>>

Scanning

- Scanning
- Selection via Scanning
- Iterators
- Example Query
- **`next_tuple()`** Function
- Relation Copying
- Scanning in PostgreSQL
- Scanning in other File Structures

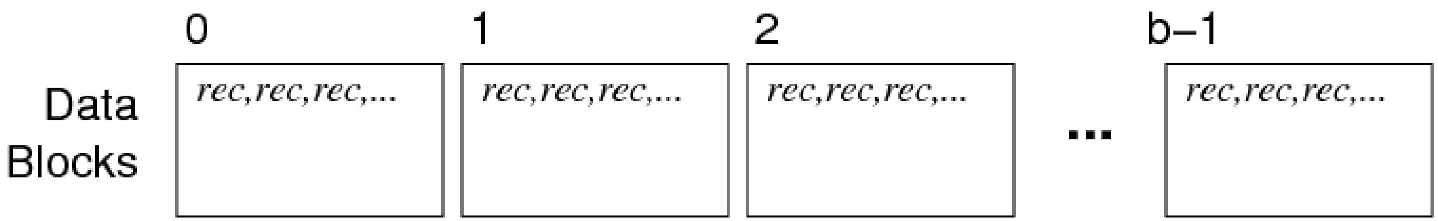
∧ >>

❖ Scanning

Consider executing the query:

```
select * from Rel;
```

where the relation has a file structure like:



This would done by a simple scan of all records/tuples.

<< ∧ >>

❖ Scanning (cont)

Abstract view of how the scan might be implemented:

```
for each tuple T in relation Rel {
    add tuple T to result set
}
```

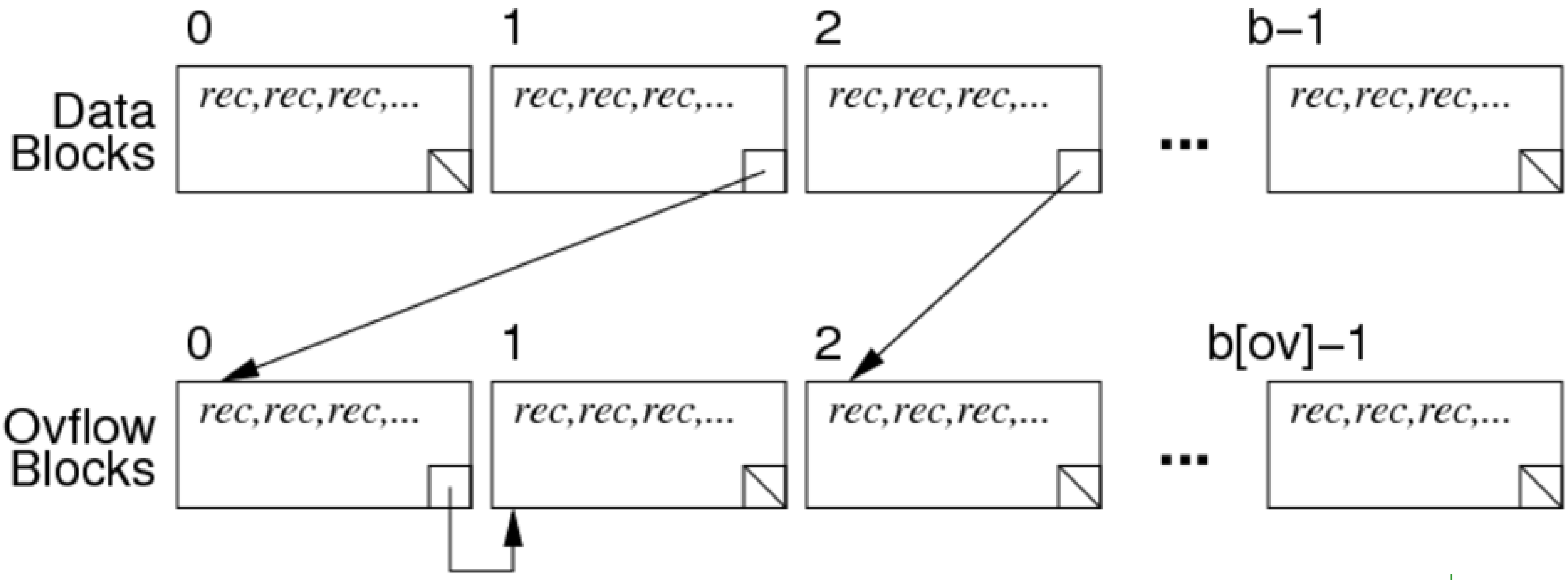
Operational view:

```
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 = b

❖ Scanning (cont)

Consider a file with overflow pages, e.g.



<< ∧ >>

❖ Scanning (cont)

In this case, the implementation changes to:

