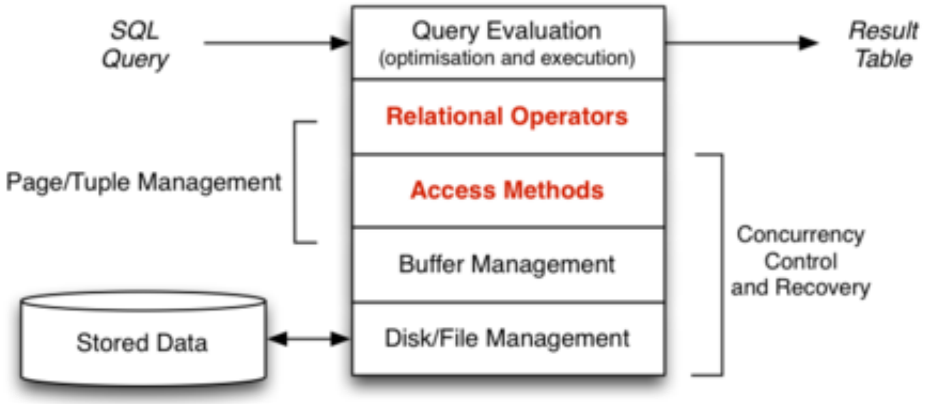
Week 05 Lecture

Implementing Relational Operations

Implementing Relational Operators

2/53

Implementation of relational operations in DBMS:



... Implementing Relational Operators

3/53

So far, have considered ...

- scanning (e.g. `select * from R`)

With 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

Now ...

- sorting (e.g. `select * from R order by x`)
- projection (e.g. `select x,y from R`)
- selection (e.g. `select * from R where` *`Cond`*)

and

- *indexes* ... search trees based on pages/keys
- *signatures* ... bit-strings which "summarize" tuples

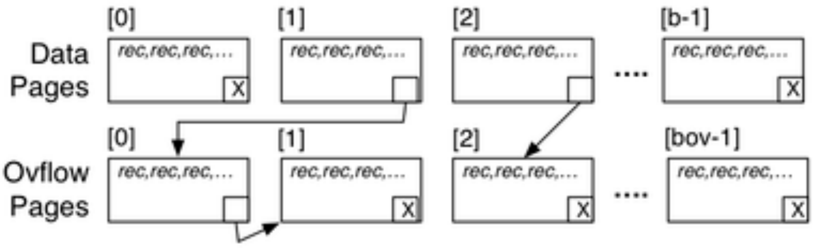
... Implementing Relational Operators

4/53

File/query Parameters ...

- *r* tuples of size *R*, *b* pages of size *B*, *c* tuples per page
- *Rel.k* attribute in `where` clause, b_q answer pages for query *q*
- b_{Ov} overflow pages, average overflow chain length *Ov*

File structures ...



Reminder on Cost Analyses

5/53

When showing the cost of operations, don't include T_r and T_w:

- for queries, simply count number of pages read
- for updates, use n_r and n_w to distinguish reads/writes

When comparing two methods for same query

- ignore the cost of writing the result (same for both)

In counting reads and writes, assume minimal buffering

- each `request_page()` causes a read
- each `release_page()` causes a write (if page is dirty)

Sorting

The Sort Operation

7/53

Sorting is explicit in queries only in the `order by` clause

```
select * from Students order by name;
```

Sorting is used internally in other operations:

- eliminating duplicate tuples for projection
- ordering files to enhance select efficiency
- implementing various styles of join
- forming tuple groups in `group by`

