Week 03 Lectures

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

Buffer pool consists of:

`BufferDescriptors`

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

`BufferBlocks`

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

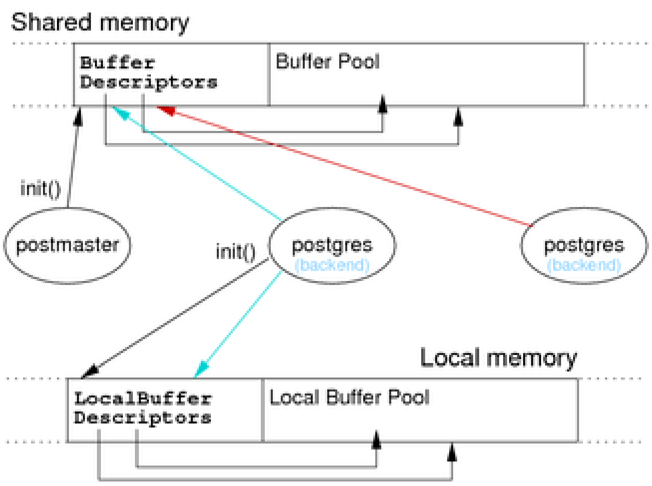
`Buffer` = index values in above arrays

- 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



... PostgreSQL Buffer Manager

`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


```
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

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

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

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

9/95

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

10/95

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 1: PostgreSQL Buffer Pool

11/95

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

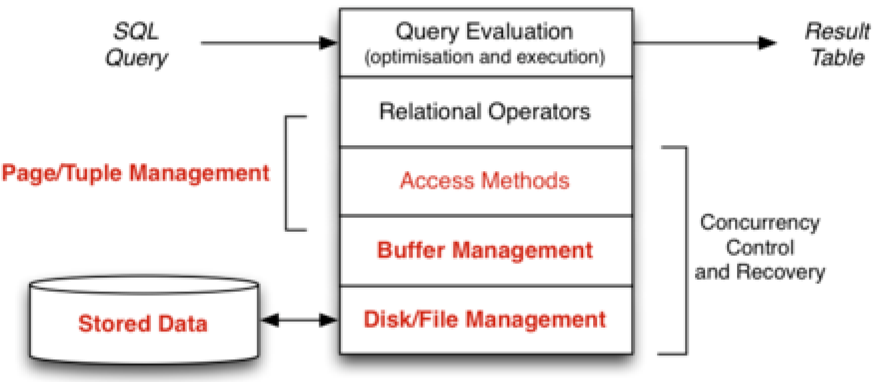
```
(tag, usage_count, refcount, freeNext)
```

Assume `freeList` and `nextVictim` global variables.

Pages

Page/Tuple Management

13/95



Pages

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

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 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

Factors affecting `Page` formats:

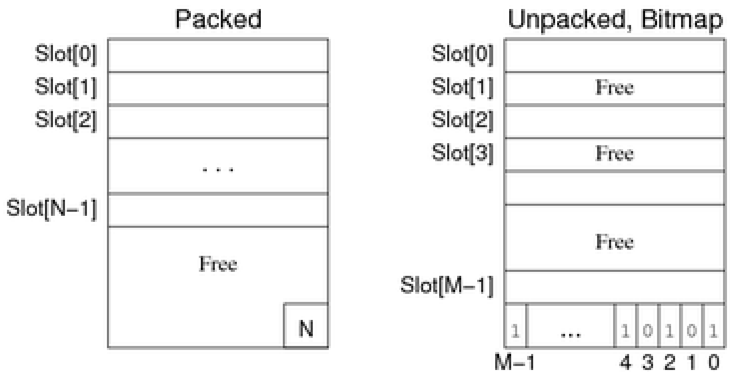
- determined by record size flexibility (fixed, variable)
- how free space within `Page` is managed
- whether some data is stored outside `Page`
  - does `Page` have an associated overflow chain?
  - are large data values stored elsewhere? (e.g. TOAST)
  - can one tuple span multiple `Page`s?

Implementation of `Page` operations critically depends on format.

... Page Formats

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