```
for each page P in data file of relation Rel {
    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 page and each overflow page once

$Cost = b + b_{Ov}$

where b_{Ov} = total number of overflow pages

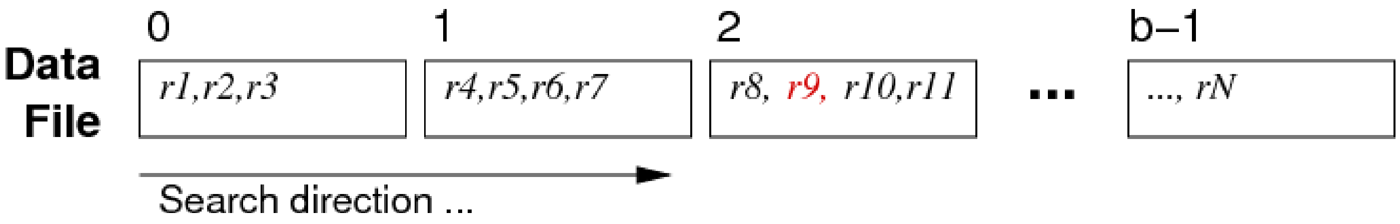
<< ∧ >>

❖ Selection via Scanning

Consider a *one* query like:

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

In an unordered file, search for matching tuple requires:



Data File

0 *r1,r2,r3*

1 *r4,r5,r6,r7*

2 *r8, r9, r10,r11*

...

b–1 *..., rN*

Search direction ...

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

<< ∧ >>

❖ Selection via Scanning (cont)

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

<< ∧ >>

❖ Iterators

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 = openRelation(db,"Employee",READ);
Scan s = start_scan(r);
Tuple t;  // current tuple
while ((t = next_tuple(s)) != NULL) {
   char *name = getStrField(t,2);
   printf("%s\n", name);
}
```

<< ∧ >>

❖ `next_tuple()` Function

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

```
typedef ScanData *Scan;

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

Assume tuples are indexed 0..**`nTuples(p)-1`**

Assume pages are indexed 0..**`nPages(rel)-1`**

<< ∧ >>

❖ next_tuple() Function (cont)

Implementation of **Tuple next_tuple(Scan)** function

```
Tuple next_tuple(Scan s)
{
   if (s->curTID >= nTuples(s->page)-1) {
      // get a new page; exhausted current page
      s->curPID++;
      if (s->curPID >= nPages(s->rel))
         return NULL;
      else {
         s->page = get_page(s->rel, s->curPID);
         s->curTID = -1;
      }
   }
   s->curTID++;
   return get_tuple(s->rel, s->page, s->curTID);
}
```

<< ∧ >>

❖ Relation Copying

Consider an SQL statement like:

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

Effectively, copies data from one table to a new table.