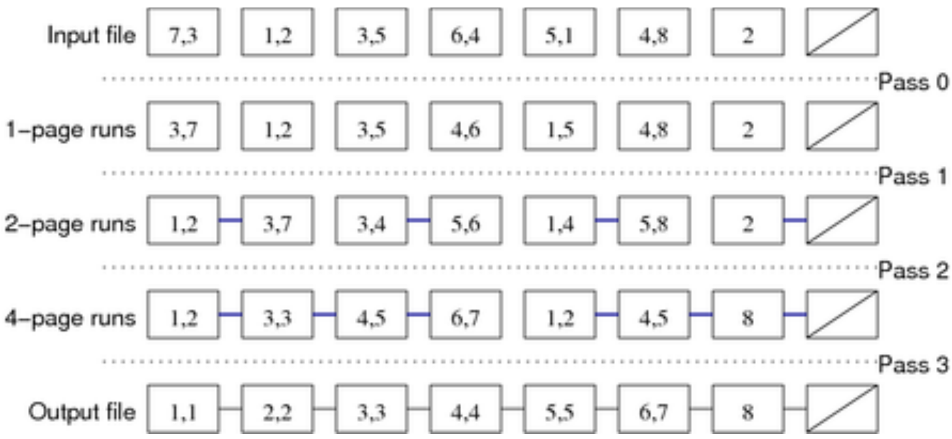
Sort methods such as quicksort are designed for in-memory data.

For large data on disks, use external sorts such as *merge sort*.

Two-way Merge Sort

8/53

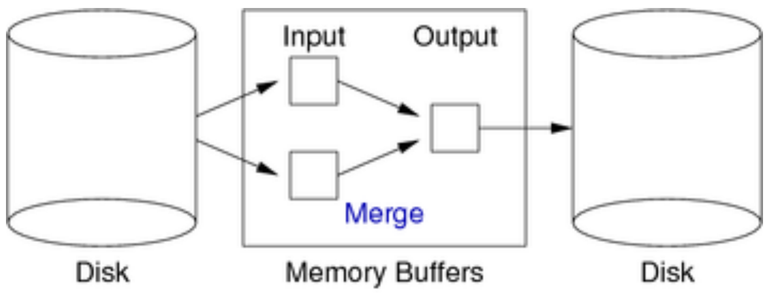
Example:



... Two-way Merge Sort

9/53

Requires three in-memory buffers:



Assumption: cost of merge on two buffers ≅ *0*.

Comparison for Sorting

10/53

Above assumes that we have a function to compare tuples.

Needs to understand ordering on different data types.

E.g. a function `tupCompare(r1,r2,f)` (cf. C's `strcmp`)

- takes two tuples `r1`, `r2` and a field name `f`
- returns negative value if `r1.f < r2.f`
- returns positive value if `r1.f > r2.f`
- returns zero value if `r1.f == r2.f`

Can work on multiple attributes (sort on first, then second if equal, ...)

```
-- example multi-attribute sort
select * from Students
order by age desc, year_enrolled
```

Cost of Two-way Merge Sort

11/53

For a file containing *b* data pages:

- require *ceil*($log_2 b$) passes to sort,
- each pass requires *b* page reads, *b* page writes

Gives total cost: $2.b.ceil(log_2 b)$

Example: Relation with $r=10^5$ and $c=50$ ⇒ $b=2000$ pages.

Number of passes for sort: $ceil(log_2 2000) = 11$

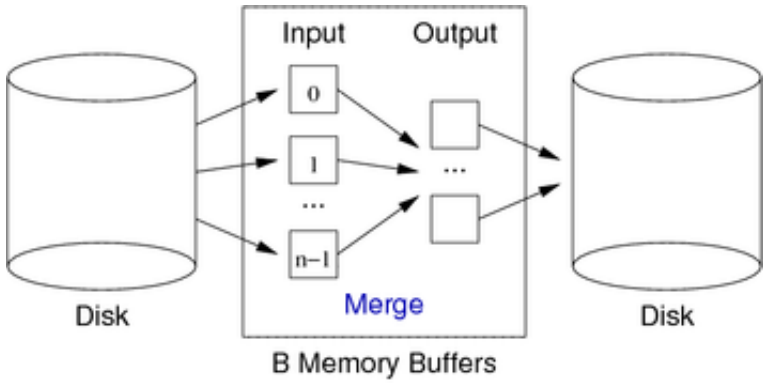
Reads/writes entire file 11 times! Can we do better?