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



Exercise 2: Fixed-length Records

18/95

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 3: Inserting/Deleting Fixed-length Records

19/95

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

20/95

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

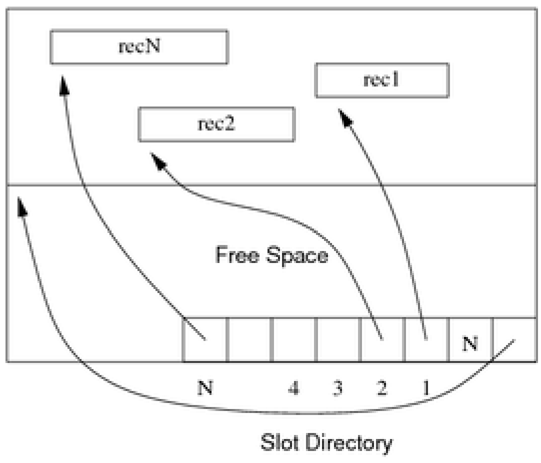
- location of tuple within page can change, tuple index does not change

---

... Page Formats

21/95

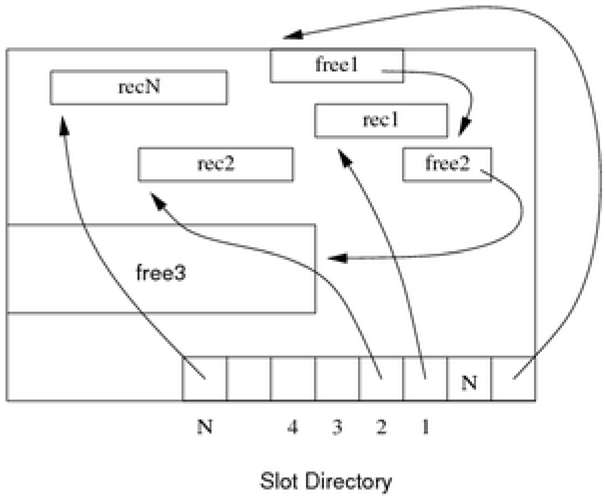
Compacted free space:



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

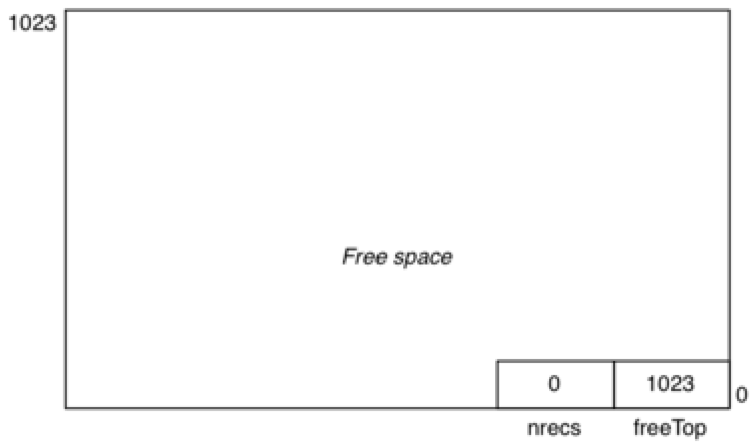
... Page Formats

Fragmented free space:



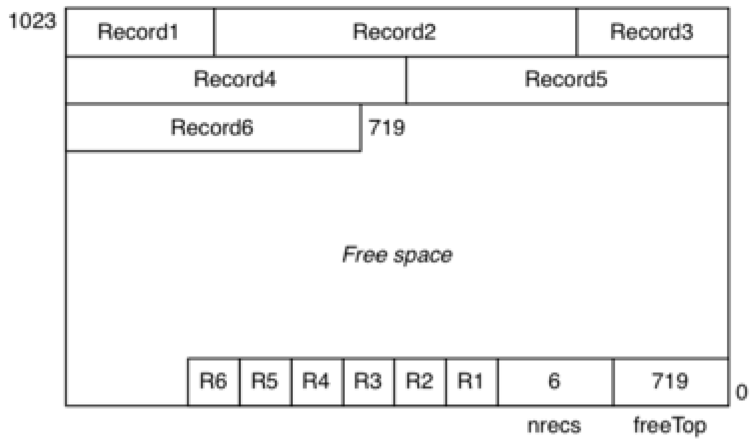
... Page Formats

Initial page state (compacted free space) ...



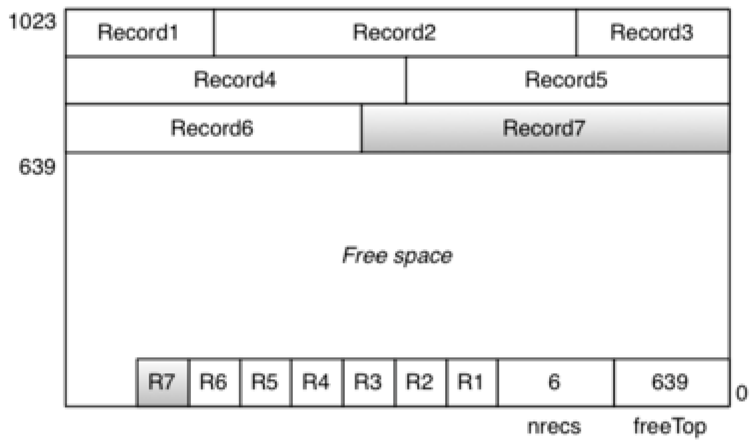
... Page Formats

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



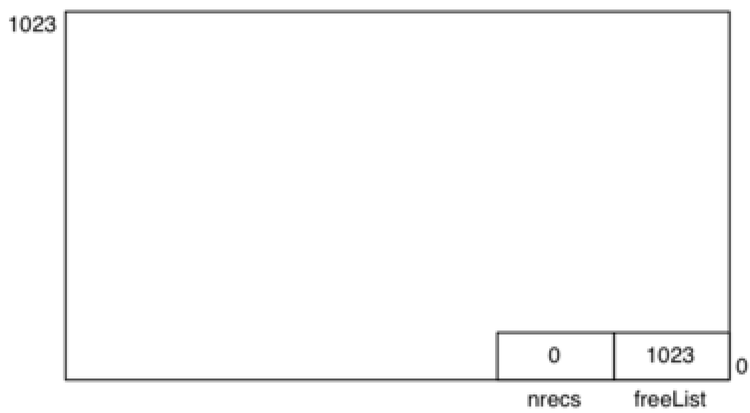
... Page Formats

After inserting record 7 (80 bytes) ...



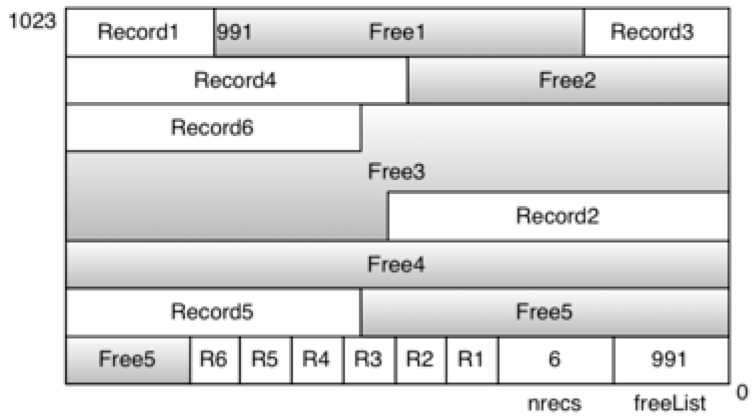
... Page Formats

Initial page state (fragmented free space) ...



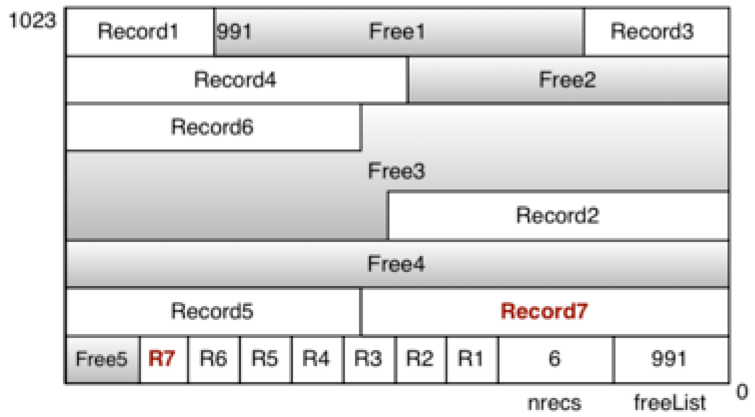
... Page Formats

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



... Page Formats

After inserting record 7 (80 bytes) ...



Exercise 4: Inserting Variable-length Records

For both of the following page formats

1. variable-length records, with compacted free space
2. variable-length records, with fragmented free space

implement the `insert()` function.

Use the above page format, but also assume:

- page size is 1024 bytes
- tuples start on 4-byte boundaries
- references into page are all 8-bits (1 byte) long
- a function `recSize(r)` gives size in bytes

