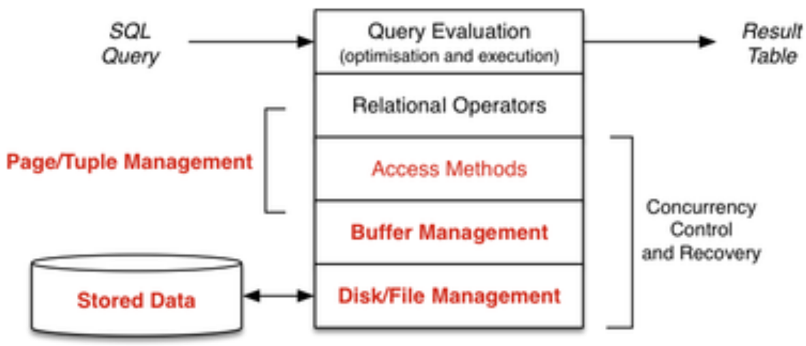
Week 03 Lectures

Pages

Page/Tuple Management

2/84



Some terminology

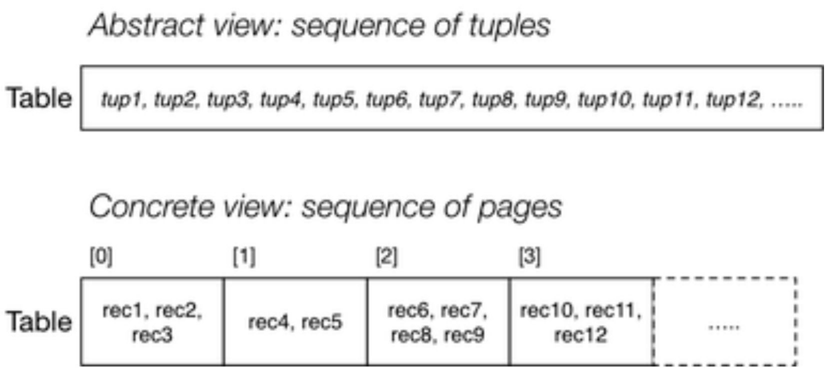
3/84

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

4/84



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

5/84

Page Formats

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

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

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 1: get_record(rel,rid)

6/84

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`

... Page Formats

7/84

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 2: Fixed-length Records (i)

8/84

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?

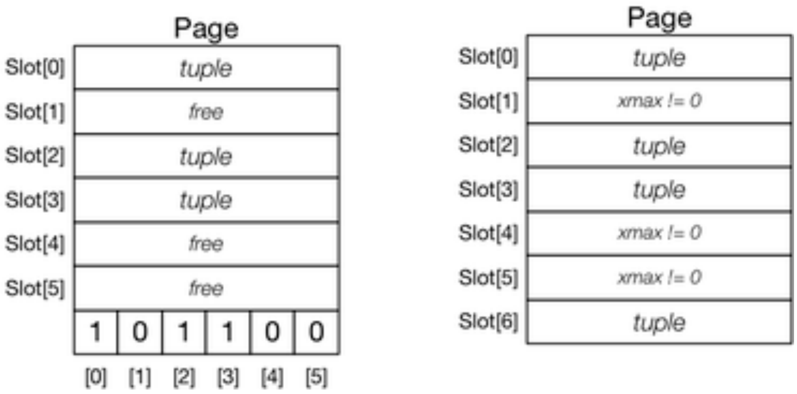
... Page Formats

9/84

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

- *insert*: place new record in first available slot

- *delete*: mark slot as free, or set *xmax*



Exercise 3: Fixed-length Records (ii)

10/84

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()`)

Page Formats

11/84

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

12/84

Possibilities for handling free-space within block:

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

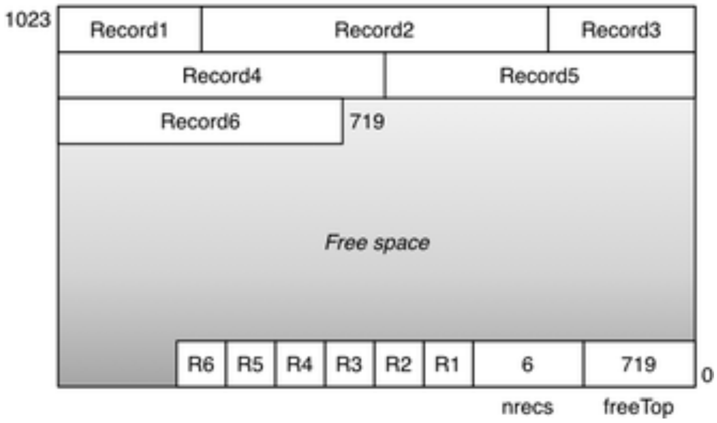
In practice, a combination is useful:

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

... Page Formats

13/84

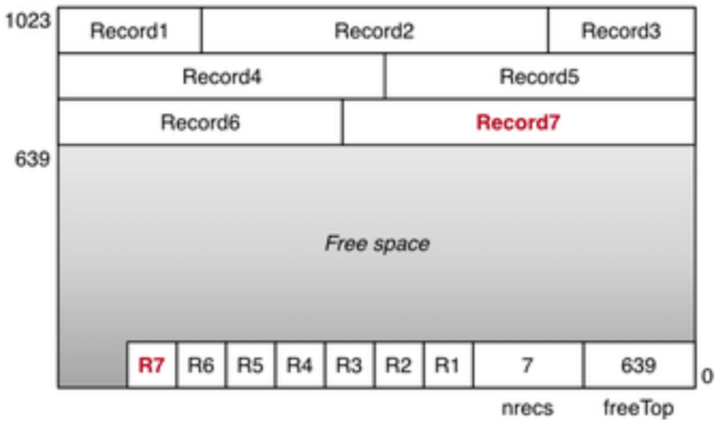
Compacted free space ... before inserting record 7



... Page Formats

14/84

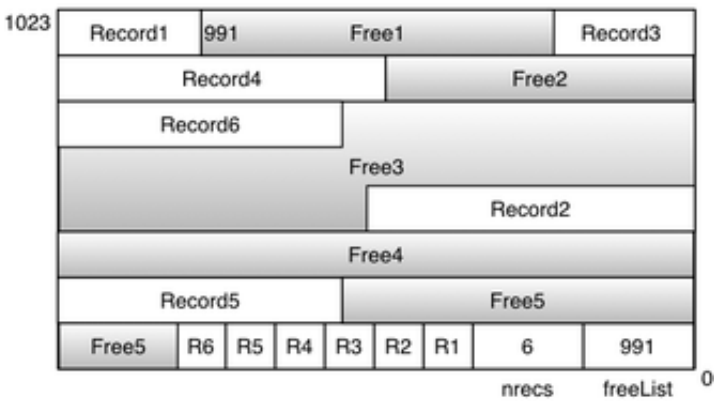
After inserting record 7 (80 bytes) ...



... Page Formats

15/84

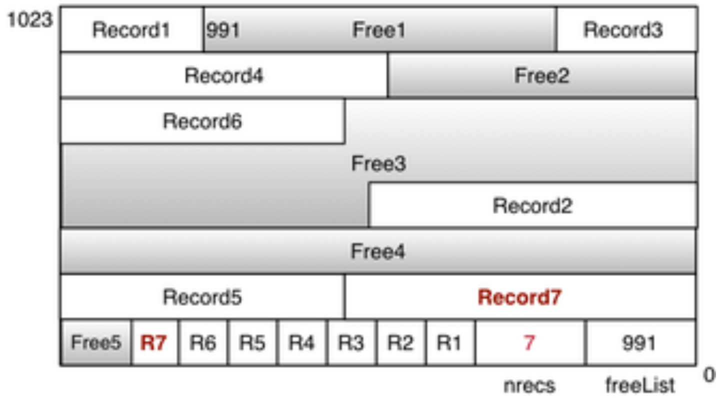
Fragmented free space ... before inserting record 7



... Page Formats

16/84

After inserting record 7 (80 bytes) ...



