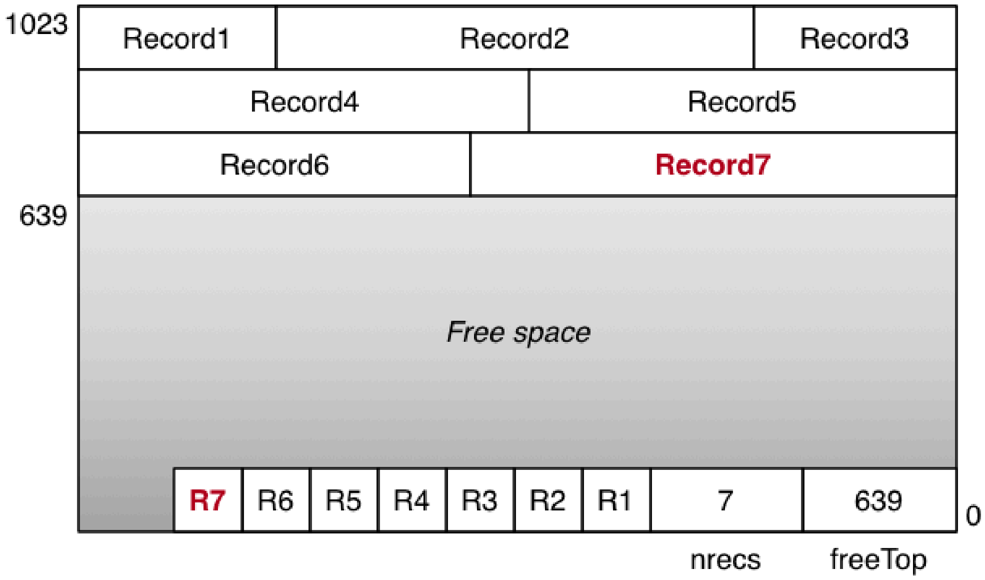
Compacted free space ... before inserting record 7



<< ^ >>

❖ Page Formats (cont)

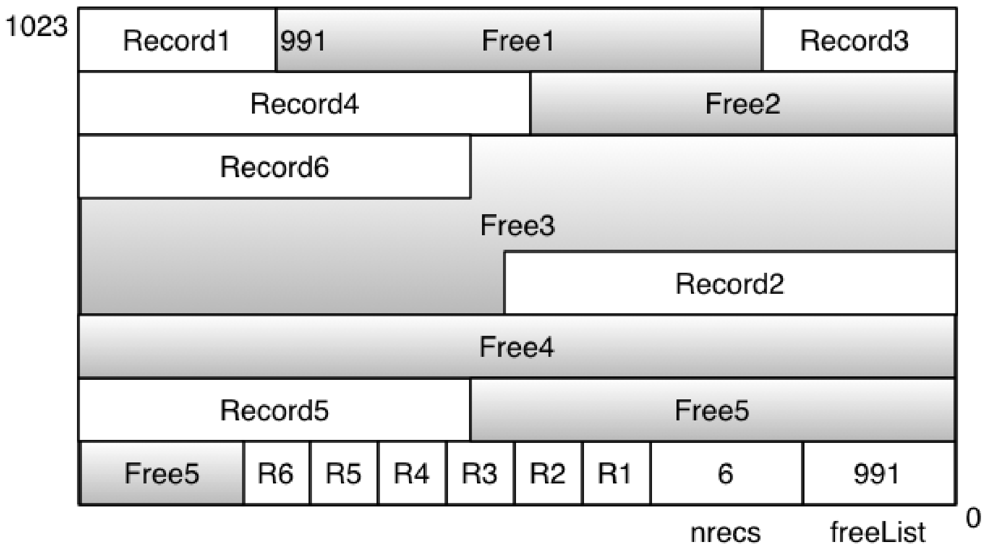
After inserting record 7 (80 bytes) ...



<< ^ >>

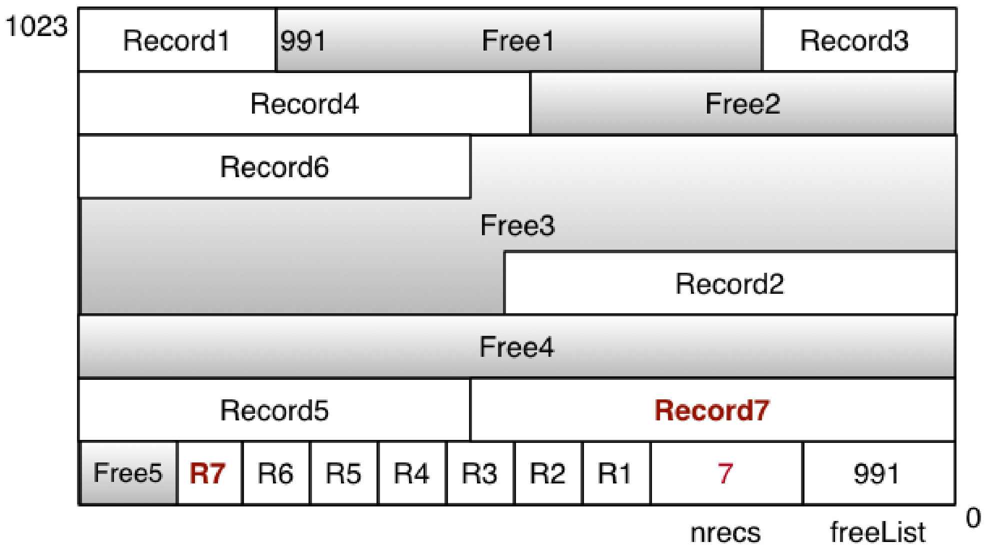
❖ Page Formats (cont)

