Week 04 Lectures

Reminder on Cost Analyses

1/106

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)

3

Relation Copying

2/106

Consider an SQL statement like:

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

Effectively, copies data from one table to another.

Process:

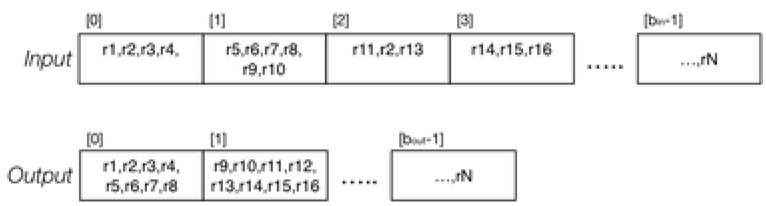
```
s = start scan of S
make empty relation T
while (t = next_tuple(s)) {
    insert tuple t into relation T
}
```

... Relation Copying

3/106

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



... Relation Copying

4/106

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++) {
    get_page(in, ipid, ibuf);
```

```
    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);
```

Exercise 1: Cost of Relation Copy

5/106

三种copy 情况看
exercise

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 ...

- *r* records in input file, *c* records/page
- b_{in} = number of pages in input file
- some pages in input file are *not* full
- all pages in output file are full (except the last)

Give cost in terms of #pages read + #pages written

Scanning in PostgreSQL

6/106

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

7/106

```
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

8/106

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

Sorting

The Sort Operation

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

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

Sorting is used internally in other operations:

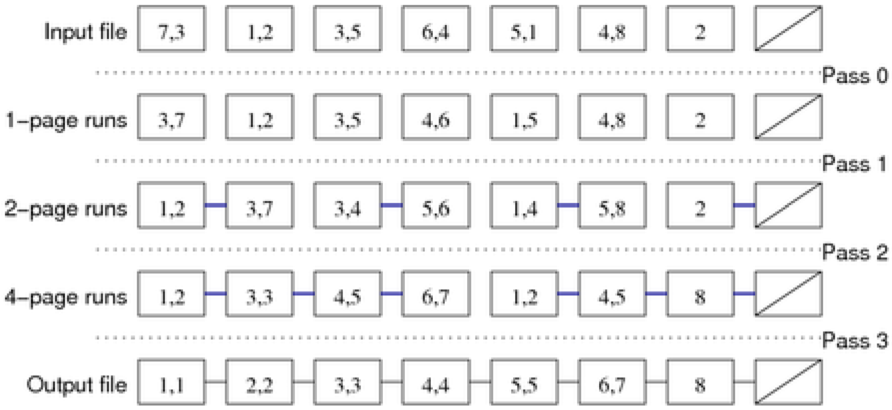
- eliminating duplicate tuples for projection
- ordering files to enhance select efficiency
- implementing various styles of join
- forming tuple groups in `group by`

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

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

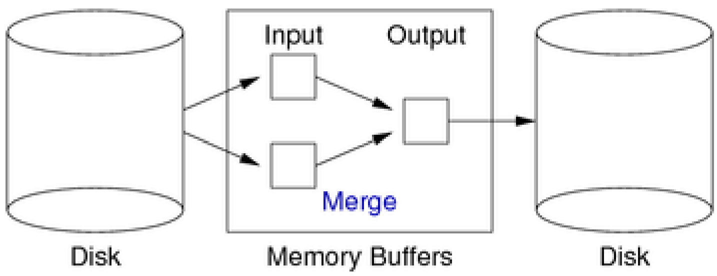
Two-way Merge Sort

Example:



... Two-way Merge Sort

Requires three in-memory buffers:



Assumption: cost of Merge operation on two in-memory buffers ≅ *0*.

Comparison for Sorting

Above assumes that we have a function to compare tuples.

Needs to understand ordering on different data types.

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

```
int tupCompare(r1,r2,f)
{
   if (r1.f < r2.f) return -1;
   if (r1.f > r2.f) return 1;
   return 0;
}
```

Assume =, <, > are available for all attribute types.

... Comparison for Sorting

14/106

In reality, need to sort on multiple attributes and ASC/DESC, e.g.

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

多中种要求排序

Sketch of multi-attribute sorting function

```
int tupCompare(r1,r2,criteria)
{
   foreach (f,ord) in criteria {
      if (ord == ASC) {
         if (r1.f < r2.f) return -1;
         if (r1.f > r2.f) return 1;
      }
      else {
         if (r1.f > r2.f) return -1;
         if (r1.f < r2.f) return 1;
      }
   }
   return 0;
}
```

Cost of Two-way Merge Sort

15/106

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 $c=50$ $\Rightarrow$ $b=2000$ pages.

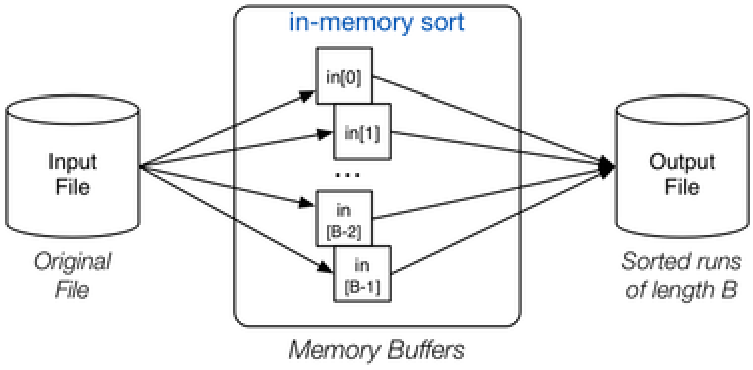
Number of passes for sort: $ceil(log_2 2000) = 11$

