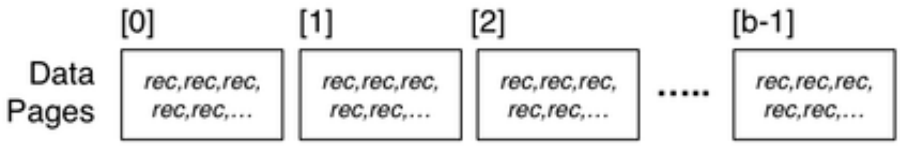
Week 03 Lecture

DBMS Parameters

1/71

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

2/71

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	-	≅ 0

Buffer Pool

Buffer Pool

4/71

Buffer operations: *(all take single `PageId` argument)*

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

Buffer pool data structures:

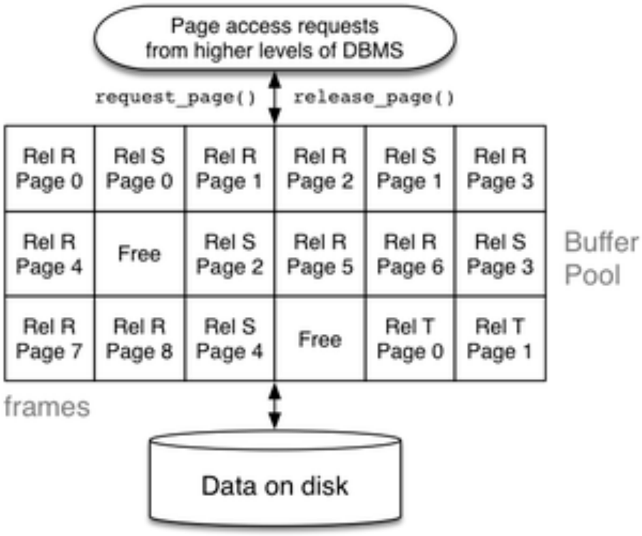
- `Page frames[NBUFS]; FrameData directory[NBUFS];`
- `Page` is `byte[BUFSIZE]`, `FrameData` is `struct {...}`

For each frame, we need to know: (`FrameData`)

- *which Page (i.e. PageId = (RelId,PageNum)) it contains, or 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)*

... Buffer Pool

5/71



Page Replacement Policies

6/71

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

7/71

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 ≥ 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

8/71

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

9/71

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 1: Buffer Management Cost Benefit (i)

10/71

Assume that:

- *the `Customer` relation has b_C pages (e.g. 5)*
- *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)*
- *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 2: Buffer Management Cost Benefit (ii)

11/71

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)*

PostgreSQL Buffer Manager

12/71

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

13/71

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*

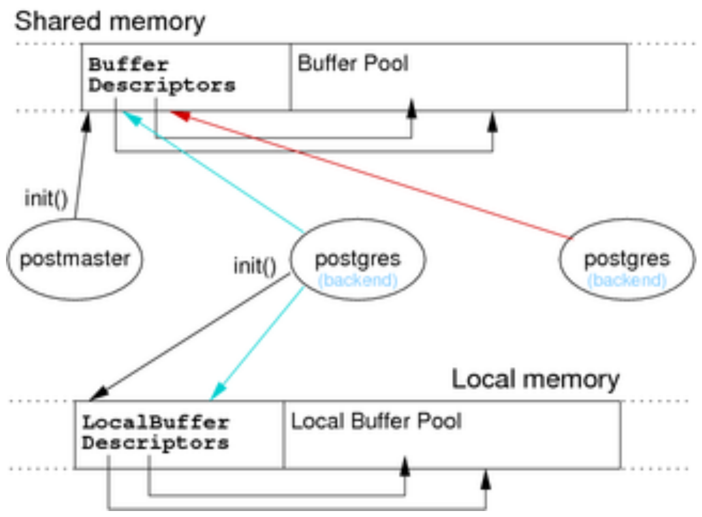
- *indexes: global buffers* `1..NBuffers`*; local buffers negative*