Exercise 4: Inserting Variable-length Records

17/84

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(rec)` gives size in bytes

Storage Utilisation

18/84

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 5: Space Utilisation

19/84

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

20/84

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

21/84

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)

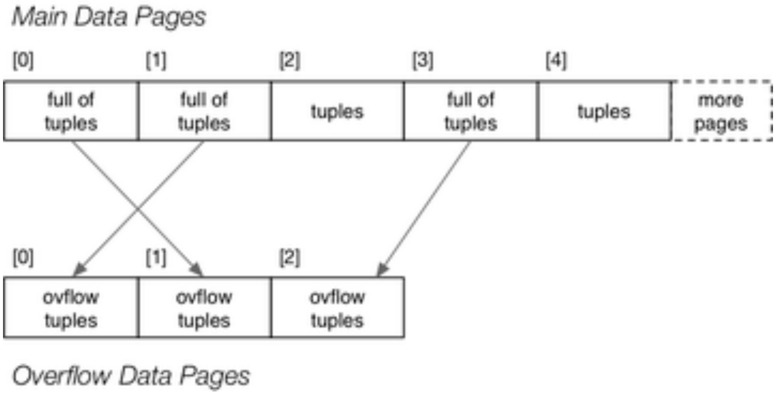
- cases (2) and (4) require an "overflow page"
- case (3) requires an "overflow file"

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

... Overflows

22/84

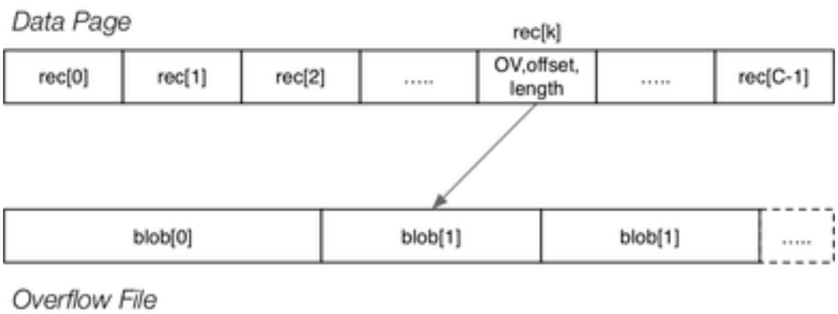
Overflow pages for full buckets in a hashed file:



... Overflows

23/84

Overflow file for very large records and BLOBs:



PostgreSQL Page Representation

24/84

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)

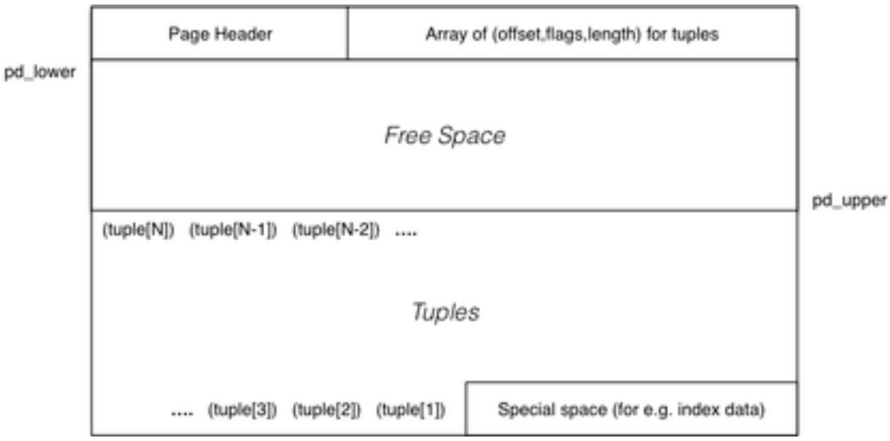
Large data items are stored in separate (TOAST) files (implicit)

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

... PostgreSQL Page Representation

25/84

PostgreSQL page layout:



... PostgreSQL Page Representation

26/84

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

27/84

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

28/84

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

29/84

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.