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 5: Space Utilisation

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

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

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"

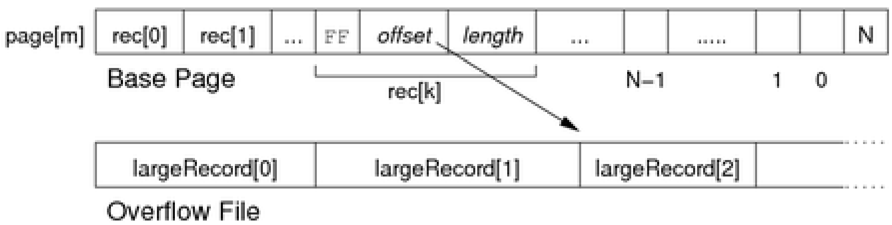
If file organisation determines record placement (e.g. hashed file)

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

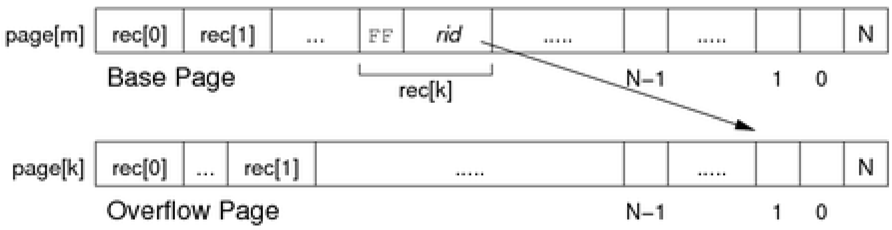
With overflow pages, *rid* structure may need modifying *(rel,page,ovfl,rec)*

... Overflows

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

35/95

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)

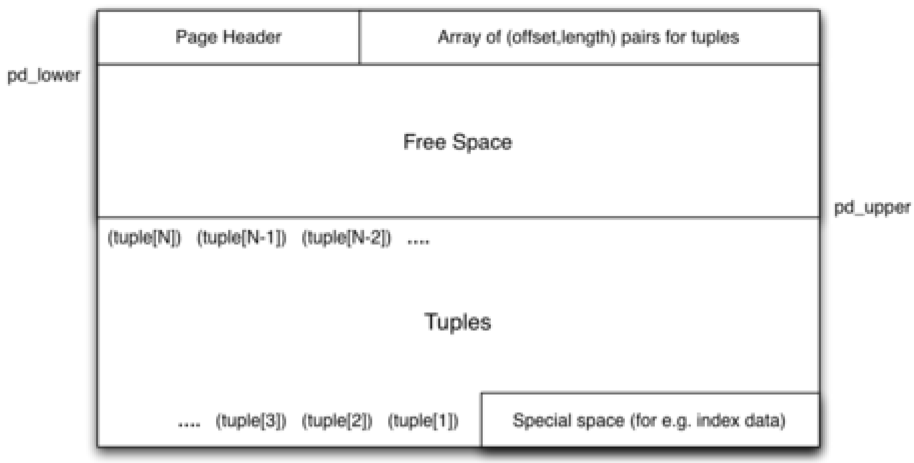
Large data items are stored in separate (TOAST) files (implicit)

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

... PostgreSQL Page Representation

36/95

PostgreSQL page layout:



... PostgreSQL Page Representation

37/95

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

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

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

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 6: PostgreSQL Pages

Draw diagrams of a PostgreSQL heap page

- when it is initially empty
- after three tuples have been inserted
  with lengths of 60, 80, and 70 bytes
- after the 80 byte tuple is deleted (but before vacuuming)
- after a new 50 byte tuple is added

Show the values in the tuple header.

Assume that there is no special space in the page.

Tuples

Tuples

Each *page* contains a collection of *tuples*

[Diagram:Pics/storage/page-tuples-small.png]

What do tuples contain? How are they structured internally?

Records vs Tuples

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