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

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

... PostgreSQL Buffer Manager

14/71



... PostgreSQL Buffer Manager

15/71

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***

Buffer Pool Data Types

16/71

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

BufFlags: BM_DIRTY, BM_VALID, BM_TAG_VALID, BM_IO_IN_PROGRESS, ...
typedef struct sbufdesc { (simplified)
   BufferTag tag;         /* ID of page contained in buffer */
   BufFlags  flags;       /* see bit definitions above */
   uint16    usage_count; /* usage counter for clock sweep */
   unsigned  refcount;    /* # of backends holding pins */
   int       buf_id;      /* buffer's index number (from 0) */
   int       freeNext;    /* link in freelist chain */
   ...
} BufferDesc;
```

Buffer Pool Functions

17/71

Buffer manager interface:

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

- *ensures n^{th} page of file for relation `r` is loaded*
  (may need to remove an existing unpinned page and read data from file)
- *increments reference (pin) count and usage count for buffer*
- *returns index of loaded page in buffer pool (`Buffer` value)*
- *assumes main fork, so no `ForkNumber` required*

Actually a special case of `ReadBuffer_Common`, which also handles variations like different replacement strategy, forks, temp buffers, ...

... Buffer Pool Functions

18/71

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

19/71

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.

... Buffer Pool Functions

20/71

