Week 04 Lectures

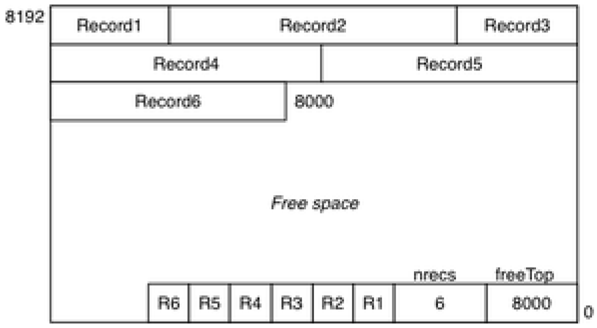
Tuples

Tuples

2/53

Tuples can be variable length or fixed length

Each *page* contains a collection of *tuples*



Tuple should be converted to records(bytes)

header

What do tuples contain? How are they structured internally?

Records vs Tuples

3/53

A *table* is defined by a *schema*, e.g.

```
create table Employee (
   id   integer primary key,
   name varchar(20) not null,
   job  varchar(10),
   dept number(4) references Dept(id)
);
```

where a schema is a collection of attributes (name,type,constraints)

Schema information (meta-data) is stored in the DB catalog

... Records vs Tuples

4/53

Here, record is how tuple stored in db

Tuple = collection of attribute values based on a schema, e.g.

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

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

01101001	11001100	01010101	00111100	10100011	01011111	01011010

Bytes need to be interpreted relative to schema to get tuple

Converting Records to Tuples

5/53

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

- representing the data values from a typed `Tuple`
- stored on disk (persistent) or in a memory buffer

A `Tuple` is a collection of named,typed values (cf. C `struct`)
Attributes

- to manipulate the values, need an "interpretable" structure

- stored in working memory, and temporary

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

... Converting Records to Tuples

6/53

Information on how to interpret bytes in a record ...

- may be contained in schema data in DBMS catalog
- may be stored in the page directory
- may be stored in the record (in a record header)
- 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
- at the least, need to know how many bytes in each value

Operations on Records

7/53

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

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

PageID = FileID + offset

TupleID = PageId + offset

Cannot use a `Record` directly; need a `Tuple`:

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

Once we have a `Tuple`, we can access individual attributes/fields

---

Operations on Tuples

8/53

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

`Tuple t = mkTuple(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 i)`

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

E.g. `int x = getIntField(t,1)`, `char *s = getStrField(t,2)`

Fixed-length Records

9/53

A possible encoding scheme for fixed-length records:

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

ield(t,2)

Page. page header

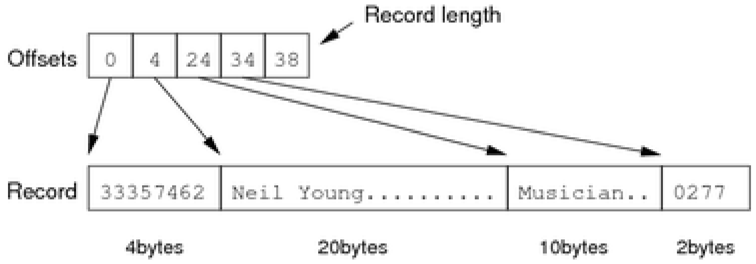
~~recode~~

records

free space

recode → record header

→ recode content



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

Variable-length Records

10/53

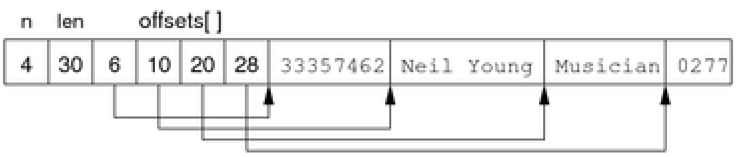
Possible encoding schemes for variable-length records:

- Prefix each field by length

- Terminate fields by delimiter 定界符

Each record include a header and a column

- Array of offsets



Data Types

11/53

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

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

This determines implementation-level data types for field values:

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

PostgreSQL allows new base types to be added

Field Descriptors

12/53

A `Tuple` could be implemented 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
```

Field <–> columns

```
   Record    data;      // pointer to record in buffer
} Tuple;
```

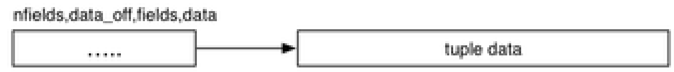
Fields are derived from relation descriptor + record instance data.

... Field Descriptors

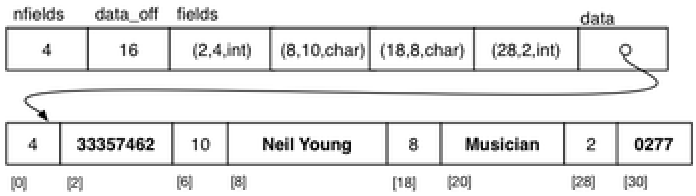
13/53

Tuple `data` could be

- a pointer to bytes stored elsewhere in memory



e.g.

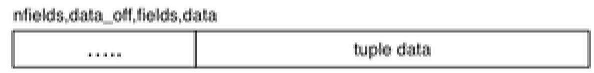


... Field Descriptors

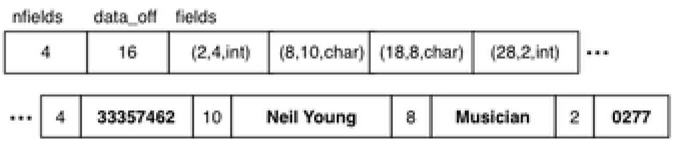
14/53

Or, tuple `data` could be ...

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



e.g.



Exercise 1: How big is a `FieldDesc`?

15/53

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

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

2 ^ 13 = 8192 > 8KB

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

13

E.g.

PostgreSQL Tuples

16/53

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[])`

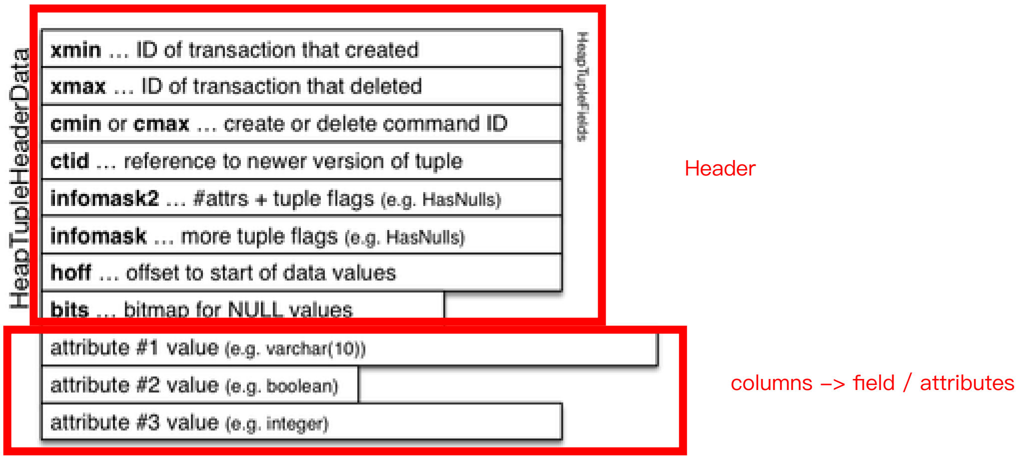
PostgreSQL implements tuples via:

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

... PostgreSQL Tuples

17/53

Tuple structure:



... PostgreSQL Tuples

18/53

Tuple-related data types: (cont)

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

typedef struct tupleDesc
{
  int          natts;       // # attributes in tuple
  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)
  TupleConstr *constr;      // constraints, or NULL if none
  FormData_pg_attribute attrs[];
  // attrs[N] is a pointer to description of attribute N+1
} *TupleDesc;
```