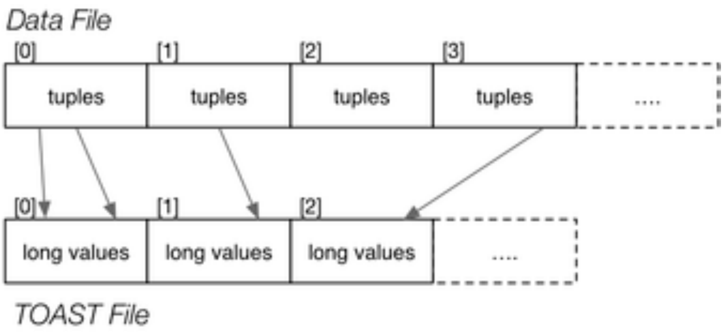
One important difference:

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

TOAST Files

30/84

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



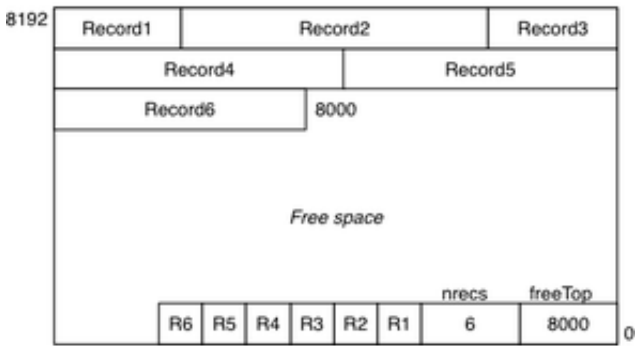
Tuples in data pages contain `rid`s for long values

TOAST = The Oversized Attribute Storage Technique

Tuples

Tuples 32/84

Each *page* contains a collection of *tuples*



What do tuples contain? How are they structured internally?

Records vs Tuples 33/84

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.

Bytes need to be interpreted relative to schema to get tuple

Operations on Records 34/84

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

```
Record get_record(Relation rel, RecordId rid) {
    (pid,tid) = rid;
    Page *buf = request_page(rel, pid);
    return get_record(buf, tid);
}
```

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 Tuples 35/84

Once we have a record, we need to interpret it as a tuple ...

`Tuple t = makeTuple(rel, rec)`

- convert record to tuple data structure for relation `rel`

Once we have a tuple, we want to examines its contents ...

`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)`

Scanning

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

`Scan s = start_scan(Relation 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

Example: simple scan of a table ...

```
select name from Employee
```

implemented as:

```
DB db = openDatabase("myDB");
Relation 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);
}
```


Exercise 6: Implement next_tuple()

Consider the following possible **`Scan`** data structure

```
typedef struct {
   Relation rel;
   Page     *curPage;  // Page buffer
   int      curPID;    // current pid
   int      curTID;    // current tid
} Scan;
```

Assume tuples are indexed 0..`nTuples(p)`

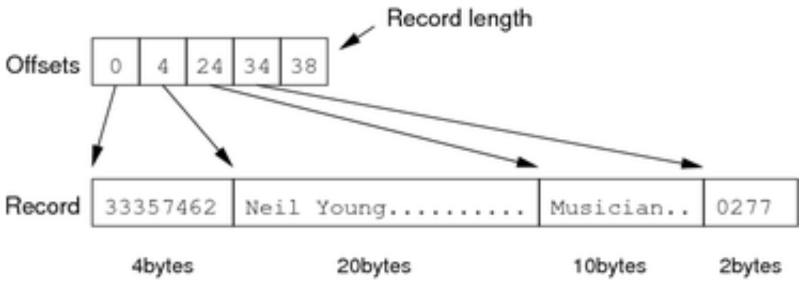
Assume pages are indexed 0..`nPages(rel)`

Implement the **`Tuple next_tuple(Scan)`** function

Fixed-length Records

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.