Important internal buffer manager function:

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

- *used by **`ReadBuffer`** to find a buffer for (r,f,n)*
- *if (r,f,n) already in pool, pin it and return descriptor*
- *if no available buffers, select buffer to be replaced*
- *returned descriptor is pinned and marked as holding (r,f,n)*
- *does not read; **`ReadBuffer`** has to do the actual I/O*

Clock-sweep Replacement Strategy

21/71

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)*

For specialised kinds of access (e.g. sequential scan), can allocate a private "buffer ring" with different replacement strategy.

Exercise: PostgreSQL Buffer Pool

22/71

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*

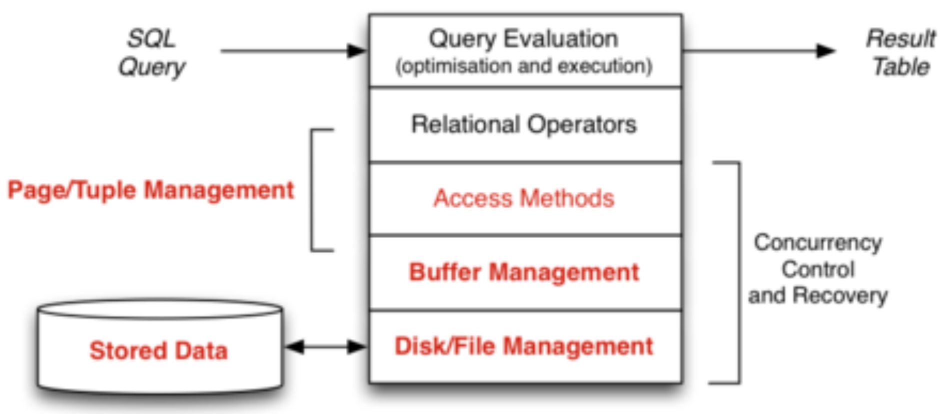
```
(tag, usage_count, refcount, freeNext)
```

Assume `freeList` *and* `nextVictim` *global variables.*

Pages

Page/Tuple Management

24/71



Pages

25/71

Database applications view data as:

- *a collection of records (tuples)*
- *records can be accessed via a `TupleId` (aka `RecordId` or `RID`)*
- *`TupleId` = (`RelId` + `PageNum` + `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

26/71

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

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

Typical operations on `Pages`:

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

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

---

... Page Formats

27/71

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 `Pages`?*

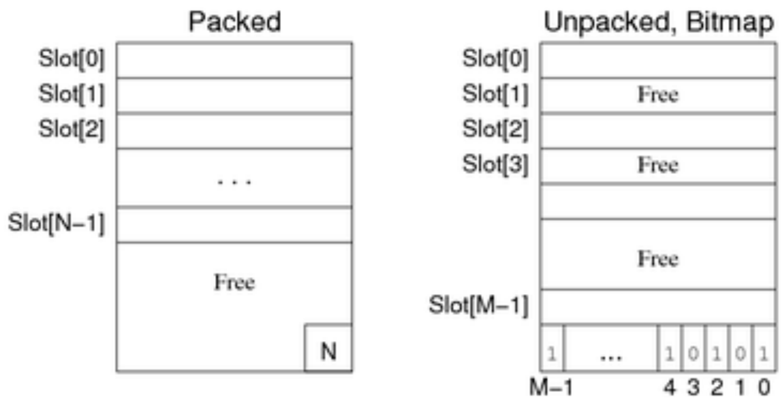
Implementation of `Page` operations critically depends on format.