Fragmented free space ... before inserting record 7



❖ 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, $HeaderSize + c*SlotSize + c*R \leq PageSize$

<< ^ >>

❖ Exercise: Space Utilisation

Consider the following page/record information:

- page size = 1KB = 1024 bytes = 2^{10} bytes
- records: `(w:int, x:varchar(20), y:char(10), z:int)`
- records are all aligned on 4-byte boundaries
- `x` field padded to ensure `z` starts on 4-byte boundary
- each record has 4 field-offsets at start of record (each 1 byte)
- `char(10)` field rounded up to 12-bytes to preserve alignment
- maximum size of `x` 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

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

1. no free-space fragment large enough
2. overall free-space in page 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)

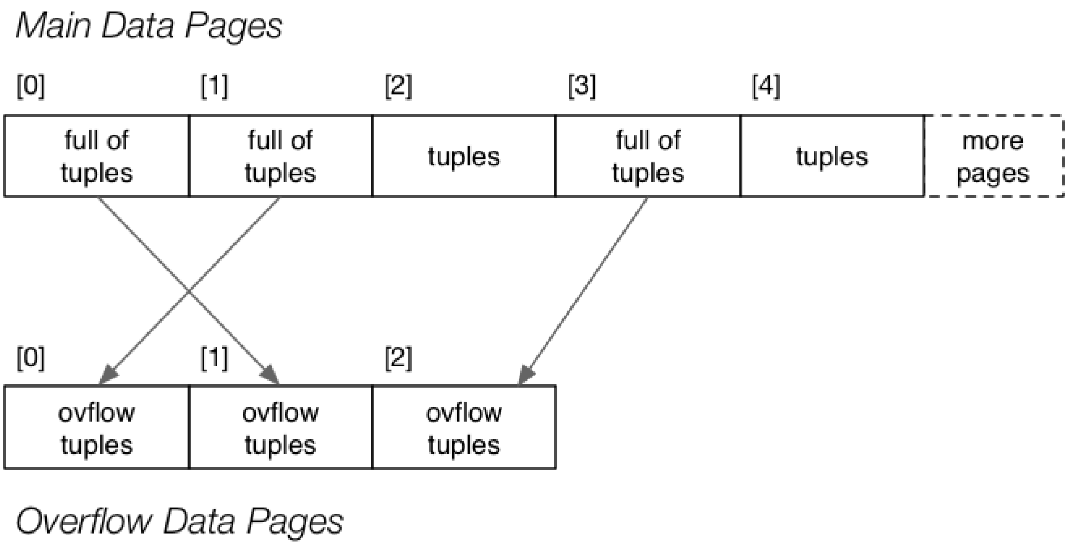
- cases (2) and (4) require an "overflow page"
- case (3) requires an "overflow file"

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

<< ^ >>

❖ Overflows (cont)

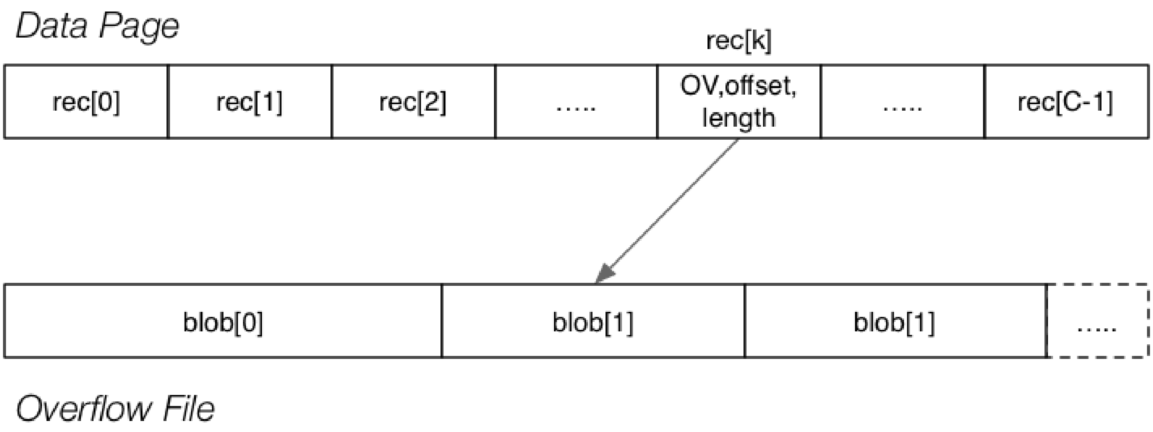
Overflow pages for full buckets in a hashed file:



<< ^ >>

❖ Overflows (cont)

Overflow file for very large records and BLOBs:



❖ 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)

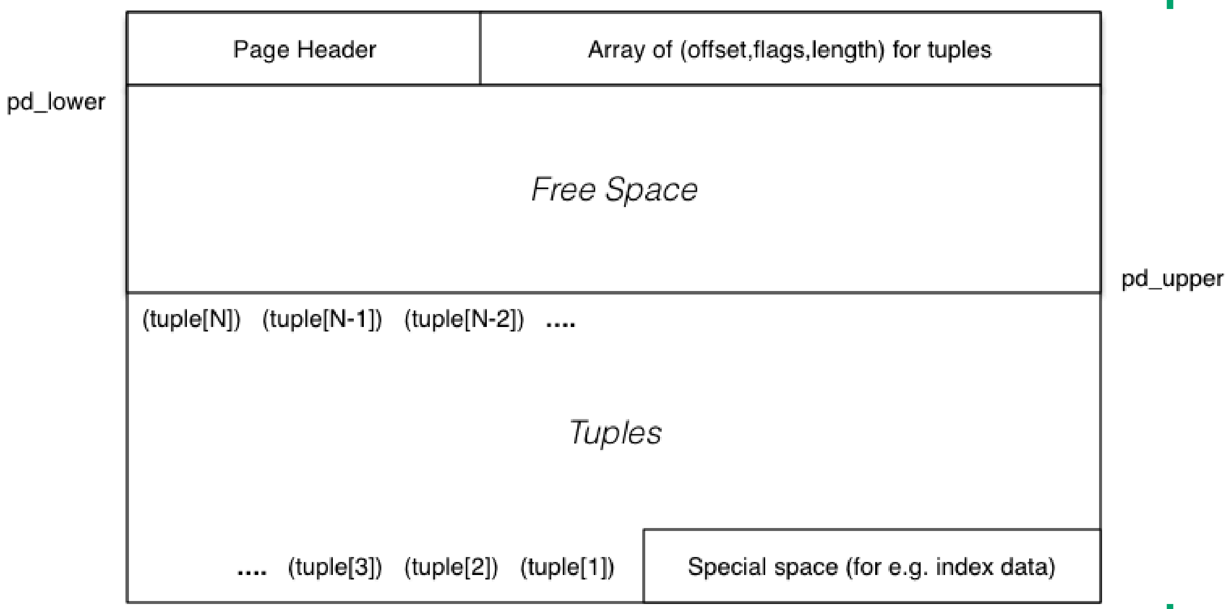
Large data items are stored in separate (TOAST) files (implicit)

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)

```
typedef struct PageHeaderData  (simplified)
{
  ...                          // transaction-related data
  uint16        pd_checksum;   // checksum
  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;
  ItemIdData    pd_linp[1];    // beginning of line pointer array
} 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.

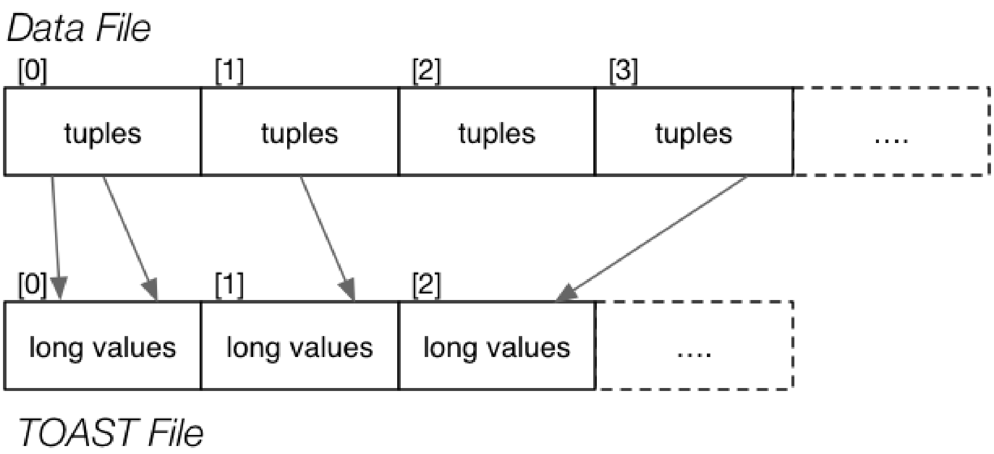
One important difference:

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

<< ^

❖ TOAST Files

Each data file has a corresponding TOAST file (if needed)



Tuples in data pages contain `rid`s for long values

TOAST = The Oversized Attribute Storage Technique

Produced: 24 Feb 2025