... PostgreSQL Tuples

19/53

Tuple-related data types: (cont)

```
// FormData_pg_attribute:
// schema-related information for one attribute

typedef struct FormData_pg_attribute
{
  Oid      attrelid;    // OID of reln containing attr
  NameData attname;     // name of attribute
  Oid      atttypid;    // OID of attribute's data type
  int16    attlen;      // attribute length
  int32    attndims;    // # dimensions if array type
  bool     attnotnull;  // can attribute have NULL value
  .....                 // and many other fields
} FormData_pg_attribute;
```

For details, see `include/catalog/pg_attribute.h`

... PostgreSQL Tuples

20/53

`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

21/53

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 newer version
  uint16          t_infomask2; // #attributes + flags
  uint16          t_infomask;  // flags e.g. has_null
  uint8           t_hoff;      // sizeof header incl. t_bits
  // above is fixed size (23 bytes) for all heap tuples
  bits8           t_bits[1];   // bitmap of NULLs, var.len.
  // OID goes here if HEAP_HASOID is set in t_infomask
  // actual data follows at end of struct
} HeapTupleHeaderData;
```

... PostgreSQL Tuples

22/53

Some of the bits in `t_infomask` ..

```
#define HEAP_HASNULL       0x0001
        /* has null attribute(s) */
#define HEAP_HASVARWIDTH   0x0002
        /* has variable-width attribute(s) */
#define HEAP_HASEXTERNAL   0x0004
        /* has external stored attribute(s) */
#define HEAP_HASOID_OLD    0x0008
        /* has an object-id field */
```

Location of `NULL`s is stored in t_bits[] array

... PostgreSQL Tuples

23/53

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`

PostgreSQL Attribute Values

24/53

Values of attributes in PostgreSQL tuples are packaged as `Datums`

```
// 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 (if large value)

... PostgreSQL Attribute Values

25/53

Attribute values can be extracted as `Datum` from `HeapTuples`

```
Datum heap_getattr(
      HeapTuple tup,      // tuple (in memory)
      int attnum,         // which attribute
      TupleDesc tupDesc,  // field descriptors
      bool *isnull        // flag to record NULL
)
```

`isnull` is set to true if value of field is `NULL`

`attnum` can be negative ... to access system attributes (e.g. OID)

For details, see `include/access/htup_details.h`

... PostgreSQL Attribute Values

26/53

Values of `Datum` objects can be manipulated via e.g.

```
// DatumGetBool:
//   Returns boolean value of a Datum.

#define DatumGetBool(X) ((bool) ((X) != 0))

// BoolGetDatum:
//   Returns Datum representation for a boolean.

#define BoolGetDatum(X) ((Datum) ((X) ? 1 : 0))
```

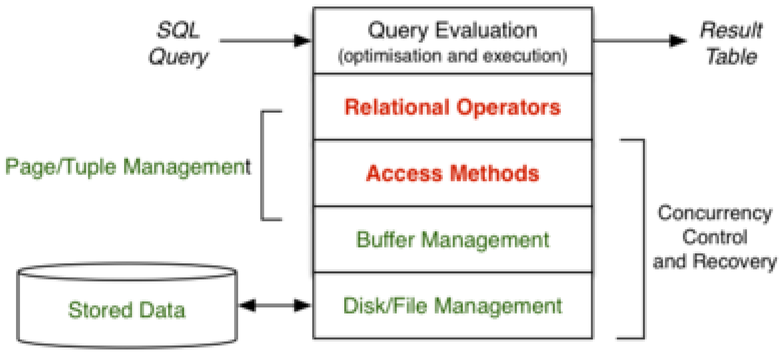
For details, see `include/postgres.h`

Implementing Relational Operations

DBMS Architecture (revisited)

28/53

Implementation of relational operations in DBMS:



29/53

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 = 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