... Page Formats

28/71

For fixed-length records, use record slots.

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



Exercise: Fixed-length Records

29/71

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?

Page Formats

30/71

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)*

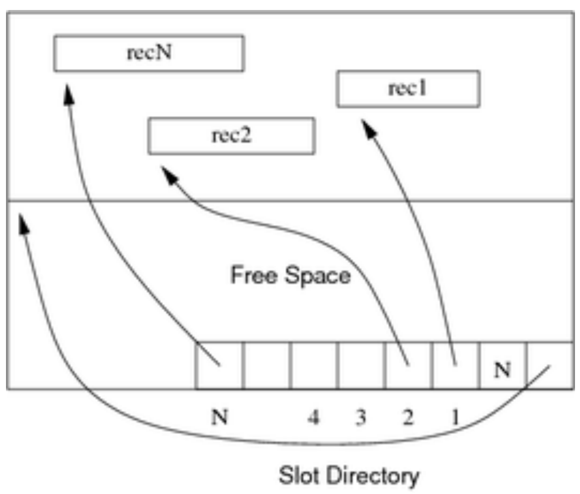
In practice, a combination is useful:

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

... Page Formats

31/71

Compacted free space:

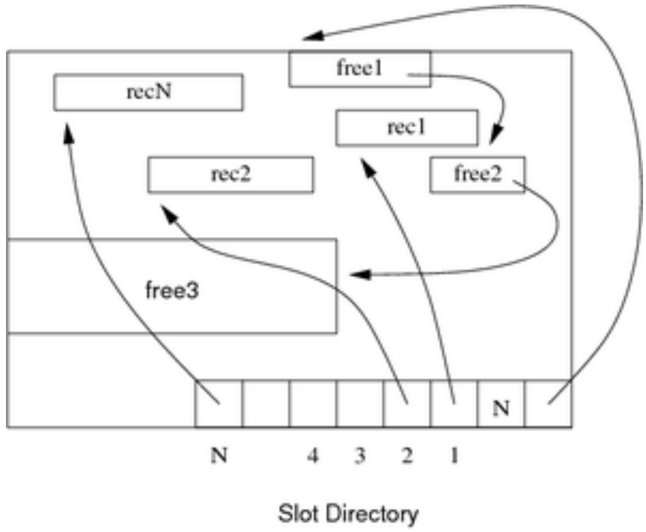


Note: "pointers" are implemented as word offsets within block.

... Page Formats

32/71

Fragmented free space:



Example: Inserting Records

33/71

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.

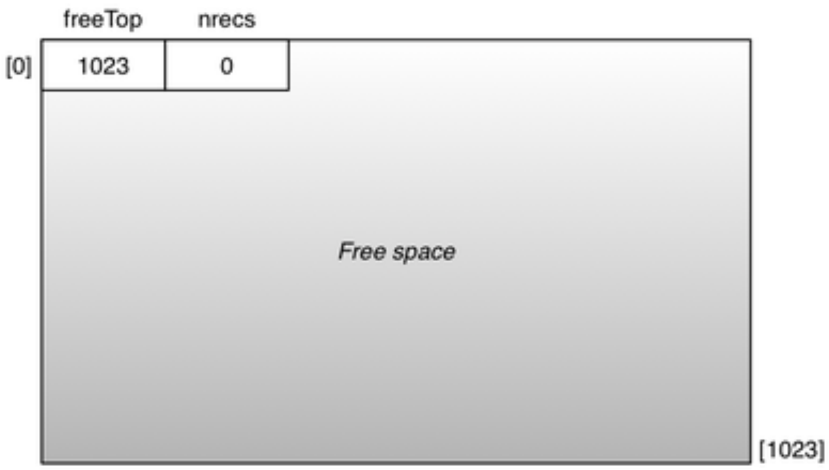
Use the page format on the following slides, but also assume:

- *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*

... Example: Inserting Records

34/71

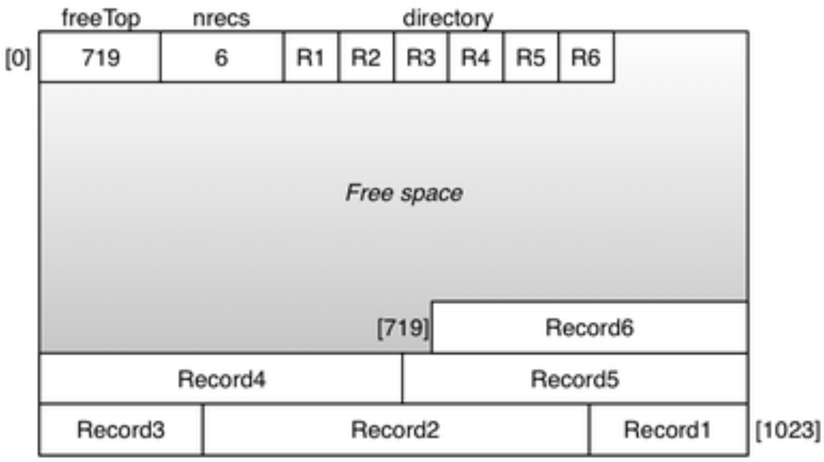
Initial page state (compacted free space) ...



... Example: Inserting Records

35/71

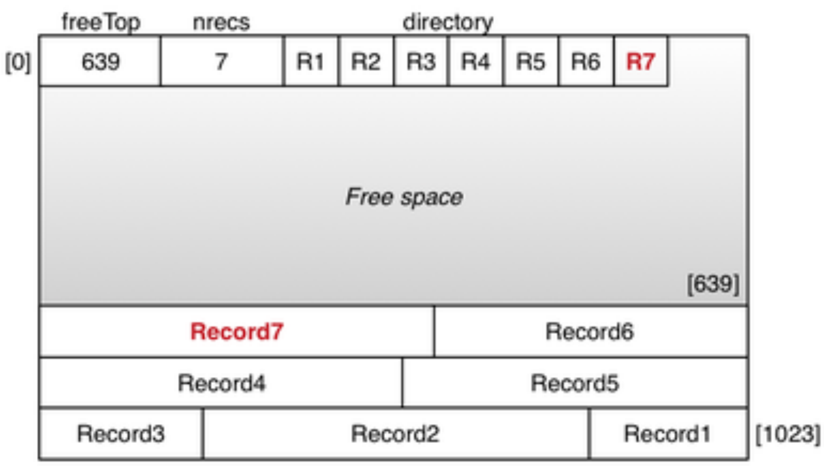
Before inserting record 7 (compacted free space) ...



... Example: Inserting Records

36/71

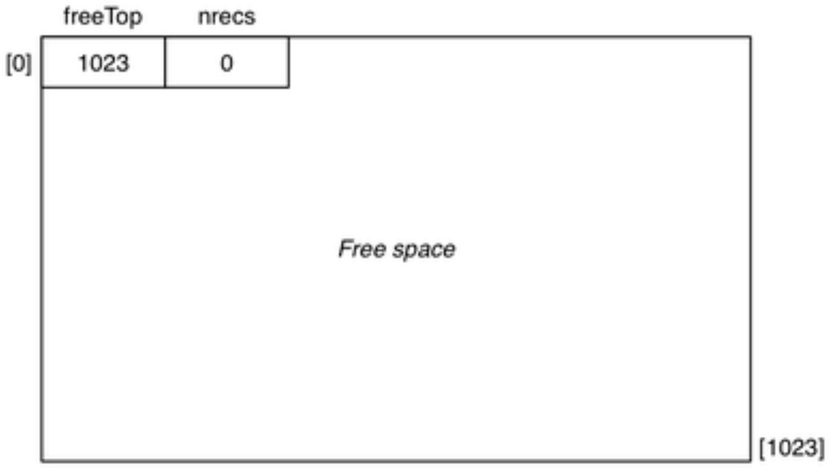
After inserting record 7 (80 bytes) ...



... Example: Inserting Records

37/71

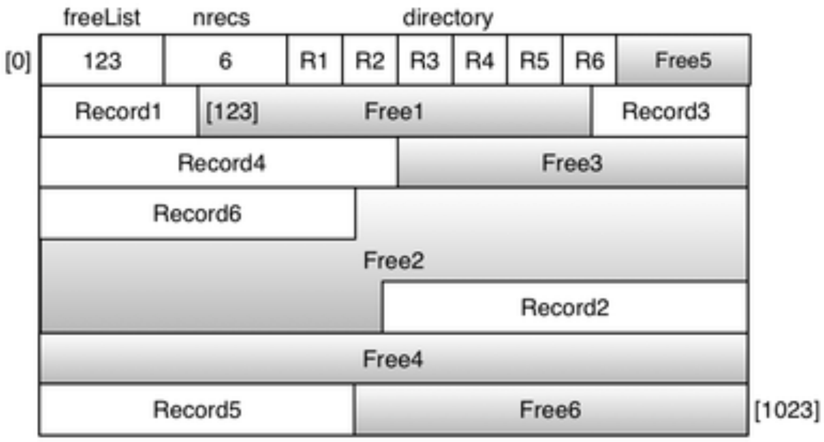
Initial page state (fragmented free space) ...



... Example: Inserting Records

38/71

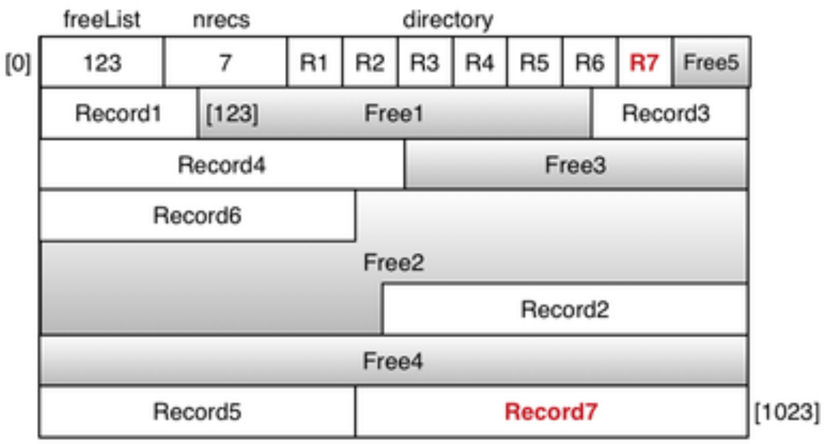
Before inserting record 7 (fragmented free space) ...