---

n-Way Merge Sort

12/53

Use *N* memory buffers: *n* input buffers, *N-n* output buffers



Typically, use: *N-1* input buffers, 1 output buffer

... n-Way Merge Sort

13/53

Method:

```
// Produce n-1-page-long runs
for each group of n-1 pages in Rel {
    read pages into memory buffers
    sort group in memory
    write pages out to Temp via output buffer
}
// Merge runs until everything sorted
numberOfRuns = ceil(b/n)
while (numberOfRuns > 1) {
    for each group of n runs in Temp {
        merge into a single run via input buffers
        write run to newTemp via output buffer
    }
    numberOfRuns = ceil(numberOfRuns/n)
    Temp = newTemp // swap input/output files
}
```

... n-Way Merge Sort

14/53

Method for merging *n* runs:

```
for i = 1..n {
   read first page of run[i] into a buffer[i]
```

```
   set current tuple cur[i] to first tuple in buffer[i]
}
while (more than 1 run still has tuples) {
   s = find buffer with smallest tuple as cur[i]
   copy tuple cur[i] to output buffer
   if (output buffer full) { write it and clear it}
   advance cur[i] to next tuple
   if (no more tuples in buffer[i]) {
      if (no more pages in run[i])
         mark run[i] as complete
      else {
         read next page of run[i] into buffer[i]
         set cur[i] to first tuple in buffer[i]
}  }  }
copy tuples in non-empty buffer to output
```

Exercise 1: Cost of n-Way Merge Sort

15/53

How many reads+writes to sort the following:

- r = 1048576 tuples (2^{20})
- R = 62 bytes per tuple (fixed-size)
- B = 4096 bytes per page
- H = 96 bytes of header data per page
- D = 1 presence bit per tuple in page directory
- all pages are full

Consider for the cases:

- 8 input buffers, 1 output buffer
- 32 input buffers, 1 output buffer
- 256 input buffers, 1 output buffer

Sorting in PostgreSQL

16/53

Sort uses a polyphase merge-sort (from Knuth):

- backend/utils/sort/tuplesort.c

Tuples are mapped to **`SortTuple`** structs for sorting:

- containing pointer to tuple and sort key
- no need to reference actual Tuples during sort
- unless multiple attributes used in sort

If all data fits into memory, sort using **`qsort()`**.

If memory fills while reading, form "runs" and do disk-based sort.

... Sorting in PostgreSQL

17/53

Disk-based sort has phases:

- divide input into sorted runs using HeapSort
- merge using N buffers, one output buffer
- N = as many buffers as `workMem` allows

Described in terms of "tapes" ("tape" ≅ sorted run)

Implementation of "tapes": backend/utils/sort/logtape.c

... Sorting in PostgreSQL

18/53

Sorting is generic and comparison operators are defined in catalog:

```
// gets pointer to function via pg_operator
SelectSortFunction(Oid sortOperator,
                   bool nulls_first,
                   Oid *sortFunction,
                   int *sortFlags);

// returns negative, zero, positive
ApplySortFunction(FmgrInfo *sortFunction,
                  int sortFlags,
                  Datum datum1, bool isNull1,
                  Datum datum2, bool isNull2);
```

Flags indicate: ascending/descending, nulls-first/last.

Implementing Projection

The Projection Operation

20/53

Consider the query:

```
select distinct name,age from Employee;
```

If the `Employee` relation has four tuples such as:

```
(94002, John, Sales, Manager,   32)
(95212, Jane, Admin, Manager,   39)
(96341, John, Admin, Secretary, 32)
(91234, Jane, Admin, Secretary, 21)
```

then the result of the projection is:

```
(Jane, 21)   (Jane, 39)   (John, 32)
```

Note that duplicate tuples (e.g. `(John,32)`) are eliminated.