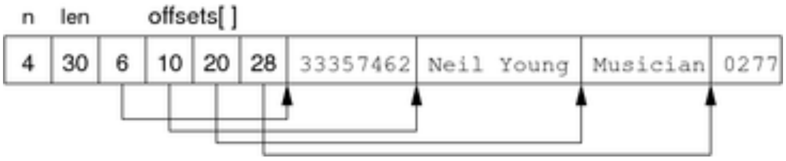
Variable-length Records

Some encoding schemes for variable-length records:

- Prefix each field by length

- Terminate fields by delimiter

- Array of offsets



Converting Records to Tuples

41/84

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 (cf. C `struct`)

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

42/84

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

43/84

A `Tuple` could 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;   // number of fields/attrs
  ushort    data_off;  // offset in struct for data
  FieldDesc fields[];  // field descriptions
  Record    data;      // pointer to record in buffer
} Tuple;
```

Fields are derived from relation descriptor + record instance data.

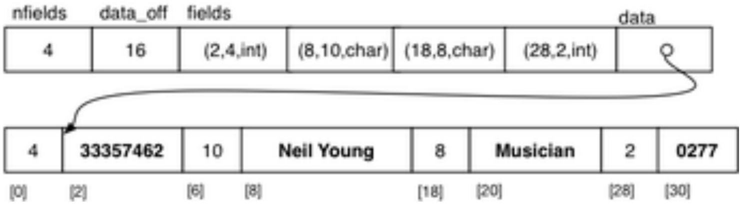
... Converting Records to Tuples

44/84

Tuple `data` could be

- a pointer to bytes stored elsewhere in memory

e.g.



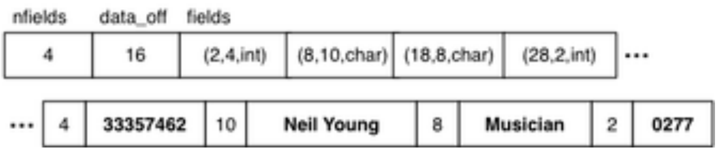
... Converting Records to Tuples

45/84

Or, tuple `data` could be ...

- appended to `Tuple struct` (used widely in PostgreSQL)

e.g.



Exercise 7: How big is a `FieldDesc`?

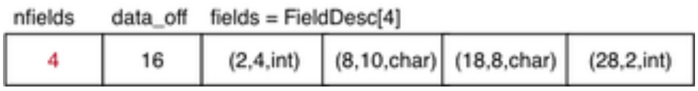
46/84

`FieldDesc` = (offset,length,type), where

- offset = offset of field within record data
- length = length (in bytes) of field
- type = data type of field

If pages are 8KB in size, how many bits are needed for each?

E.g.



PostgreSQL Tuples

47/84

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

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

- `HeapTuple heap_form_tuple(desc,values[],isnull[])`
- `heap_deform_tuple(tuple,desc,values[],isnull[])`

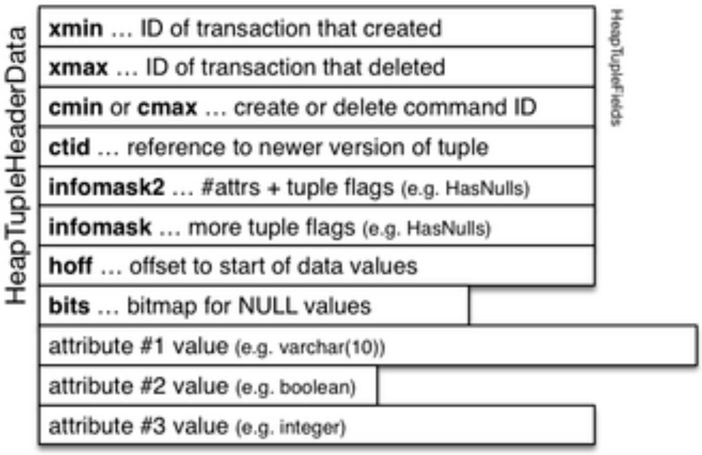
PostgreSQL defines tuples via:

- 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

48/84

Tuple structure:



... PostgreSQL Tuples

49/84

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

50/84

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

51/84

`HeapTupleData` contains information about a stored tuple