... Example: Inserting Records

39/71

After inserting record 7 (80 bytes) ...



Storage Utilisation

40/71

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*

Exercise 3: Space Utilisation

41/71

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

42/71

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

43/71

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)

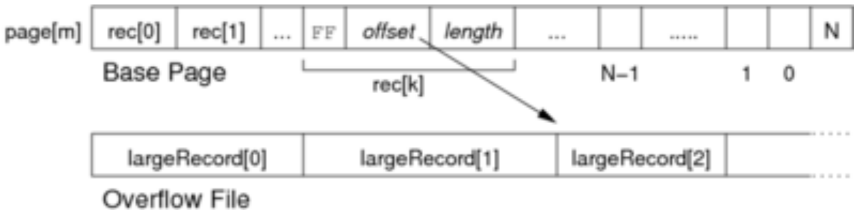
- *cases (2) and (4) require an "overflow page"*
- *case (3) requires an "overflow file"*

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

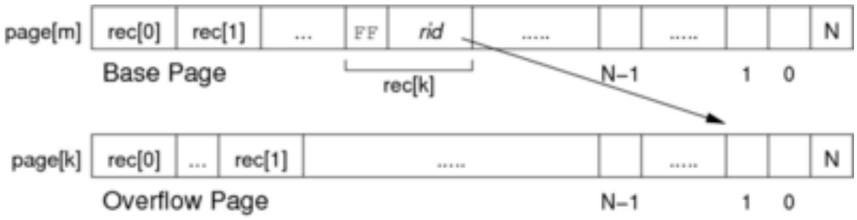
... Overflows

44/71

Overflow files for very large records and BLOBs:



Record-based handling of overflows:



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

PostgreSQL Page Representation

45/71

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)*

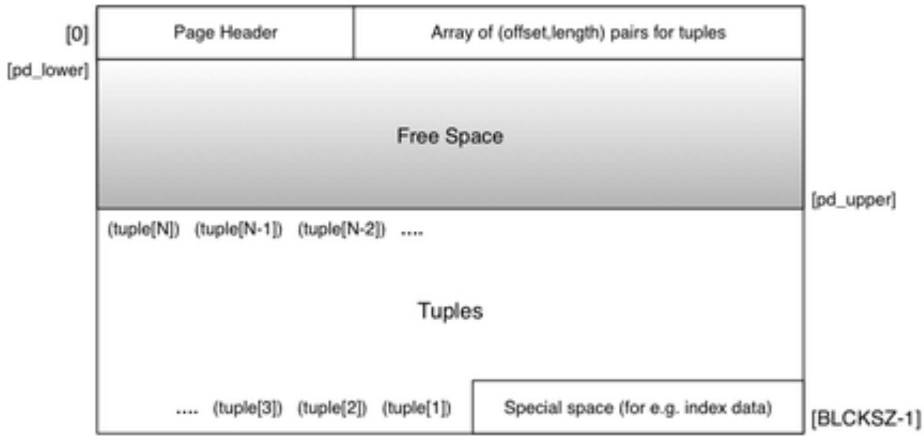
Large data items are stored in separate (TOAST) files (implicit)

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

... PostgreSQL Page Representation

46/71

PostgreSQL page layout:



... PostgreSQL Page Representation

47/71

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

48/71

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

49/71

Operations on `Pages`*:*

`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

50/71

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: PostgreSQL Pages

51/71

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.

Tuples

Records vs Tuples

53/71

A table is defined by a collection of attributes (schema), e.g.