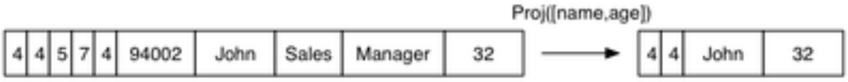
... The Projection Operation

21/53

The projection operation needs to:

1. scan the entire relation as input
   - already seen how to do scanning
2. remove unwanted attributes in output tuples
   - implementation depends on tuple internal structure
3. eliminate any duplicates produced
   - two approaches: sorting or hashing

Example of task 2:



Sort-based Projection

22/53

Requires a temporary file/relation (`Temp`)

```
for each tuple T in Rel {
    T' = mkTuple([attrs],T)
    write T' to Temp
}

sort Temp on [attrs]

for each tuple T in Temp {
    if (T == Prev) continue
    write T to Result
    Prev = T
}
```

Exercise 2: Cost of Sort-based Projection

23/53

Consider a table *R(x,y,z)* with tuples:

```
Page 0:  (1,1,'a')   (11,2,'a')  (3,3,'c')
Page 1:  (13,5,'c')  (2,6,'b')   (9,4,'a')
Page 2:  (6,2,'a')   (17,7,'a')  (7,3,'b')
Page 3:  (14,6,'a')  (8,4,'c')   (5,2,'b')
Page 4:  (10,1,'b')  (15,5,'b')  (12,6,'b')
Page 5:  (4,2,'a')   (16,9,'c')  (18,8,'c')
```

SQL: `create T as (select distinct y from R)`

Assuming:

- 3 memory buffers, 2 for input, one for output
- pages/buffers hold 3 `R` tuples (i.e. $c_R=3$), 6 `T` tuples (i.e. $c_T=6$)

Show how sort-based projection would execute this statement.

Cost of Sort-based Projection

24/53

The costs involved are (assuming *n+1* buffers for sort):

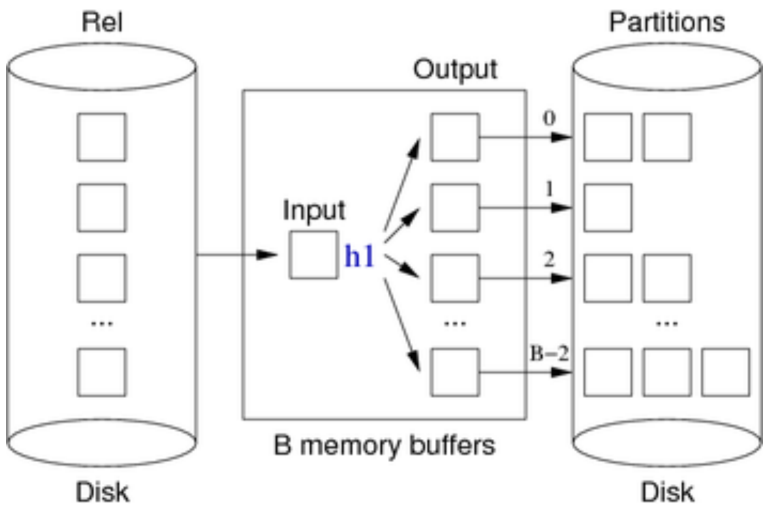
- scanning original relation `Rel`: b_R (with c_R)
- writing `Temp` relation: b_T (smaller tuples, $c_T > c_R$, *sorted*)
- sorting `Temp` relation: $2.b_T.ceil(log_n b_0)$ where $b_0 = ceil(b_T/n)$
- removing duplicates from `Temp`: b_T
- writing the result relation: b_{Out} (maybe less tuples)