```
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;
```

`HeapTupleHeader` is a pointer to a location in a buffer

... PostgreSQL Tuples

52/84

PostgreSQL stores a single block of data for tuple

- containing a tuple header, followed by data `byte[]`

```
typedef struct HeapTupleHeaderData // simplified
{
  HeapTupleFields t_heap;
  ItemPointerData t_ctid;      // TID of this tuple or newer version
  uint16          t_infomask2; // #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
  // OID goes here if HEAP_HASOID is set in t_infomask
  // actual data follows at end of struct
} HeapTupleHeaderData;
```

... PostgreSQL Tuples

53/84

Tuple-related data types: (cont)

```
typedef struct HeapTupleFields  // simplified
{
  TransactionId t_xmin;  // inserting xact ID
  TransactionId t_xmax;  // deleting or locking xact ID
  union {
    CommandId   t_cid;   // inserting or deleting command ID
    TransactionId t_xvac;// old-style VACUUM FULL xact ID
  } t_field3;
} HeapTupleFields;
```

Note that not all system fields from stored tuple appear

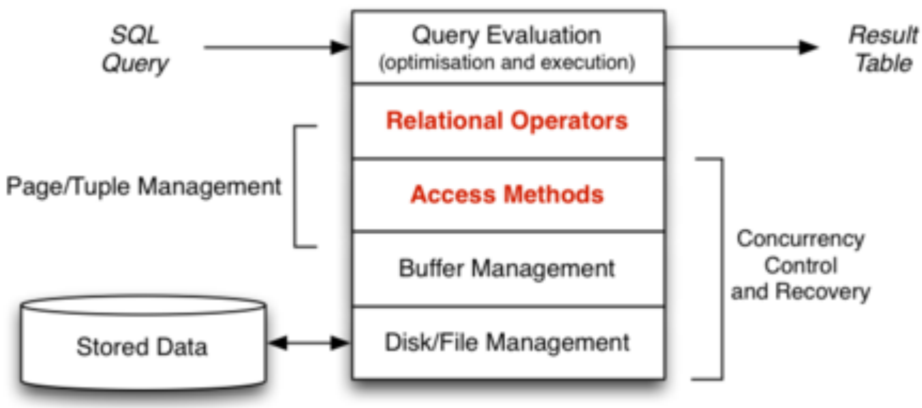
- `oid` is stored after the tuple header, if used
- both `xmin`/`xmax` are stored, but only one of `cmin`/`cmax`

Implementing Relational Operations

DBMS Architecture (revisited)

55/84

Implementation of relational operations in DBMS:



Relational Operations

56/84

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 = collection of data values under some schema ≅ record
- page = block = collection of tuples + management data = i/o unit
- relation = table ≅ file = collection of tuples

