Lecture 3

**PostgreSQL Buffer Manager**

Provides a shared pool of memory buffers for all backends

All access methods get data from disk via buffer manager

Buffer pool consists of:

* **BufferDescriptors**

Shared fixed array (size **NBuffers**) of **BufferDesc**

* **BufferBlocks**

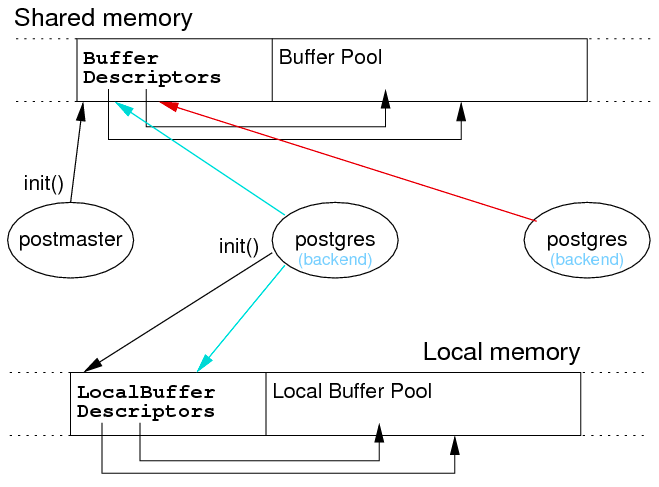
Shared fixed array (size **NBuffers**) of **Buffer**

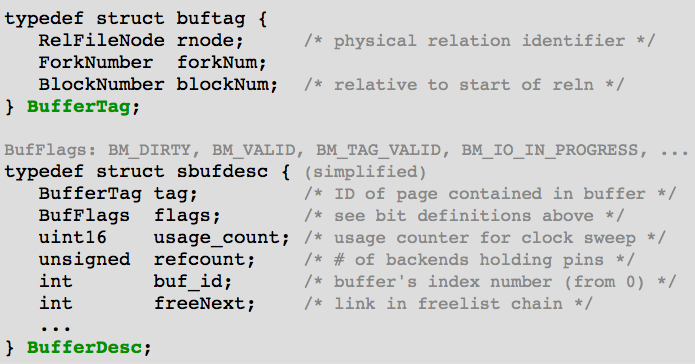
* **Buffer** = index values in above arrays

Indexes: global buffers **1..NBuffers**; local buffers negative

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







**Buffer Pool Functions**

**Buffer ReadBuffer (Relation r, BlockNumber n)**

**void ReleaseBuffer (Buffer buf)**

**void MarkBufferDirty (Buffer buf)**

**Page BufferGetPage (Buffer buf)**

**BufferIsPinned (Buffer buf)**

**CheckPointBuffers**

* Write data in checkpoint logs (for recovery)
* Flush all dirty blocks in buffer pool to disk

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

**Clock-sweep Replacement Strategy**

PostgreSQL page replacement strategy: clock-sweep

* Treat buffer pool as circular list of buffer slots
* **NextVictimBuffer** holds index of next possible evictee
* If 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)

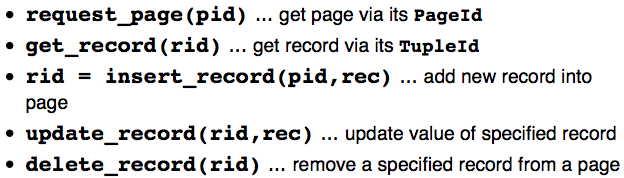
**Pages**

**TupleId** (aka **RecordId** or **RID**) **= (RelId + PageNum + TupIndex)**

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

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

Typical operations on **Page**s:



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

**Page Formats**

For fixed-length records, use record slots

Insert: place new record in first available slot

Delete: two possibilities for handling free record slots:



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:

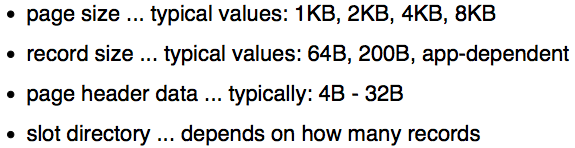
* Normally fragmented (cheap to maintain)
* Compacted when needed (e.g. record won’t fit)