Cost = sum of above = $b_R + b_T + 2.b_T.ceil(log_n b_0) + b_T + b_{Out}$

Hash-based Projection

25/53

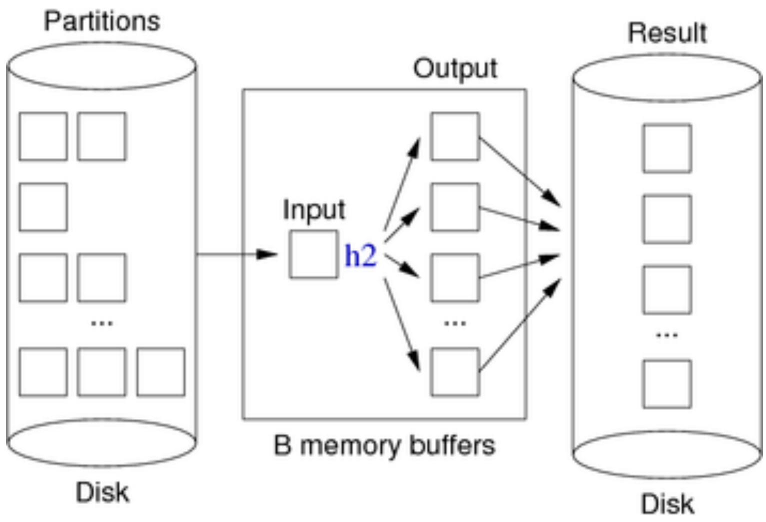
Partitioning phase:



... Hash-based Projection

26/53

Duplicate elimination phase:



... Hash-based Projection

27/53

Algorithm for both phases:

```
for each tuple T in relation Rel {
    T' = mkTuple([attrs],T)
    H = h1(T', n)
    B = buffer for partition[H]
    insert T' into B
    if (B full) write and clear B
}
for each partition P in 0..n-1 {
    for each tuple T in partition P {
        H = h2(T, n)
```

```
        B = buffer for hash value H
        if (T not in B) insert T into B
        // assumes B never gets full
    }
    write and clear all buffers
}
```

Exercise 3: Cost of Hash-based Projection

Consider a table *R(x,y,z)* with tuples:

```
Page 0:  (1,1,'a')   (11,2,'a')  (3,3,'c')
Page 1:  (13,5,'c')  (2,6,'b')   (9,4,'a')
Page 2:  (6,2,'a')   (17,7,'a')  (7,3,'b')
Page 3:  (14,6,'a')  (8,4,'c')   (5,2,'b')
Page 4:  (10,1,'b')  (15,5,'b')  (12,6,'b')
Page 5:  (4,2,'a')   (16,9,'c')  (18,8,'c')
-- and then the same tuples repeated for pages 6-11
```

SQL: `create T as (select distinct y from R)`

Assuming:

- 4 memory buffers, one for input, 3 for partitioning
- pages/buffers hold 3 `R` tuples (i.e. $c_R=3$), 4 `T` tuples (i.e. $c_T=4$)
- hash functions: h1(x) = x%3, h2(x) = (x%4)%3

Show how hash-based projection would execute this statement.

Cost of Hash-based Projection

The total cost is the sum of the following:

- scanning original relation `Rel`: b_R
- writing partitions: b_P (b_R vs b_P ?)
- re-reading partitions: b_P
- writing the result relation: b_{Out}

Cost = $b_R + 2b_P + b_{Out}$

To ensure that *n* is larger than the largest partition ...

- use hash functions (h1,h2) with uniform spread
- allocate at least $sqrt(b_R)$ buffers
- if insufficient buffers, maybe significant re-reading overhead

---

Index-only Projection

Can do projection without accessing data file iff ...

- relation is indexed on $(A_1,A_2,...A_n)$ (indexes described later)
- projected attributes are a prefix of $(A_1,A_2,...A_n)$

Basic idea:

- scan through index file (which is already sorted on attributes)
- duplicates are already adjacent in index, so easy to skip

Cost analysis ...