... Relational Operations

57/84

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

58/84

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

60/84

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

61/84

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

Estimating costs with multiple concurrent ops *and* buffering is difficult!!

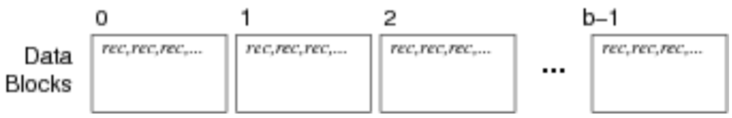
Addtional assumption: every page request leads to some i/o

... Cost Models

62/84

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

63/84

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

64/84

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*	-

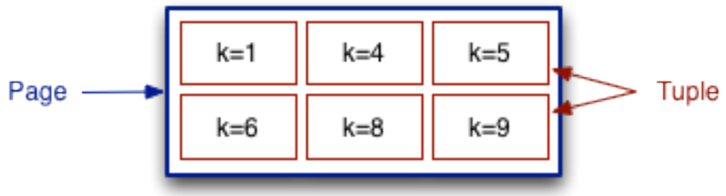
Different query classes exhibit different query processing behaviours.

Example File Structures

65/84

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



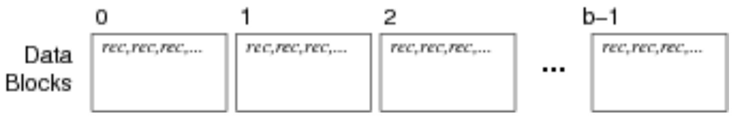
... Example File Structures

66/84

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 8: Operation Costs

67/84

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)

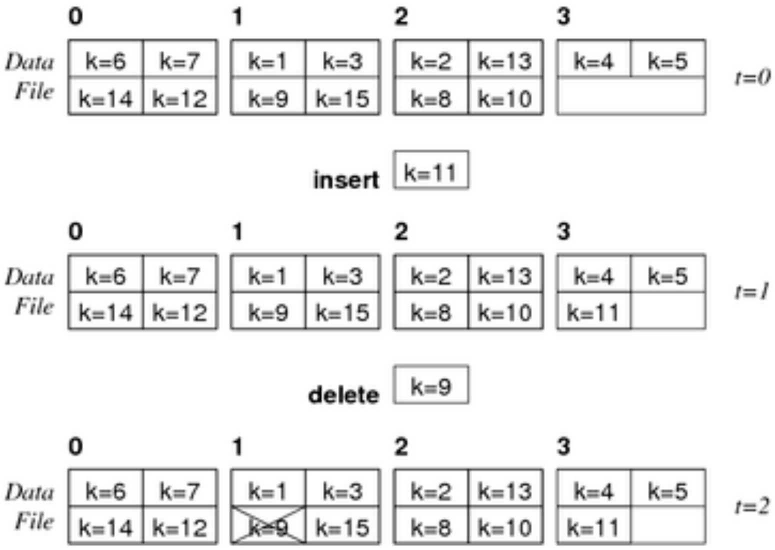
Assume also

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

Operation Costs Example

68/84

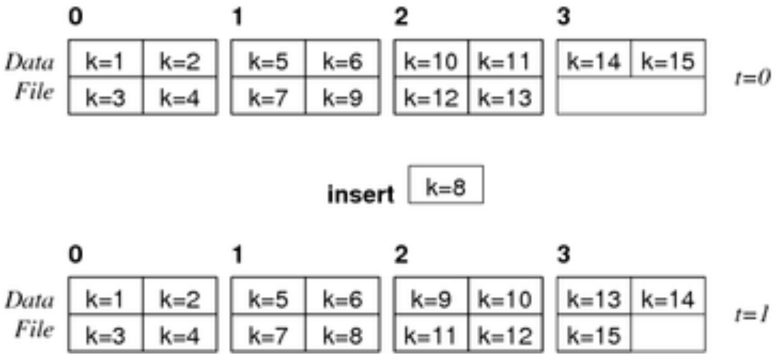
Heap file with *b = 4, c = 4*:



... Operation Costs Example

69/84

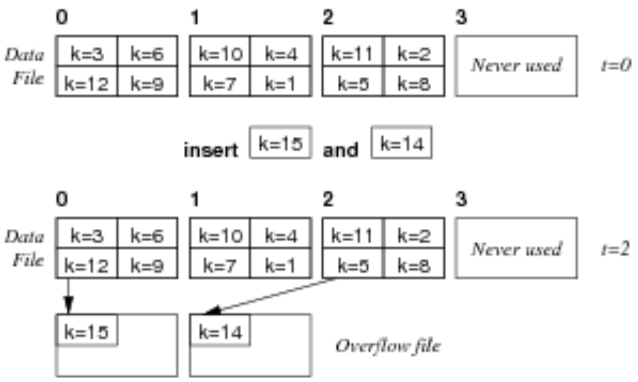
Sorted file with *b = 4, c = 4*:



... Operation Costs Example

70/84

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



Scanning

Scanning

72/84

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
   }
}
```

