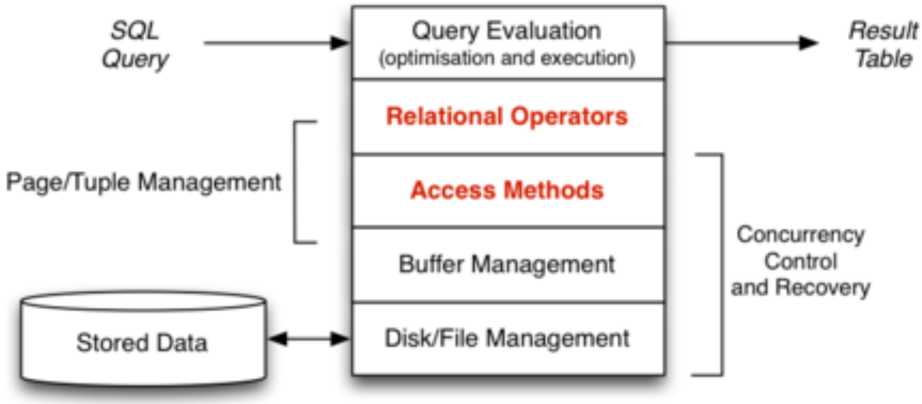
Week 04 Lecture

Implementing Relational Operations

DBMS Architecture (revisited)

2/31

Implementation of relational operations in DBMS:



Relational Operations

3/31

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

4/31

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

5/31

SQL vs DBMS engine

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

Cost Models

Cost Models

7/31

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

Assumptions in our cost models:

- memory (RAM) is "small", fast, byte-at-a-time
- disk storage is very large, slow, page-at-a-time
- every request to read/write a page results in a read/write

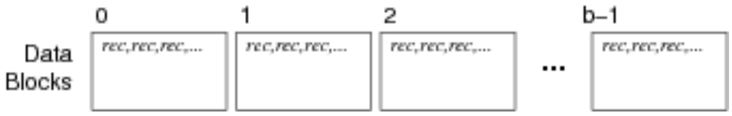
Trying to estimate costs with multiple concurrent ops *and* buffering is difficult!

... Cost Models

8/31

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

9/31

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

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

Back-of-the-envelope calculation: how many piano tuners in Sydney?

- Sydney has ≅ 4 000 000 people
- Average household size ≅ 3 ∴ 1 300 000 households
- Lets 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

10/31

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.

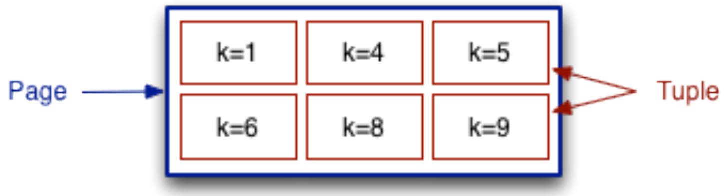
Example File Structures

11/31

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



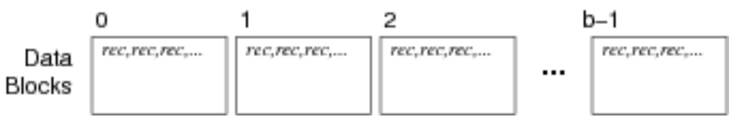
... Example File Structures

12/31

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 1: Operation Costs

13/31

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)

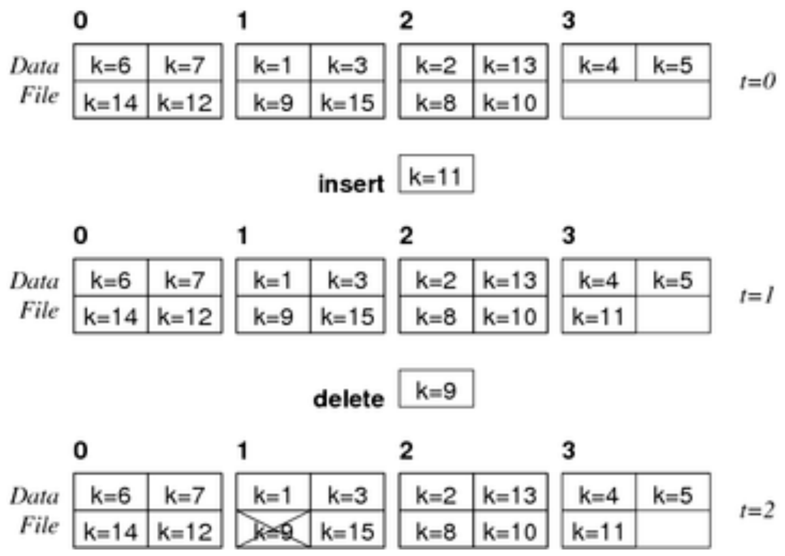
Assume also

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

Operation Costs Example

14/31

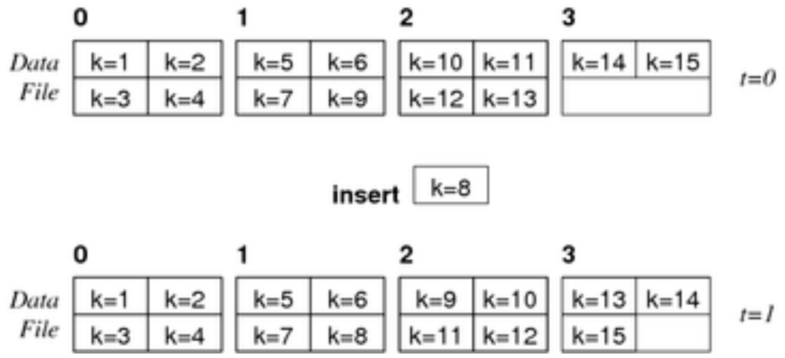
Heap file with *b = 4, c = 4*:



... Operation Costs Example

15/31

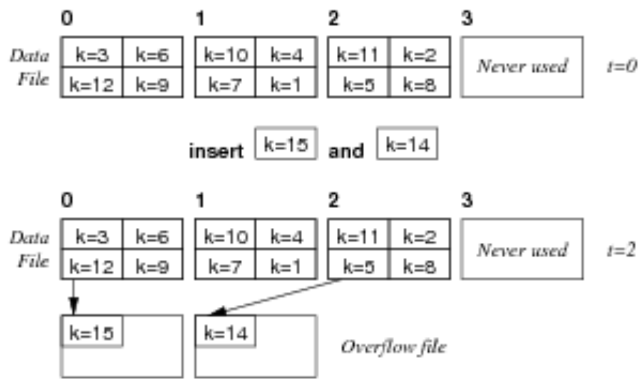
Sorted file with *b = 4, c = 4*:



... Operation Costs Example

16/31

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



Scanning

18/31

Scanning

Consider the query:

```
select * from Rel;
```

Operational view:

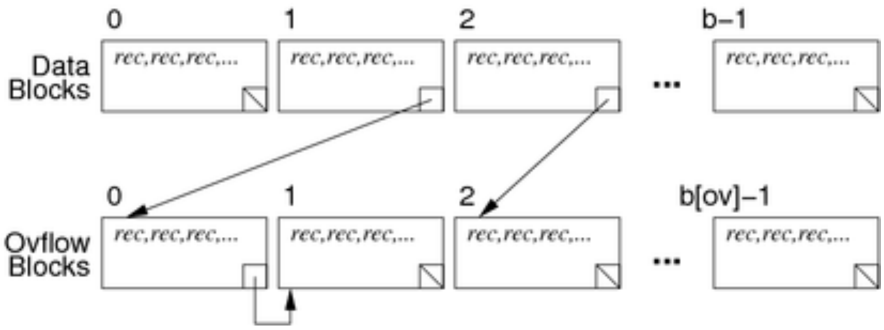
```
for each page P in file of relation Rel {
   for each tuple t in page P {
      add tuple t to result set
   }
}
```

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

Time Cost = $(b + b_{Ov})T_r$, *Page Cost* = $b + b_{Ov}$

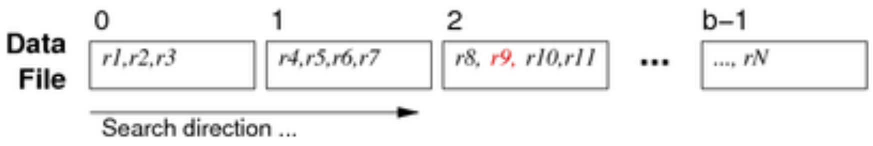
where b_{Ov} = total number of overflow pages

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; could be in any page.

... Selection via Scanning

22/31

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

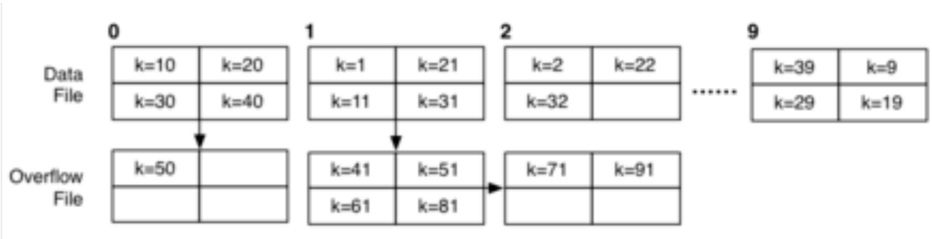
- best case: read one page, find tuple
- worst case: read all *b* pages, find in last (or don't find)
- average case: read half of the pages (*b/2*)

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

Exercise 2: Cost of Search in Hashed File

23/31

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)

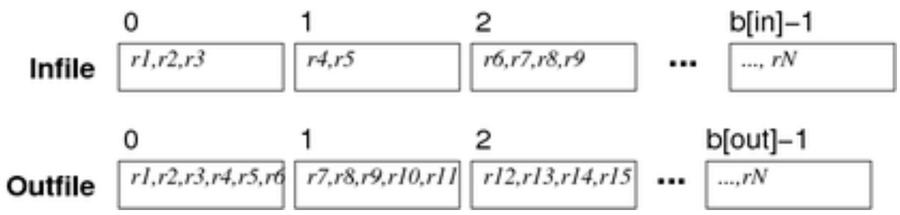
Relation Copying

24/31

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

25/31

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 buf,obuf; // input/output file buffers

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

Exercise 3: Cost of Relation Copy

26/31

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

Cost Calculations (revisited)

27/31

Assumptions:

- disk read time $T_r \cong T_w$ disk write time
- average disk read/write time is a *large* constant value

- the *real* measure of cost is number of page↔disk transfers

So, in all future analyses, we ignore T_r and T_w

- measure *Cost* as *number of pages* read and written

Also, when comparing two algorithms for same task

- ignore cost of writing result; same in both cases

Exercise 4: PostgreSQL Tuple Visibility

28/31

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

29/31

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

30/31

```
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

31/31

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: 21 Mar 2016