>>

Sorted Files

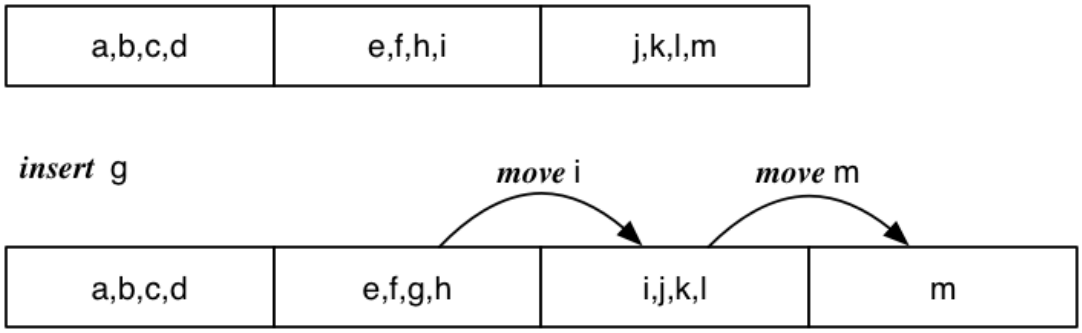
^ >>

❖ Sorted Files

Records stored in file in order of some field k (the sort key).

Makes searching more efficient; makes insertion less efficient

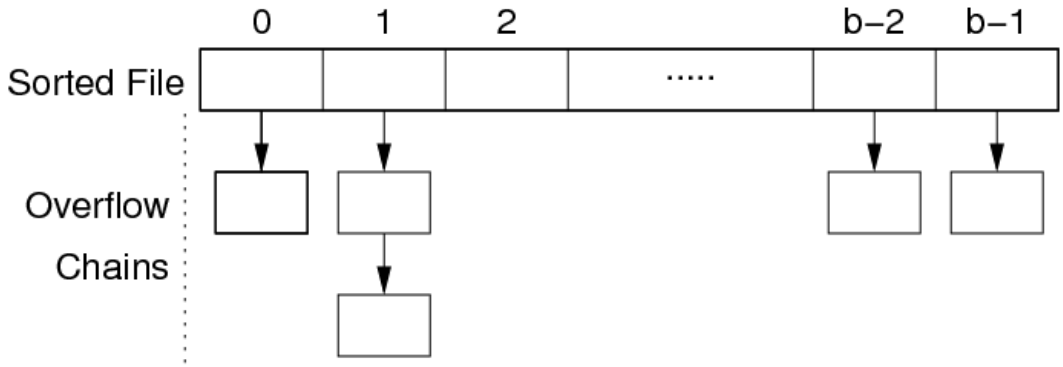
E.g. assume *c = 4*



<< ^ >>

❖ Sorted Files (cont)

In order to mitigate insertion costs, use overflow pages.



Total number of overflow pages = b_{ov}.

Average overflow chain length = $Ov = b_{ov} / b$.

Bucket = data page + its overflow page(s)

<< ^ >>

❖ Selection in Sorted Files

For *one* queries on sort key, use binary search.

```
// select * from R where k = val  (sorted on R.k)
lo = 0; hi = nPages(rel)-1
while (lo <= hi) {
    mid = (lo+hi) / 2;  // int division with truncation
    (tup,loVal,hiVal) = searchBucket(rel,mid,x,val);
    if (tup != NULL) return tup;
    else if (val < loVal) hi = mid - 1;
    else if (val > hiVal) lo = mid + 1;
    else return NOT_FOUND;
}
return NOT_FOUND;
```

where **`rel`** is relation handle, **`mid,lo,hi`** are page indexes,
`k` is a field/attr, **`val,loVal,hiVal`** are values for **`k`**

<< ^ >>

❖ Selection in Sorted Files (cont)

Search a page and its overflow chain for a key value

```
searchBucket(rel,p,k,val)
{
    get_page(rel,p,buf);
    (tup,min,max) = searchPage(buf,k,val,+INF,-INF)
    if (tup != NULL) return(tup,min,max);
    ovf = openOvFile(f);
    ovp = ovflow(buf);
    while (tup == NULL && ovp != NO_PAGE) {
        get_page(ovf,ovp,buf);
        (tup,min,max) = searchPage(buf,k,val,min,max)
        ovp = ovflow(buf);
    }
    return (tup,min,max);
}
```

Assumes each page contains index of next page in Ov chain

<< ^ >>

❖ Selection in Sorted Files (cont)

Search within a page for key; also find min/max key values

```
searchPage(buf,k,val,min,max)
{
    res = NULL;
    for (i = 0; i < nTuples(buf); i++) {
        tup = get_tuple(buf, i);
        if (tup.k == val) res = tup;
        if (tup.k < min) min = tup.k;
        if (tup.k > max) max = tup.k;
    }
    return (res,min,max);
}
```

<< ^ >>

❖ Selection in Sorted Files (cont)

The above method treats each bucket like a single large page.