```
create table Employee (
   id#  integer primary key,
   name varchar(20),    -- or char(20)
   job  varchar(10),    -- or char(10)
   dept number(4)
);
```

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

```
    (33357462, 'Neil Young', 'Musician', 0277)
```

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

Operations on Records

54/71

Simplest operation to access a record via its RID:

```
Record get_record(RecordId rid) {
    Page buf = request_page(relId(rid), pageNum(rid));
    return get_record_from_page(buf, recNum(rid));
}
```

where `TupleId = RecordId = (RelId, PageNum, TupIndex)`

Gives a sequence of bytes, which needs to be "tuple-fied", e.g.

```
Record r = get_record(rid)
Tuple t = makeTuple(rel,rec)
```

Requires knowledge of relation schema (`rel`)

... Operations on Records

55/71

Other operations on records (via their RID) ...

`update_record(rid,rec)`

- *modifies a record "in place" (replaced by new rec)*

`rid = insert_record(pid,rec)`

- *insert record into specified page, returning RID of new record*

`delete_record(rid)`

- *remove record (mark as deleted)*

All of the above, first require a page fetch (via buffer pool)

Operations on Tuples

56/71

`Tuple t = makeTuple(rel,rec)`

- *convert record to tuple data structure (may be identity mapping)*

`Typ   getTypField(Tuple t, int fno)`

- *extract the* `fno`*'th field from a* `Tuple` *as a value of type Typ*

E.g. `getIntField(t,1)`, `getStrField(t,2)`

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

- *set the value of the* `fno`*'th field of a* `Tuple` *to* `val`

E.g. `setIntField(t,1,42)`, `setStrField(t,2,"abc")`

Operations for Access Methods

57/71

Tuple get_tuple(RecordId rid)

- *fetch the tuple specified by `rid`; return reference to `Tuple`*

Tuple get_tuple_from_page(Page p, int recNum)

- *get the `recNum`'th tuple from an already-buffered page*

Access methods typially involve iterators, e.g.

Tuple next_tuple(Scan s)

- *return `Tuple` immediately following last accessed one*
- *returns `NULL` if no more `Tuples` left in the relation*
- *`Scan` holds data on progress through file (e.g. current page)*
- *`Scan` may include condition to implement `WHERE`-clause*

Example Query

58/71

Example: simple scan of a table ...