```
create table Employee (
   id   integer primary key,
   name varchar(20),
   job  varchar(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, e.g.

[Diagram:Pics/storage/tuple-bytes-small.png]

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

Operations on Records

Common operation one records ... access record via `RecordId`:

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

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

Gives a sequence of bytes, which needs to be interpreted, e.g.

```
Relation rel = ... // relation schema
Record r = get_record(rid)
Tuple t = makeTuple(rel,r)
```

Once we have a tuple, we can access individual attributes/fields

... Operations on Records

Other operations on records (via their `RecordId`) ...

`update_record(rid,rec)`

- modifies a record "in place" (replaced by new rec)
- note: PostgreSQL marks old record as "obsolete", creates new modified record

`rid = insert_record(pid,rec)`

- insert record into specified page, returning `RecordId` 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

47/95

`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. `int x = getIntField(t,1)`, `char *s = 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

48/95

`Tuple get_tuple(RecordId rid)`

- fetch the tuple specified by `rid`; return reference to `Tuple`
- used for access via an index, where index entries are (key,`rid`)

`Tuple get_tuple_from_page(Page p, int rno)`

- get the `rno`'th tuple from an already-buffered page
- called during a scan, once we have loaded relevant page
- used to implement `for each tuple t in page p`

... Operations for Access Methods

49/95

Access methods typically involve *iterators*, e.g.

`Scan s = start_scan(Rel r, ...)`

- commence a scan of relation `r`
- `Scan` may include condition to implement `WHERE`-clause
- `Scan` holds data on progress through file (e.g. current page)

`Tuple next_tuple(Scan s)`

- return `Tuple` immediately following last accessed one
- returns `NULL` if no more `Tuples` left in the relation