Cases:

- best: find tuple in first data page we read
- worst: full binary search, and not found
  - examine $log_2 b$ data pages
  - plus examine all of their overflow pages
- average: examine some data pages + their overflow pages

$Cost_{one}$: Best = 1 Worst = $log_2 b + b_{ov}$

Average case cost analysis needs assumptions (e.g. data distribution)

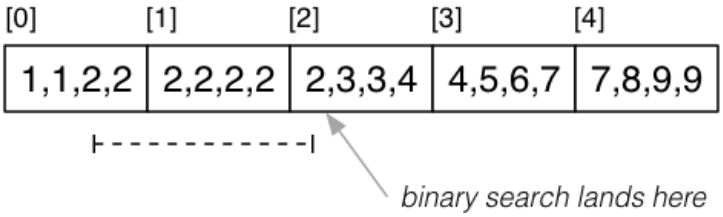
<< ^ >>

❖ Selection in Sorted Files (cont)

For *pmr* query, on non-unique attribute *k*, where file is sorted on *k*

- tuples containing *k* may span several pages

E.g. **`select * from R where k = 2`**



Begin by locating a page *p* containing **`k=`***val* (as for *one* query).

Scan backwards and forwards from *p* to find matches.

Thus, $Cost_{pmr} = Cost_{one} + (b_q\text{-}1).(1+Ov)$

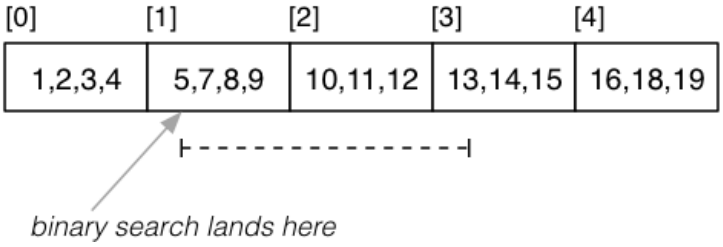
<< ^ >>

❖ Selection in Sorted Files (cont)

For *range* queries on unique sort key (e.g. primary key):

- use binary search to find lower bound
- read sequentially until reach upper bound

E.g. `select * from R where k >= 5 and k <= 13`



$Cost_{range} = Cost_{one} + (b_q - 1).(1+Ov)$

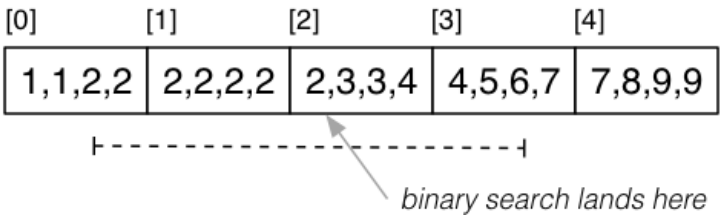
<< ^ >>

❖ Selection in Sorted Files (cont)

For *range* queries on non-unique sort key, similar method to *pmr*.

- binary search to find lower bound
- then go backwards to start of run
- then go forwards to last occurence of upper-bound

E.g. `select * from R where k >= 2 and k <= 6`



$Cost_{range} = Cost_{one} + (b_q - 1).(1+Ov)$

<< ^ >>

❖ Selection in Sorted Files (cont)

So far, have assumed query condition involves sort key *k*.

But what about **`select * from R where j = 100.0`**?

If condition contains attribute *j*, not the sort key

- file is unlikely to be sorted by *j* as well
- sortedness gives no searching benefits

$Cost_{one}$, $Cost_{range}$, $Cost_{pmr}$ as for heap files

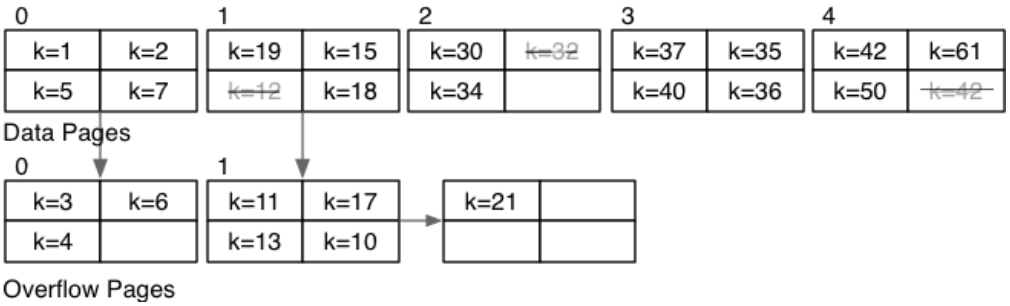
❖ Insertion into Sorted Files

Insertion approach:

- find appropriate page for tuple (via binary search)
- if page not full, insert into page
- otherwise, insert into next overflow page with space

Thus, $Cost_{insert} = Cost_{one} + \delta_w$ (where δ_w = 1 or 2)

Consider insertions of k=33, k=25, k=99 into:



<< ^

❖ Deletion from Sorted Files

E.g. **`delete from R where k = 2`**

Deletion strategy:

- find matching tuple(s)
- mark them as deleted

Cost depends on selectivity of selection condition

Recall: selectivity determines b_q (# pages with matches)

Thus, $Cost_{delete} = Cost_{select} + b_{qw}$

Produced: 7 Mar 2021