```
select name from Employee
```

implemented as:

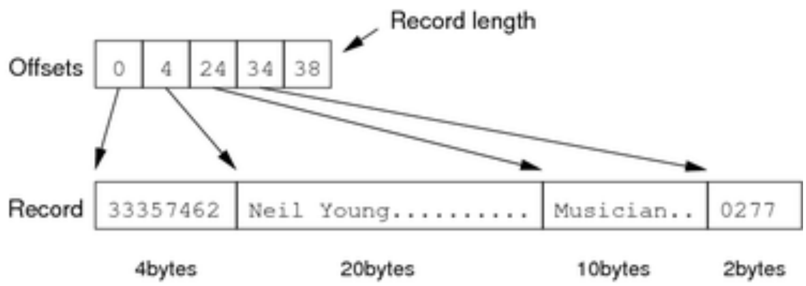
```
DB db = openDatabase("myDB");
Rel r = openRel(db,"Employee");
Scan s = start_scan(r);
Tuple t;
while ((t = next_tuple(s)) != NULL)
{
   char *name = getStrField(t,2);
   printf("%s\n", name);
}
```

Fixed-length Records

59/71

Encoding scheme for fixed-length records:

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



Since record format is frequently used at query time, should be in memory.

Variable-length Records

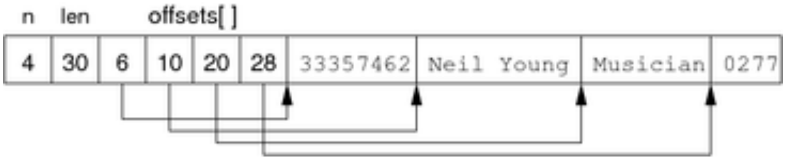
60/71

Some encoding schemes for variable-length records:

- *Prefix each field by length*

- *Terminate fields by delimiter*

- *Array of offsets*



Converting Records to Tuples

61/71

A `Record` is an array of bytes (`byte[]`)

- *representing the data values from a typed `Tuple`*

A `Tuple` is a collection of named,typed values

- *analogous to a `struct` in C*

Information on how to interpret the bytes as typed values

- *will be contained in schema data in DBMS catalogue*
- *may be stored in the header for the data file*
- *may be stored partly in the record and partly in the schema*

For variable-length records, some formatting info ...

- *must be stored in the record or in the page directory*

... Converting Records to Tuples

62/71

DBMSs typically define a fixed set of field types, e.g.

`DATE`, `FLOAT`, `INTEGER`, `NUMBER(n)`, `VARCHAR(n)`, ...

This determines implementation-level data types:

DATE	time_t
FLOAT	float,double
INTEGER	int,long
NUMBER(*n*)	int[] (?)
VARCHAR(*n*)	char[]

... Converting Records to Tuples

63/71

A `Tuple` can be defined as

- *a list of field descriptors for a record instance*
  (where a `FieldDesc` gives (offset,length,type) information)
- *along with a reference to the `Record` data*

```
typedef struct {
    ushort    nfields;  // # fields
    FieldDesc fields[]; // field descriptions
    Record    data;
} Tuple;
```

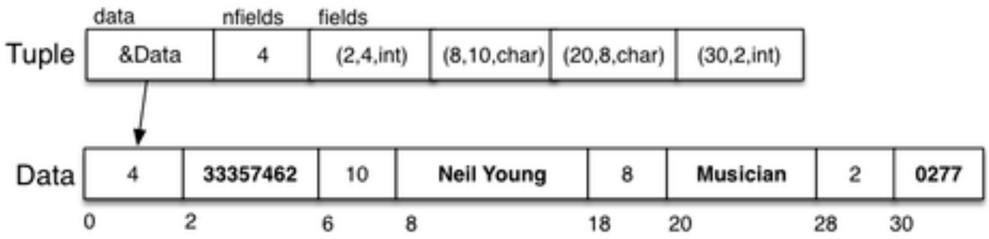
Fields are derived from relation descriptor + record instance data.

... Converting Records to Tuples

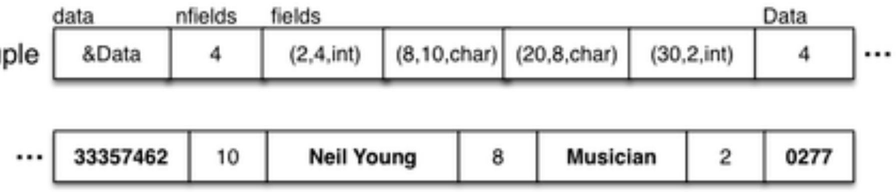
64/71

The `data` field could be either

- *a pointer to byte-chunk stored elsewhere in memory*



- *data itself appended to `struct`* *(used widely in PostgreSQL)*



PostgreSQL Tuples

65/71

Definitions: **`include/postgres.h`**, **`include/access/*tup*.h`**

Functions: **`backend/access/common/*tup*.c`**

- *e.g.* **`HeapTuple heap_form_tuple(desc, values[], isnull[])`**
- *e.g.* **`heap_deform_tuple(tuple, desc, values[], isnull[])`**

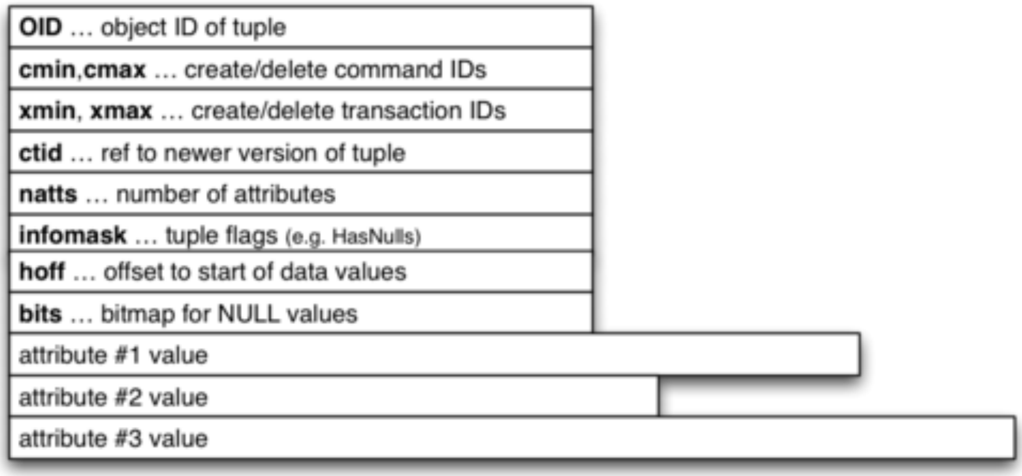
PostgreSQL defines tuples via:

- *a contiguous chunk of memory*
- *starting with a header giving e.g. #fields, nulls*
- *followed by the data values (as sequence of `Datum`)*

... PostgreSQL Tuples

66/71

Tuple structure:



... PostgreSQL Tuples

67/71

Tuple-related data types:

```
// representation of a data value
typedef uintptr_t Datum;
```

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*

... PostgreSQL Tuples

68/71

Tuple-related data types: (cont)

```
typedef struct HeapTupleData
{
    uint32          t_len;   // length of *t_data
    ItemPointerData t_self;  // SelfItemPointer
    Oid         t_tableOid;  // table the tuple came from
    HeapTupleHeader t_data;  // tuple header and data
} HeapTupleData;
```

PostgreSQL allocates a single block of data for tuple

- *containing the above struct, followed by data `byte[]`*
- *no explicit field for data, it comes after bitmap* (see next)

... PostgreSQL Tuples

69/71

Tuple-related data types: (cont)

```
typedef struct HeapTupleHeaderData // simplified
{
    HeapTupleFields t_heap;
    ItemPointerData t_ctid;      // TID of this tuple or newer version
    uint16          t_infomask2; // number of attributes + flags
    uint16          t_infomask;  // flags e.g. has_null, has_varwidth
```

```
    uint8          t_hoff;      // sizeof header incl. bitmap+padding
    // above is fixed size (23 bytes) for all heap tuples
    bits8          t_bits[1];   // bitmap of NULLs, variable length
    // actual data follows at end of struct
} HeapTupleHeaderData;
```

... PostgreSQL Tuples

70/71

Tuple-related data types: (cont)

```
typedef struct HeapTupleFields  // simplified
{
    TransactionId t_xmin;  // inserting xact ID
    TransactionId t_xmax;  // deleting or locking xact ID
    CommandId     t_cid;   // inserting/deleting command ID
} HeapTupleFields;
```

Note that not all system fields from stored tuple appear

- *both `xmin`/`xmax` are stored, but only one of `cmin`/`cmax`*

... PostgreSQL Tuples

71/71

Operations on Tuples:

```
// create Tuple from values
HeapTuple
heap_form_tuple(TupleDesc tupDesc, Datum *values, bool *isnull)

// return Datum given Tuple, attr and descriptor
//   sets isnull to true if value is NULL
#define heap_getattr(tup, attnum, tupleDesc, isnull) ...

// returns true if attribute has no value
bool heap_attisnull(HeapTuple tup, int attnum) ...

// produce a modified tuple from an existing one
HeapTuple
heap_modify_tuple(HeapTuple tuple, TupleDesc tupleDesc,
                  Datum *replValues, bool *replIsnull,
                  bool *doReplace)
```

Produced: 14 Mar 2016