- index has b_i pages (where $b_i \ll b_R$)
- Cost = b_i reads + b_{Out} writes

Comparison of Projection Methods

31/53

Difficult to compare, since they make different assumptions:

- index-only: needs an appropriate index
- hash-based: needs buffers and good hash functions
- sort-based: needs only buffers ⇒ use as default

Best case scenario for each (assuming $n+1$ in-memory buffers):

- index-only: $b_i + b_{Out} \ll b_R + b_{Out}$
- hash-based: $b_R + 2.b_P + b_{Out}$
- sort-based: $b_R + b_T + 2.b_T.ceil(log_n b_0) + b_T + b_{Out}$

We normally omit b_{Out}, since each method produces the same result

---

Projection in PostgreSQL

32/53

Code for projection forms part of execution iterators:

- backend/executor/execQual.c

Functions involved with projection:

- `ExecProject(projInfo,...)` ... extracts/stores projected data
- `ExecTargetList(...)` ... makes new tuple from old tuple + projection info
- `ExecStoreTuple(newTuple,...)` ... save tuple in output slot

Implementing Selection

Varieties of Selection

34/53

Selection: `select * from R where C`

- filters a subset of tuples from one relation `R`
- based on a condition `C` on the attribute values

We consider three distinct styles of selection:

- 1-d (one dimensional) (condition uses only *1* attribute)
- *n*-d (multi-dimensional) (condition uses >*1* attribute)
- similarity (approximate matching, with ranking)

Each style has several possible file-structures/techniques.

... Varieties of Selection

35/53

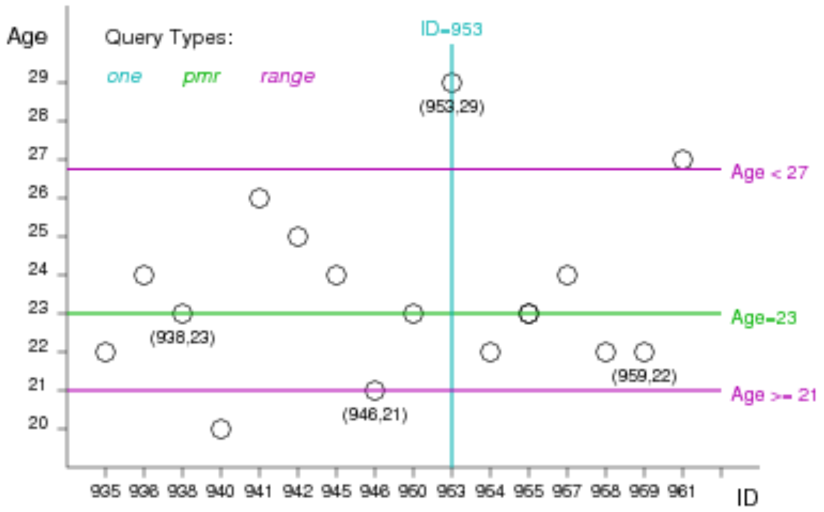
We can view a relation as defining a *tuple space*

- assume relation *R* with attributes $a_1,...,a_n$
- attribute domains of *R* specify a *n*-dimensional space
- each tuple $(v_1,v_2,...,v_n) \in R$ is a point in that space
- queries specify values/ranges on $N \geq 1$ dimensions
- a query defines a point/line/plane/region of the *n*-d space
- results are tuples lying at/on/in that point/line/plane/region

E.g. if *N=n*, we are checking existence of a tuple (at a point)

... Varieties of Selection

36/53



... Varieties of Selection

37/53

One-dimensional selection queries = condition on single attribute.