30/53

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

31/53

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

33/53

An important aspect of this course is

- analysis of cost of various query methods

Cost can be measured in terms of

O(n) X —>>> page cost

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

1. Read >> write , output: read
2. Write >> Read, output: write
3. Read + write
4. Reading from Buffer :
   no cost

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

34/53

Since *time cost* is affected by many factors

The first time is usually slower as it is not in the cache

- 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

35/53

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

36/53

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

37/53

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`	*Sort[x]R*	*ord*

	`order by x`		
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*	-

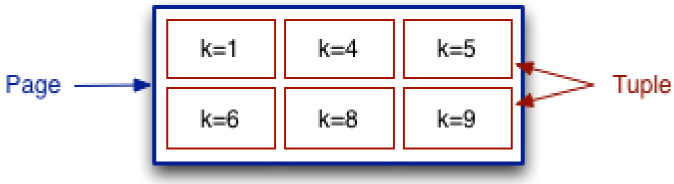
Different query classes exhibit different query processing behaviours.

Example File Structures

38/53

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

39/53

Heap file:
Best: read from the beginning , the first space is free, then the cost is (1 + 1 + header page)
Worst: read from beginning to the b th page, cost: (b_r + 1 + header page)

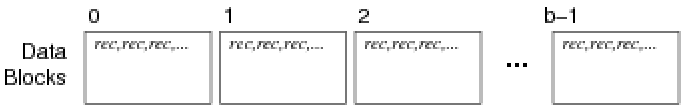
Consider three simple file structures:

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

Sorted file:
Cost: (log_2 b + 1 + header page)

Hash file:
Depend on the hash size:
For example, hash size is 3:
Then 100 is place in index 1
(100 % 3 = 1)

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



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

Exercise 2: Operation Costs

40/53

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)

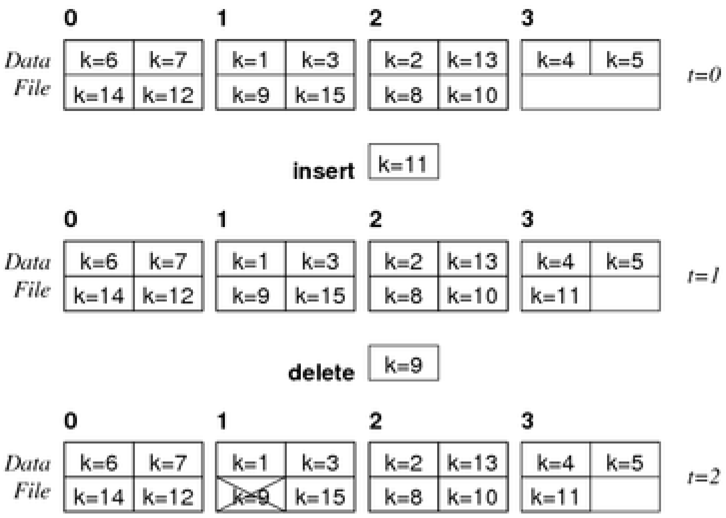
Assume also

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

Operation Costs Example

41/53

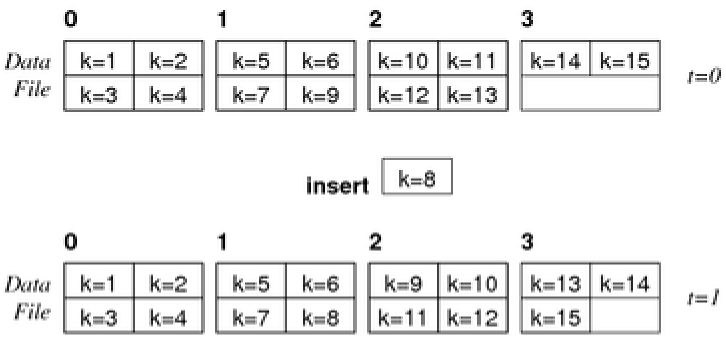
Heap file with *b = 4, c = 4*:



... Operation Costs Example

42/53

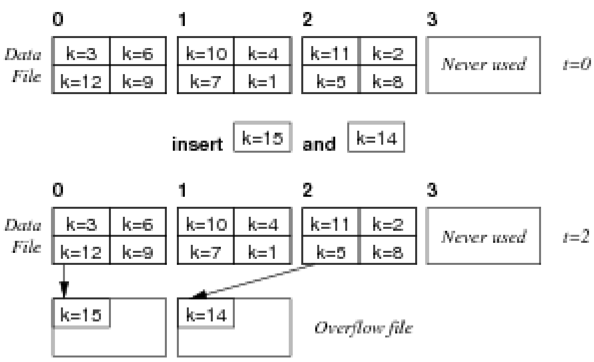
Sorted file with *b = 4, c = 4*:



... Operation Costs Example

43/53

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



Scanning

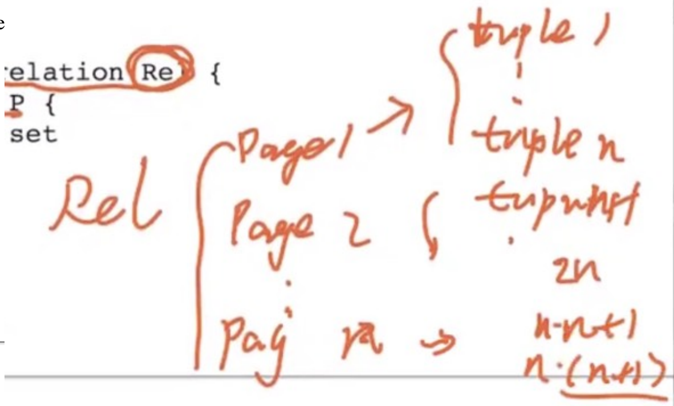
Scanning

45/53

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
   }
}
```

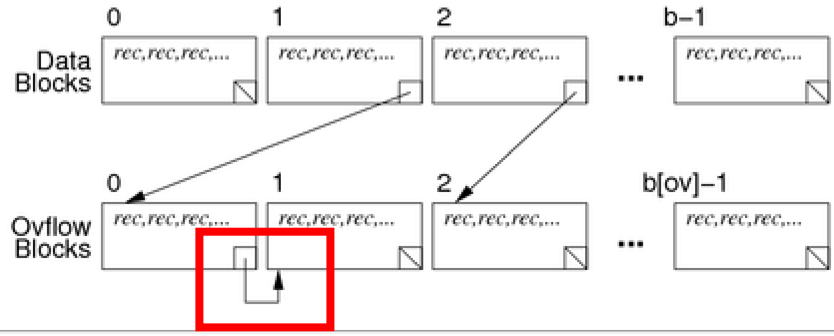
Cost: read every data page once

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

Each page is stored in a file block, but when the size of page is larger than the block size, it is overflow

... Scanning

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



... Scanning

47/53

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

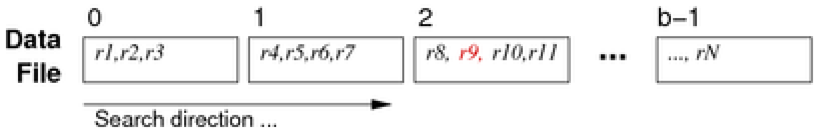
Selection via Scanning

48/53

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

49/53

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

Scanning

50/53

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

51/53

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 3: Implement next_tuple()

52/53

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

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

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

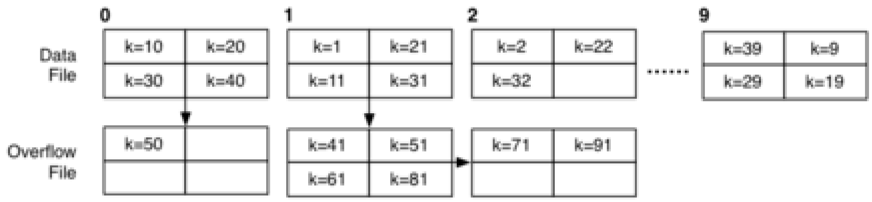
Assume pages are indexed 0..`nPages(rel)`

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

P.S. What's in a `Relation` object?

Exercise 4: Cost of Search in Hashed File

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

Produced: 9 Mar 2020