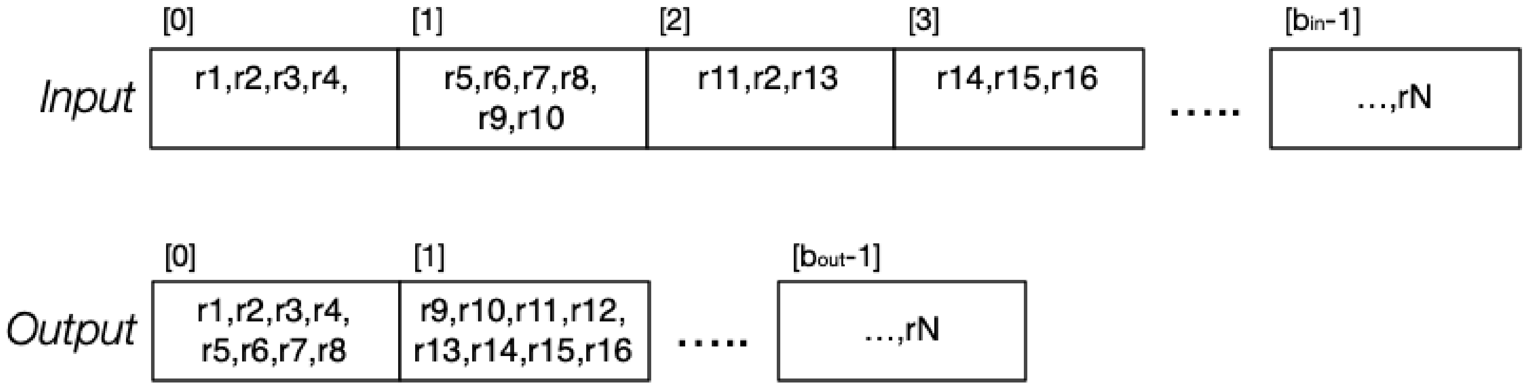
Process:

```
make empty relation T
s = start scan of S
while (t = next_tuple(s)) {
    insert tuple t into relation T
}
```

❖ Relation Copying (cont)

It is possible that **`T`** is smaller than **`S`**

- may be unused free space in **`S`** where tuples were removed
- if **`T`** is built by simple append, will be compact



Input

[0]	[1]	[2]	[3]		[b_{in}-1]
r1,r2,r3,r4,	r5,r6,r7,r8, r9,r10	r11,r2,r13	r14,r15,r16	.....	...,rN

Output

[0]	[1]		[b_{out}-1]
r1,r2,r3,r4, r5,r6,r7,r8	r9,r10,r11,r12, r13,r14,r15,r16	.....	...,rN

<< ∧ >>

❖ Relation Copying (cont)

In terms of existing relation/page/tuple operations:

```
Relation in;       // relation handle (incl. files)
Relation out;      // relation handle (incl. files)
int ipid,opid,tid; // page and record indexes
Record rec;        // current record (tuple)
Page ibuf,obuf;    // input/output file buffers

in = openRelation("S", READ);
out = openRelation("T", NEW|WRITE);
clear(obuf);  opid = 0;
for (ipid = 0; ipid < nPages(in); ipid++) {
    ibuf = get_page(in, ipid);
    for (tid = 0; tid < nTuples(ibuf); tid++) {
        rec = get_record(ibuf, tid);
        if (!hasSpace(obuf,rec)) {
            put_page(out, opid++, obuf);
            clear(obuf);
        }
        insert_record(obuf,rec);
}   }
if (nTuples(obuf) > 0) put_page(out, opid, obuf);
```

<< ∧ >>

❖ Scanning in PostgreSQL

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

Implements iterator data/operations:

- `HeapScanDesc` ... struct containing iteration state
- `scan = heap_beginscan(rel,...,nkeys,keys)`
- `tup = heap_getnext(scan, direction)`
- `heap_endscan(scan)` ... frees up **`scan`** struct
- `res = HeapKeyTest(tuple,...,nkeys,keys)`
  ... performs **`ScanKeys`** tests on tuple ... is it a result tuple?

<< ∧ >>

❖ Scanning in PostgreSQL (cont)

```
typedef HeapScanDescData *HeapScanDesc;

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

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 (update=delete+insert)
  - other file-specific operators

Produced: 28 Feb 2021