- *one*: `select * from R where k = val`

  where `k` is a `unique` attribute and `val` is a constant

- *pmr*: `select * from R where k = val`

  where `k` is non-`unique` and `val` is a constant

- *range*: `select * from R where k ≥ lo and k ≤ hi`

  where `k` is any attribute and `lo` and `hi` are constants

  either `lo` or `hi` may be omitted for open-ended range

Exercise 4: Query Types

38/53

Using the relation:

```
create table Courses (
   id        integer primary key,
   code      char(8),   -- e.g. 'COMP9315'
```

```
    title    text,     -- e.g. 'Computing 1'
    year     integer,  -- e.g. 2000..2016
    convenor integer references Staff(id),
    constraint once_per_year unique (code,year)
);
```

give examples of each of the following query types:

1. a 1-d *one* query, an n-d *one* query
2. a 1-d *pmr* query, an n-d *pmr* query
3. a 1-d *range* query, an n-d *range* query

Suggest how many solutions each might produce ...

Implementing Select Efficiently

39/53

Two basic approaches:

- physical arrangement of tuples
  - sorting (search strategy)
  - hashing (static, dynamic, *n*-dimensional)
- additional indexing information
  - index files (primary, secondary, trees)
  - signatures (superimposed, disjoint)

Our analyses assume: 1 input buffer available for each relation.

If more buffers are available, most methods benefit.

Heap Files

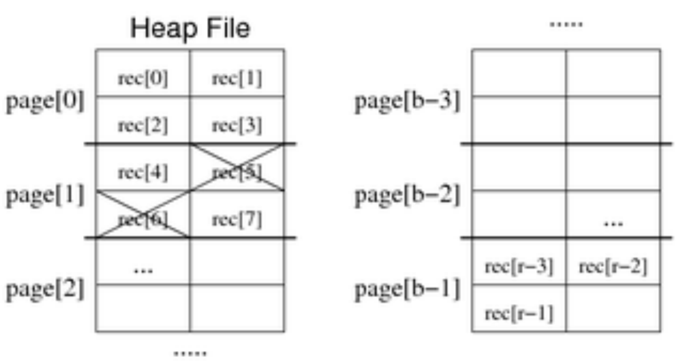
> Note: this is **not** "heap" as in the top-to-bottom ordered tree.
> It means simply an unordered collection of tuples in a file.

Heap File Structure

41/53

The simplest possible file organisation.

New tuples inserted at end of file; tuples deleted by marking.



42/53

Selection in Heaps

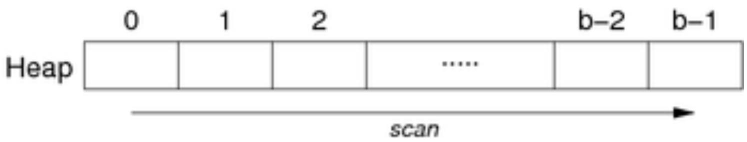
For all selection queries, the only possible strategy is:

```
// select * from R where C
f = openFile(fileName("R"),READ);
for (p = 0; p < nPages(f); p++) {
    buf = readPage(f, p);
    for (i = 0; i < nTuples(buf); i++) {
        tup = getTuple(buf,i);
        if (tup satisfies C)
            add tup to result set
    }
}
```

i.e. linear scan through file searching for matching tuples

... Selection in Heaps

The heap is scanned from the first to the last page:



$Cost_{range} = Cost_{pmr} = b$

If we know that only one tuple matches the query (*one* query),
a simple optimisation is to stop the scan once that tuple is found.

$Cost_{one}$: Best = *1* Average = *b/2* Worst = *b*

Insertion in Heaps

Insertion: new tuple is appended to file (in last page).

```
f = openFile(fileName("R"),READ|WRITE);
b = nPages(f)-1;
buf = readPage(f, b);  // request page
if (isFull(buf)) // all slots used
    { b++; clear(buf); }
if (tooLarge(newTup,buf)) // not enough space for tuple
    { deal with oversize tuple }
insertTuple(newTup, buf);
writePage(f, b, buf);  // mark page as dirty & release
```

$Cost_{insert} = 1_r + 1_w$

Plus possible extra writes for oversize tuples, e.g. PostgreSQL's TOAST files

---

... Insertion in Heaps

Alternative strategy:

- find any page from `R` with enough space
- preferably a page already loaded into memory buffer

PostgreSQL's strategy:

- use last updated page of `R` in buffer pool
- otherwise, search buffer pool for page with enough space
- assisted by free space map (FSM) associated with each table
- for details: **`backend/access/heap/{heapam.c,hio.c}`**

... Insertion in Heaps

46/53

PostgreSQL's tuple insertion:

```
heap_insert(Relation relation,    // relation desc
            HeapTuple newtup,     // new tuple data
            CommandId cid, ...)   // SQL statement
```

- finds page which has enough free space for `newtup`
- ensures page loaded into buffer pool and locked
- copies tuple data into page buffer, sets `xmin`, etc.
- marks buffer as dirty
- writes details of insertion into transaction log
- returns OID of new tuple if relation has OIDs

Deletion in Heaps

47/53

SQL: `delete from` *R* `where` *Condition*

Implementation of deletion:

```
f = openFile(fileName("R"),READ|WRITE);
for (p = 0; p < nPages(f); p++) {
    buf = readPage(f, p);
    ndels = 0;
    for (i = 0; i < nTuples(buf); i++) {
        tup = getTuple(buf,i);
        if (tup satisfies Condition)
            { ndels++; deleteTuple(buf,i); }
    }
    if (ndels > 0) writePage(f, p, buf);
    if (ndels > 0 && unique) break;
}
```

If buffers, `read` = `request`, `write` = `mark-as-dirty`

Exercise 5: Cost of Deletion in Heaps

48/53

Consider the following queries ...

```
delete from Employees where id = 12345   -- one
delete from Employees where dept = 'Marketing'  -- pmr
delete from Employees where 40 <= age and age < 50  -- range
```

Show how each will be executed and estimate the cost, assuming:

- $b = 100$, $b_{q2} = 3$, $b_{q3} = 20$

State any other assumptions.

Generalise the cost models for each query type.

... Deletion in Heaps

49/53

PostgreSQL tuple deletion:

```
heap_delete(Relation relation,     // relation desc
            ItemPointer tid, ..., // tupleID
            CommandId cid, ...)    // SQL statement
```

- gets page containing tuple into buffer pool and locks it
- sets flags, commandID and `xmax` in tuple; dirties buffer
- writes indication of deletion to transaction log (at commit time)

Vacuuming eventually compacts space in each page.

Updates in Heaps

50/53

SQL: `update` *R* `set` *F* = *val* `where` *Condition*

Analysis for updates is similar to that for deletion

- scan all pages
- replace any updated tuples (within each page)
- write affected pages to disk

$Cost_{update} = b_r + b_{qw}$

Complication: new version of tuple larger than old version (too big for page)

Solution: delete, re-organise free space, then insert

... Updates in Heaps

51/53

PostgreSQL tuple update:

```
heap_update(Relation relation,      // relation desc
            ItemPointer otid,       // old tupleID
            HeapTuple newtup, ..., // new tuple data
            CommandId cid, ...)     // SQL statement
```

- essentially does `delete(otid)`, then `insert(newtup)`
- also, sets old tuple's `ctid` field to reference new tuple
- can also update-in-place if no referencing transactions

Heaps in PostgreSQL

52/53

PostgreSQL stores all table data in heap files (by default).

Typically there are also associated index files.

If a file is more useful in some other form:

- PostgreSQL may make a transformed copy during query execution
- programmer can set it via `create index...using hash`

Heap file implementation: src/backend/access/heap

... Heaps in PostgreSQL

53/53

PostgreSQL "heap file" may use multiple physical files

- files are named after the OID of the corresponding table
- first data file is called simply `OID`
- if size exceeds 1GB, create a *fork* called `OID.1`
- add more forks as data size grows (one fork for each 1GB)
- other files:
    - free space map (`OID_fsm`), visibility map (`OID_vm`)
    - optionally, TOAST file (if table has varlen attributes)
- for details: Chapter 55 in PostgreSQL documentation

Produced: 4 Apr 2016