Example Query

50/95

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;  // current tuple
while ((t = next_tuple(s)) != NULL)
```

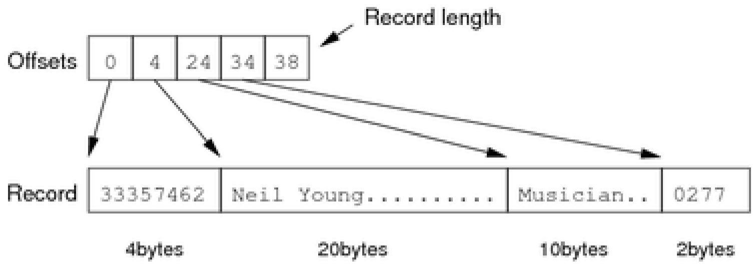
```
{
   char *name = getStrField(t,2);
   printf("%s\n", name);
}
```

Fixed-length Records

51/95

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

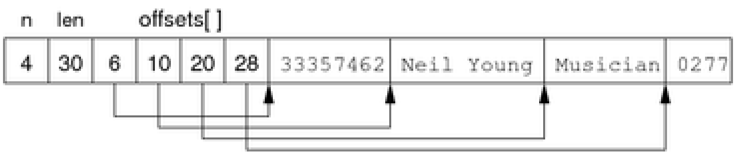
52/95

Some encoding schemes for variable-length records:

- Prefix each field by length

- Terminate fields by delimiter

- Array of offsets



Converting Records to Tuples

53/95

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

54/95

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

55/95

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

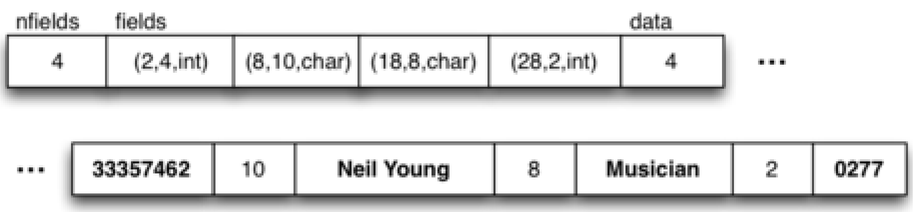
56/95

The `data` field could be either

- a pointer to byte-chunk stored elsewhere in memory

  [Diagram:Pics/storage/rec8-small.png]

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



PostgreSQL Tuples

57/95

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

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

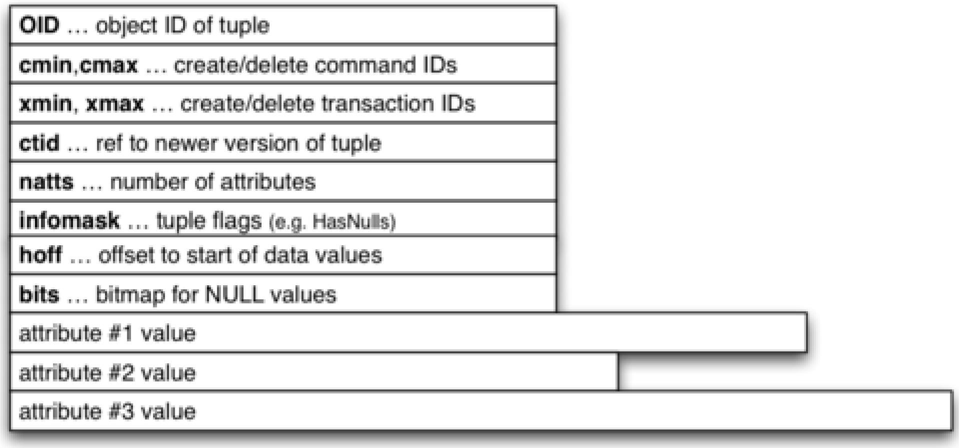
- e.g. **`HeapTuple heap_form_tuple(desc, values[], isnull[])`**
- e.g. **`heap_deform_tuple(tuple, desc, values[], isnull[])`**

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

Tuple structure:



... PostgreSQL Tuples

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

Tuple-related data types: (cont)

```
// TupleDesc: schema-related information for HeapTuples

typedef struct tupleDesc
{
    int         natts;          // number of attributes in the tuple
    Form_pg_attribute *attrs;
    // attrs[N] is a pointer to description of attribute N+1
    TupleConstr *constr;        // constraints, or NULL if none
    Oid         tdtypeid;       // composite type ID for tuple type
    int32       tdtypmod;       // typmod for tuple type
    bool        tdhasoid;       // does tuple have oid attribute?
    int         tdrefcount;     // reference count (-1 if not counting)
} *TupleDesc;
```

... PostgreSQL Tuples

Tuple-related data types: (cont)

```
typedef HeapTupleData *HeapTuple;

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

Tuple-related data types: (cont)

```
typedef struct HeapTupleHeaderData // simplified
{
    HeapTupleFields t_heap;
    ItemPointerData t_ctid;      // TID of this tuple or newer version
    uint16          natts;       // number of attributes
    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

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

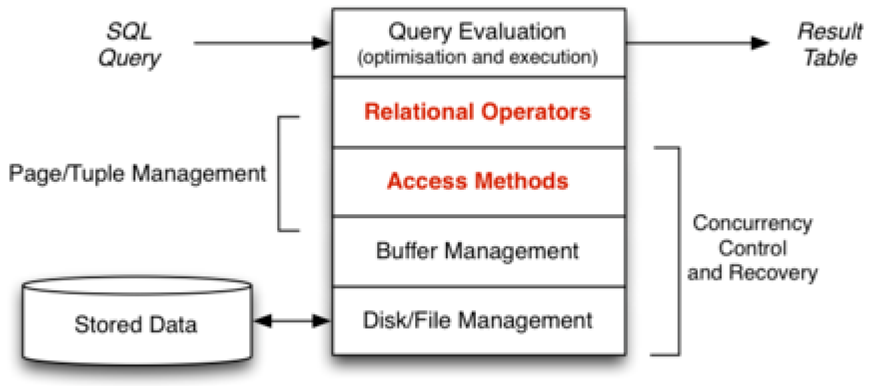
Operations on Tuples:

```
// 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)
```

Implementing Relational Operations

DBMS Architecture (revisited)

Implementation of relational operations in DBMS:



Relational Operations

DBMS core = relational engine, with implementations of

- selection, projection, join, set operations
- scanning, sorting, grouping, aggregation, ...

In this part of the course:

- examine methods for implementing each operation
- develop cost models for each implementation
- characterise when each method is most effective

Terminology reminder:

- 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

... Relational Operations

Two "dimensions of variation":

- which relational operation (e.g. Sel, Proj, Join, Sort, ...)
- which access-method (e.g. file struct: heap, indexed, hashed, ...)

Each *query method* involves an operator and a file structure:

- e.g. primary-key selection on hashed file
- e.g. primary-key selection on indexed file
- e.g. join on ordered heap files (sort-merge join)
- e.g. join on hashed files (hash join)
- e.g. two-dimensional range query on R-tree indexed file

As well as query costs, consider update costs (insert/delete).

... Relational Operations

SQL vs DBMS engine

- `select ... from R where C`
  - find relevant tuples (satisfying C) in file(s) of R
- `insert into R values(...)`
  - place new tuple in some page of a file of R
- `delete from R where C`
  - find relevant tuples and "remove" from file(s) of R
- `update R set ... where C`
  - find relevant tuples in file(s) of R and "change" them

Cost Models

Cost Models

An important aspect of this course is

- analysis of cost of various query methods

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

... Cost Models

Since *time cost* is affected by many factors

- speed of i/o devices (fast/slow disk, SSD)
- load on machine

we do not consider time cost in our analyses.

For comparing methods, *page cost* is better

- identifies workload imposed by method
- BUT is clearly affected by buffering

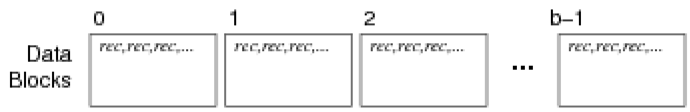
Trying to estimate costs with multiple concurrent ops *and* buffering is difficult!

Addtional assumption: every page request leads to some i/o

... Cost Models

In developing cost models, we also assume:

- 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
- the tuples which answer query q are contained in b_q pages
- data is transferred disk↔memory in whole pages
- cost of disk↔memory transfer $T_{r/w}$ is very high



... Cost Models

Our cost models are "rough" (based on assumptions)

But do give an *O(x)* feel for how expensive operations are.

Example "rough" estimation: how many piano tuners in Sydney?

- Sydney has ≅ 4 000 000 people
- Average household size ≅ 3 ∴ 1 300 000 households
- Let's say that 1 in 10 households owns a piano
- Therefore there are ≅ 130 000 pianos
- Say people get their piano tuned every 2 years (on average)
- Say a tuner can do 2/day, 250 working-days/year
- Therefore 1 tuner can do 500 pianos per year
- Therefore Sydney would need ≅ 130000/2/500 = 130 tuners

Actual number of tuners in Yellow Pages = 120

Example borrowed from Alan Fekete at Sydney University.

Query Types

75/95

Type	SQL	RelAlg	a.k.a.
Scan	`select * from R`	*R*	-
Proj	`select` *x,y* `from R`	*Proj[x,y]R*	-
Sort	`select * from R order by` *x*	*Sort[x]R*	*ord*
Sel_1	`select * from R where id =` *k*	*Sel[id=k]R*	*one*
Sel_n	`select * from R where a =` *k*	*Sel[a=k]R*	-
$Join_1$	`select * from R,S where R.id = S.r`	*R Join[id=r] S*	-

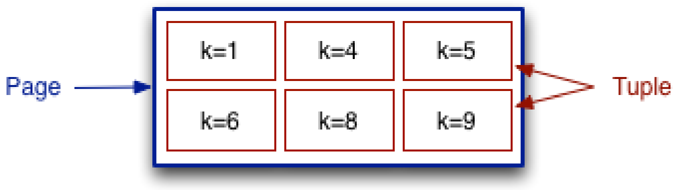
Different query classes exhibit different query processing behaviours.

Example File Structures

76/95

When describing file structures

- use a large box to represent a *page*
- use either a small box or tup_i (or rec_i) to represent a *tuple*
- sometimes refer to tuples via their *key*
  - mostly, *key* corresponds to the notion of "primary key"
  - sometimes, *key* means "search key" in selection condition



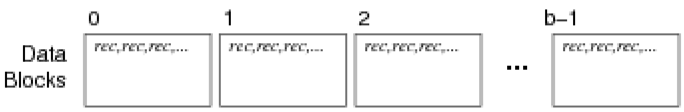
... Example File Structures

77/95

Consider three simple file structures:

- *heap file* ... tuples added to any page which has space
- *sorted file* ... tuples arranged in file in key order
- *hash file* ... tuples placed in pages using hash function

All files are composed of *b* primary blocks/pages



Some records in each page may be marked as "deleted".

Exercise 7: Operation Costs

78/95

For each of the following file structures

- determine #page-reads + #page-writes for each operation

You can assume the existence of a file header containing

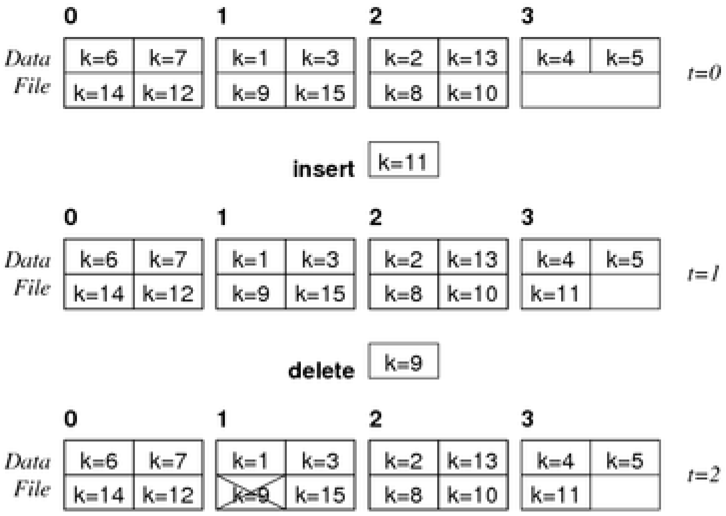
- values for *r*, *R*, *b*, *B*, *c*
- index of first page with free space (and a free list)

Assume also

- each page contains a header and directory as well as tuples
- no buffering (worst case scenario)

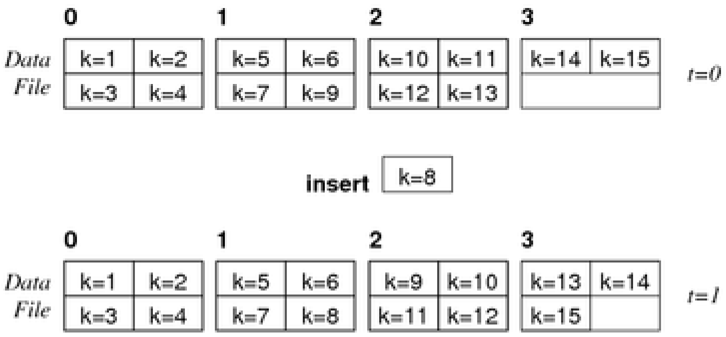
Operation Costs Example

Heap file with *b = 4, c = 4*:



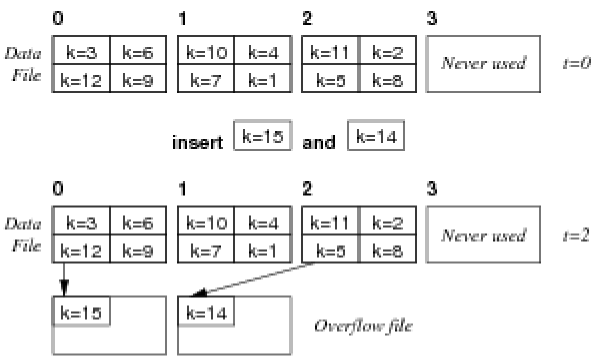
... Operation Costs Example

Sorted file with *b = 4, c = 4*:



... Operation Costs Example

Hashed file with *b = 3, c = 4, h(k) = k%3*



Scanning

Scanning

Consider the query:

```
select * from Rel;
```

Operational view:

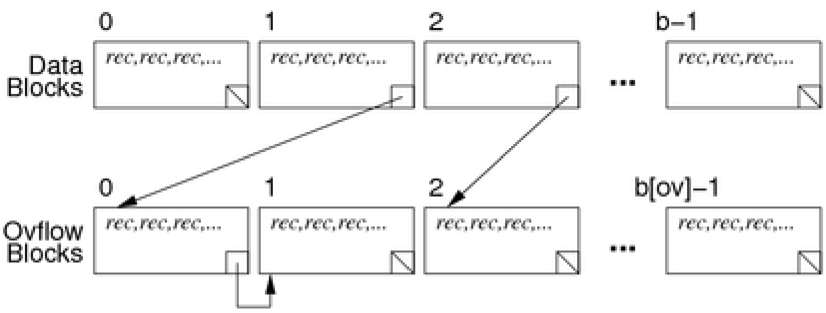
```
for each page P in file of relation Rel {
   for each tuple t in page P {
      add tuple t to result set
   }
}
```

Cost: read every data page once

Time Cost = $b.T_r$,   *Page Cost* = b

... Scanning

Scan implementation when file has overflow pages, e.g.



... Scanning

In this case, the implementation changes to:

```
for each page P in file of relation T {
    for each tuple t in page P {
        add tuple t to result set
    }
    for each overflow page V of page P {
        for each tuple t in page V {
            add tuple t to result set
}   }   }
```

Cost: read each data and overflow page once

Cost = $b + b_{Ov}$

where b_{Ov} = total number of overflow pages

Selection via Scanning

Consider a *one* query like:

```
select * from Employee where id = 762288;
```

In an unordered file, search for matching tuple requires:

Guaranteed at most one answer; but could be in any page.

... Selection via Scanning

87/95

Overview of scan process:

```
for each page P in relation Employee {
    for each tuple t in page P {
        if (t.id == 762288) return t
}   }
```

Cost analysis for *one* searching in unordered file

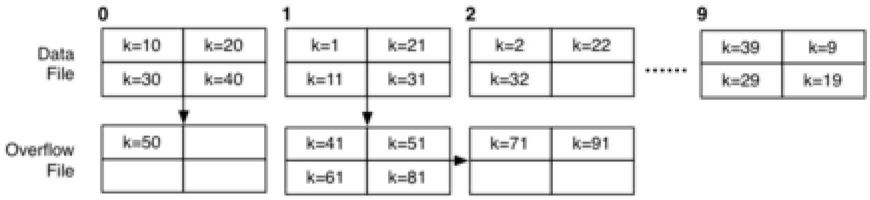
- best case: read one page, find tuple
- worst case: read all *b* pages, find in last (or don't find)
- average case: read half of the pages (*b/2*)

Page Costs: $Cost_{avg} = b/2$ $Cost_{min} = 1$ $Cost_{max} = b$

Exercise 8: Cost of Search in Hashed File

88/95

Consider the hashed file structure $b = 10, c = 4, h(k) = k\%10$



Describe how the following queries

```
select * from R where k = 51;
select * from R where k > 50;
```

might be solved in a file structure like the above ($h(k) = k\%b$).

Estimate the minimum and maximum cost (as #pages read)

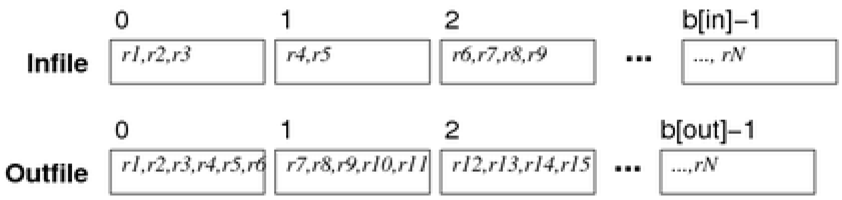
Relation Copying

89/95

Consider an SQL statement like:

```
create table T as (select * from S);
```

Effectively, copies data from one file to another.



Conceptually:

```
make empty relation T
for each tuple t in relation S {
```

```
    append tuple t to relation T
}
```

... Relation Copying

90/95

In terms of file operations:

```
File inf,outf;     // input/output file handles
int ip,op;         // input/output page numbers
int i;             // tuple number in input buf
Tuple t;           // current tuple
Buffer ibuf,obuf; // input/output file buffers

inf = openFile(relFileName("S"), READ);
outf = openFile(relFileName("T"), CREATE);
clear(obuf);
for (ip = op = 0; ip < nPages(inf); ip++) {
    ibuf = readPage(inf, ip);
    for (i = 0; i < nTuples(buf); i++) {
        t = getTuple(ibuf, i);
        addTuple(t, obuf);
        if (isFull(obuf)) {
            writePage(outf, op++, obuf);
            clear(obuf);
}   }   }
if (nTuples(obuf) > 0) writePage(outf, op, obuf);
```

Exercise 9: Cost of Relation Copy

91/95

Analyse cost for relation copying:

1. if both input and output are heap files
2. if input is sorted and output is heap file
3. if input is heap file and output is sorted

Assume b_{in} = number of pages in input file

Give cost in terms of #pages read + #pages written

Exercise 10: PostgreSQL Tuple Visibility

92/95

Due to MVCC, PostgreSQL's `getTuple(b,i)` is not so simple

- `i`th tuple in buffer `b` may be "live" or "dead" or ... ?

How does PostgreSQL recognise "dead" tuples?

What possible states might tuples have?

Assume: multiple concurrent transactions on tables.

Hint: tuple = (oid,xmin,xmax,...rest of data...)

Hint: include/access/htup.h

Hint: backend/utils/time/tqual.c

Scanning in PostgreSQL

93/95

Scanning defined in: backend/access/heap/heapam.c

Implements iterator data/operations:

- **`HeapScanDesc`** ... struct containing iteration state

- **`scan = heap_beginscan(rel,...,nkeys,keys)`**
  (uses **`initscan()`** to do half the work (shared with rescan))
- **`tup = heap_getnext(scan, direction)`**
  (uses **`heapgettup()`** to do most of the work)
- **`heap_endscan(scan)`** ... frees up `scan` struct
- **`HeapKeyTest()`** ... implements key match test

... Scanning in PostgreSQL

94/95

```
typedef struct HeapScanDescData
{
  // scan parameters
  Relation      rs_rd;        // heap relation descriptor
  Snapshot      rs_snapshot;  // snapshot ... tuple visibility
  int           rs_nkeys;     // number of scan keys
  ScanKey       rs_key;       // array of scan key descriptors
  ...
  // state set up at initscan time
  PageNumber    rs_npages;    // number of pages to scan
  PageNumber    rs_startpage; // page # to start at
  ...
  // scan current state, initally set to invalid
  HeapTupleData rs_ctup;      // current tuple in scan
  PageNumber    rs_cpage;     // current page # in scan
  Buffer        rs_cbuf;      // current buffer in scan
   ...
} HeapScanDescData;
```

Scanning in other File Structures

95/95

Above examples are for *heap* files

- simple, unordered, maybe indexed, no hashing

Other access file structures in PostgreSQL:

- **`btree`**, **`hash`**, **`gist`**, **`gin`**
- each implements:
  - startscan, getnext, endscan
  - insert, delete
  - other file-specific operators