Important aspect of using slot directory

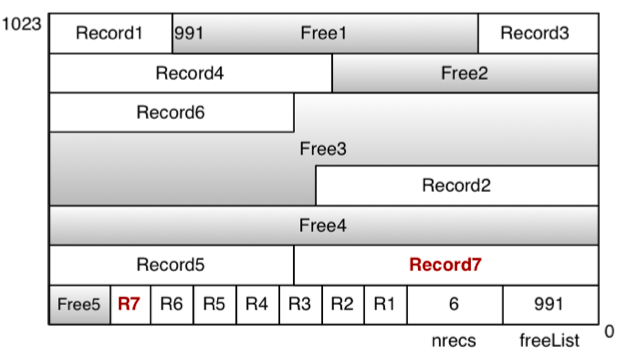
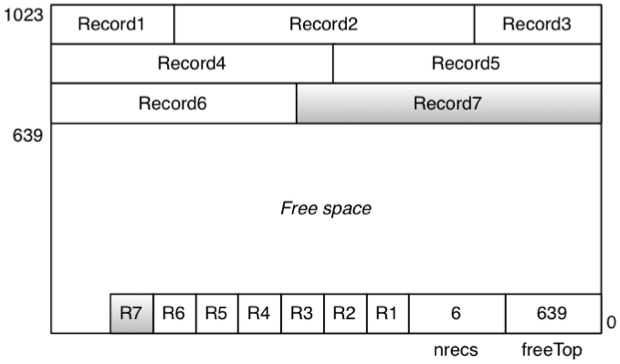
* Location of tuple within page can change, tuple index does not change

**Storage Utilisation**

How many records can fit in a page? (C = capacity) **HeaderSize + C\*SlotSize + C\*R ≤ PageSize**



e.g. Insert record 7 -- Compacted free space **vs** fragmented free space



**Overflows**

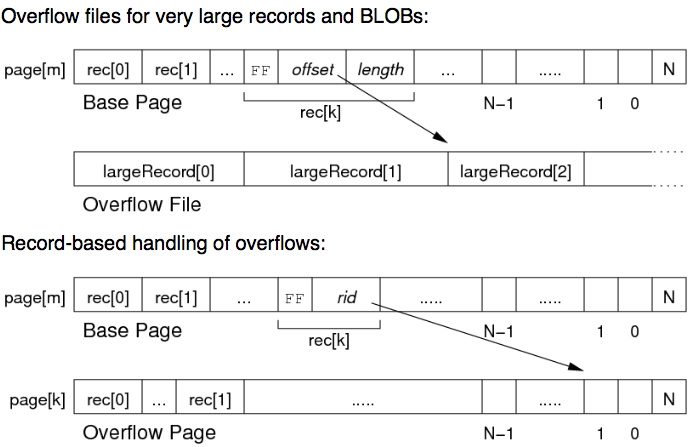
Sometimes, it may not be possible to insert a record into a page:

1. No free-space fragment large enough // can first try to compact free-space within the page
2. Overall free-space is not large enough // can make a new page
3. The record is larger than the page // requires either spanned records or “overflow file”
4. No more free directory slots in page // can make a new page

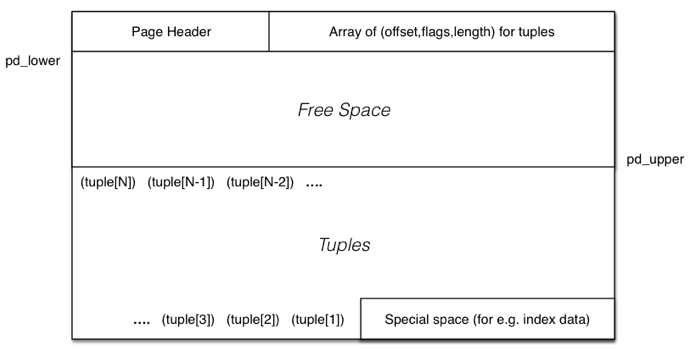
The above solutions are when records may be inserted anywhere that there is free space

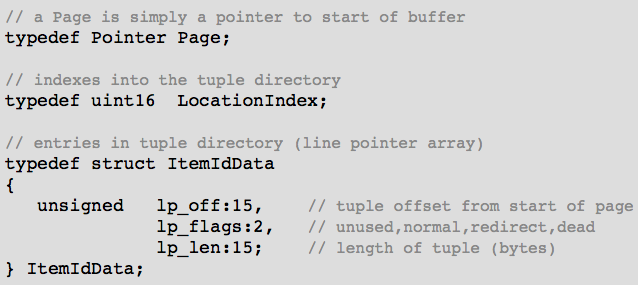
If file organisation determines record placement (e.g. hashed file)

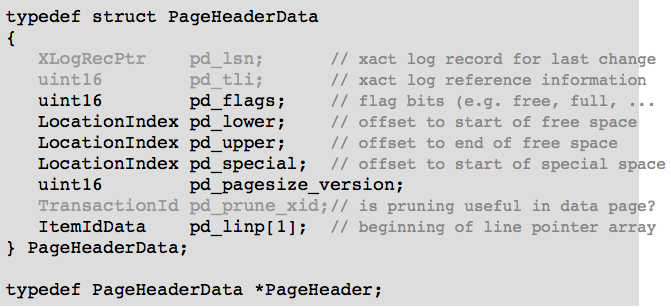
* 2&4 require an “overflow page”
* 3 requires an “overflow file”



**PostgreSQL Page Representation**

****

****

****

Operations on **Page**s:

void PageInit (Page page, Size pageSize, …)

* In particular, set **pd\_lower** and **pd\_upper**

OffsetNumber PageAddItem (Page page, Item item, Size size, …)

void pageRepairFragmentation (Page page)

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

But index entries tend to be a smaller than tuples, can typically fit more index entries per page

**Records vs Tuples**

Tuple = collection of attribute values for such a schema, e.g.