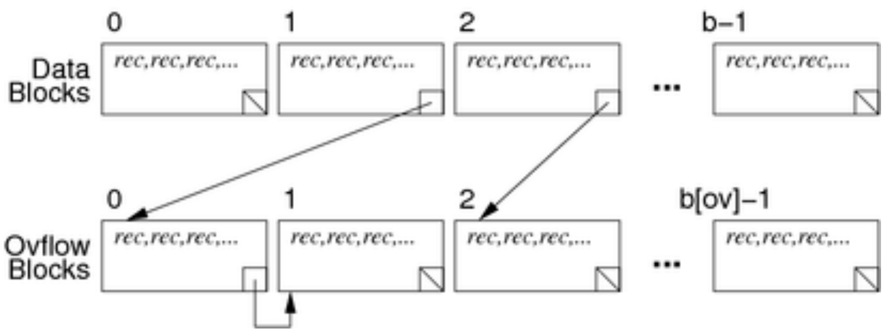
Cost: read every data page once

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

... Scanning

73/84

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



... Scanning

74/84

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

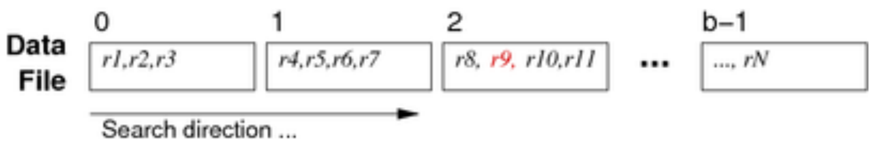
Selection via Scanning

75/84

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

76/84

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

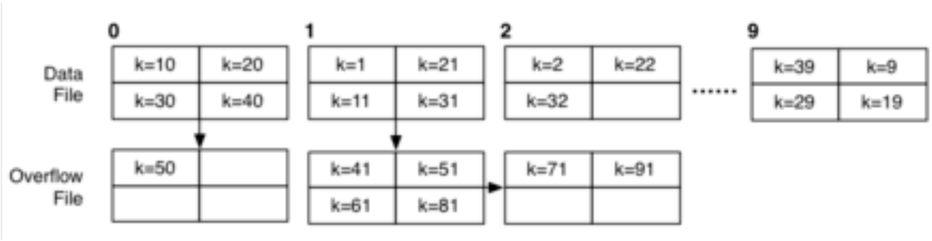
- best case: read one page, find tuple
- worst case: read all *b* pages, find in last (or don't find)
- average case: read half of the pages (*b/2*)

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

Exercise 9: Cost of Search in Hashed File

77/84

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)

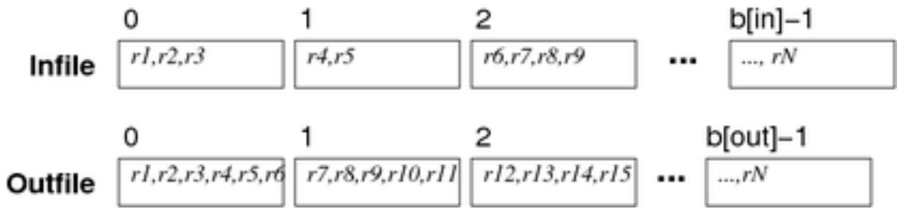
Relation Copying

78/84

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

79/84

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 10: Cost of Relation Copy

80/84

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 11: PostgreSQL Tuple Visibility

81/84

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

82/84

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

83/84

```
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

84/84

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

Produced: 20 Jun 2019