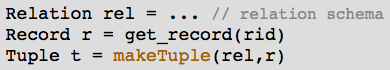
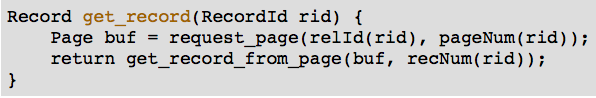
Record = sequence of bytes, containing data for one tuple, e.g.



Byte-sequence needs to be interpreted relative to schema to get tuple

**Operations on Records**

Access record via **RecordId**: **RecordId = TupleId = (RelId, PageNum, TupIndex)**



update\_record (rid, rec)

rid = insert\_record (pid, rec)

delete\_record (rid)

**Operations on Tuples**

Tuple t = makeTuple (rel, rec)

Typ getTypField (Tuple t, int fno)

void setTypField (Tuple t, int fno, Typ val)

**Operations for Access Methods**

Tuple get\_tuple (RecordId rid)

Tuple get\_tuple\_from\_page (Page p, int rno)

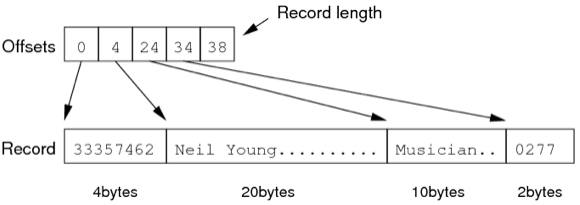
Scan s = start\_scan (Rel r, …)

Tuple next\_tuple(Scan s)

**Fixed-length Records**

Encoding scheme for fixed-length records:

* Record format (length + offsets) stored in catalogue
* Data values stores in fixed-size slots in data pages



**Variable-length Records**

Some encoding schemes for variable-length records:

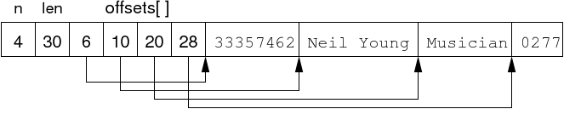
* Prefix each field by length



* Terminate fields by delimiter

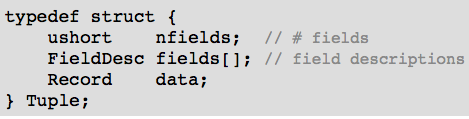


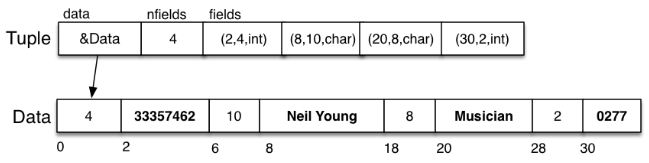
* Array of offsets



**Converting Records to Tuples**

**FieldDesc** gives (offset, length, type) information





**PostgreSQL Tuples**

Tuple-related data types:



The actual data value:

* May be stored in the **Datum** (e.g. **int**)
* May have a header with length (for varlen attributes)
* May be stored in a TOAST file

**Relational Operations**

Tuple = record – collection of data values under some schema

Page = block = collection of tuples + management data = i/o unit

Relation = table ≈ file = collection of tuples

Two “dimensions of variation”

* Which relational operation (e.g. Sel, Proj, Join, Sort, …)
* Which access-method (e.g. file struct; heap, indexed, hashed, …)

**Cost Models**

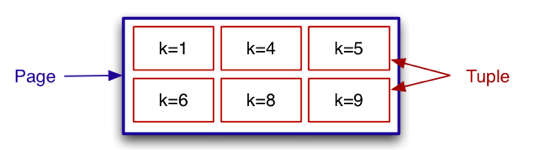
Cost can be measured in terms of

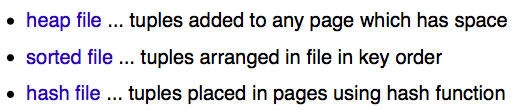
* Time Cost: total time taken to execute method, or
* Page Cost: number of pages read and/or written

Primary assumptions in our cost models:

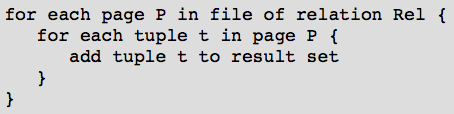
* Memory (RAM) is “small”, fast, byte-at-a-time
* Disk storage is very large, slow, page-at-a-time

When describing file structures, sometimes refer to tuples via their key





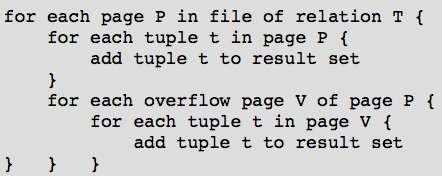
**Scanning**



Cost: read every data page once

Time Cost = b\*Tr Page Cost = b

When file has overflow pages



Cost: read each data and overflow page once

Cost = b + bov, where bov = total number of overflow pages