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

n-Way Merge Sort

16/106

Initial pass uses: *B* total buffers

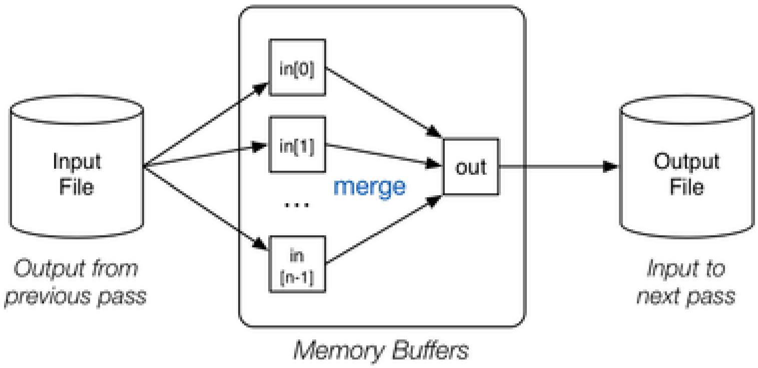


Reads *B* pages at a time, sorts in memory, writes out in order

... n-Way Merge Sort

17/106

Merge passes use: *n* input buffers, *1* output buffer



... n-Way Merge Sort

18/106

Method:

```
// Produce B-page-long runs
for each group of B pages in Rel {
    read B pages into memory buffers
    sort group in memory
    write B pages out to Temp
}
// Merge runs until everything sorted
numberOfRuns = ⌈b/B⌉
while (numberOfRuns > 1) {
    // n-way merge, where n=B-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 = ⌈numberOfRuns/n⌉
    Temp = newTemp // swap input/output files
}
```

Cost of n-Way Merge Sort

19/106

Consider file where $b = 4096$, $B = 16$ total buffers:

- pass 0 produces 256 × 16-page sorted runs
- pass 1
  - performs 15-way merge of groups of 16-page sorted runs
  - produces 18 × 240-page sorted runs (17 full runs, 1 short run)
- pass 2
  - performs 15-way merge of groups of 240-page sorted runs
  - produces 2 × 3600-page sorted runs (1 full run, 1 short run)
- pass 3
  - performs 15-way merge of groups of 3600-page sorted runs
  - produces 1 × 4096-page sorted runs

(cf. two-way merge sort which needs 11 passes)

... Cost of n-Way Merge Sort

20/106

Generalising from previous example ...

For b data pages and B buffers

- first pass: read/writes b pages, gives $b_0 = \lceil b/B \rceil$ runs
- then need $\lceil log_n b_0 \rceil$ passes until sorted
- each pass reads and writes b pages (i.e. $2.b$ page accesses)

$Cost = 2.b.(1 + \lceil log_n b_0 \rceil)$, where $b_0 = \lceil b/B \rceil$

Exercise 2: Cost of n-Way Merge Sort

21/106

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

试一下

Consider for the cases:

- 9 total buffers, 8 input buffers, 1 output buffer
- 33 total buffers, 32 input buffers, 1 output buffer
- 257 total buffers, 256 input buffers, 1 output buffer

Sorting in PostgreSQL

post版本的sort

Sort uses a merge-sort (from Knuth) similar to above:

- backend/utils/sort/tuplesort.c
- include/utils/sortsupport.h

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

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

Sorting comparison operators are obtained via catalog (in *Type*.o):

```
// gets pointer to function via pg_operator
struct Tuplesortstate { ... SortTupleComparator ... };

// returns negative, zero, positive
ApplySortComparator(Datum datum1, bool isnull1,
                    Datum datum2, bool isnull2,
                    SortSupport sort_helper);
```

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

`ApplySortComparator()` is PostgreSQL's version of `tupCompare()`

Implementing Projection

The Projection Operation

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

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
   - essentially, make a new tuple with fewer attributes
     and where the values may be computed from existing attributes
3. eliminate any duplicates produced (if `distinct`)
   - two approaches: sorting or hashing

Sort-based Projection

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 3: Cost of Sort-based Projection

29/106

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

3种projection

30/106

The costs involved are (assuming $B=n+1$ buffers for sort):

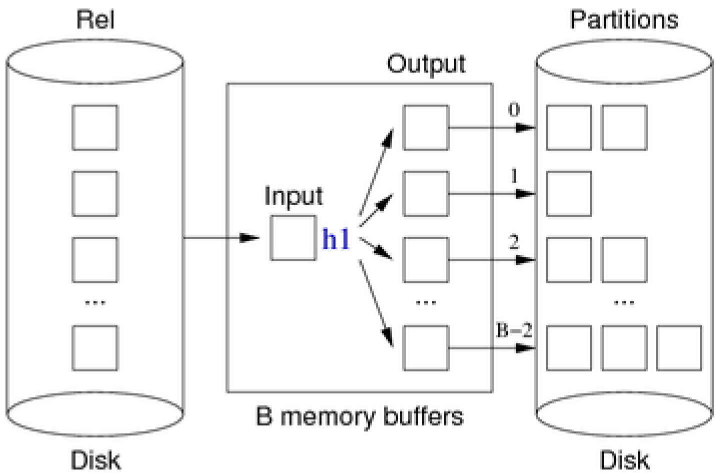
- scanning original relation `Rel`: b_R (with c_R)
- writing `Temp` relation: b_T (smaller tuples, $c_T > c_R$, *sorted*)
- sorting `Temp` relation:
  $2.b_T.(1+ceil(log_n b_0))$ where $b_0 = ceil(b_T/B)$
- scanning `Temp`, removing duplicates: b_T
- writing the result relation: b_{Out} (maybe less tuples)

Cost = sum of above = $b_R + b_T + 2.b_T.(1+ceil(log_n b_0)) + b_T + b_{Out}$

Hash-based Projection

31/106

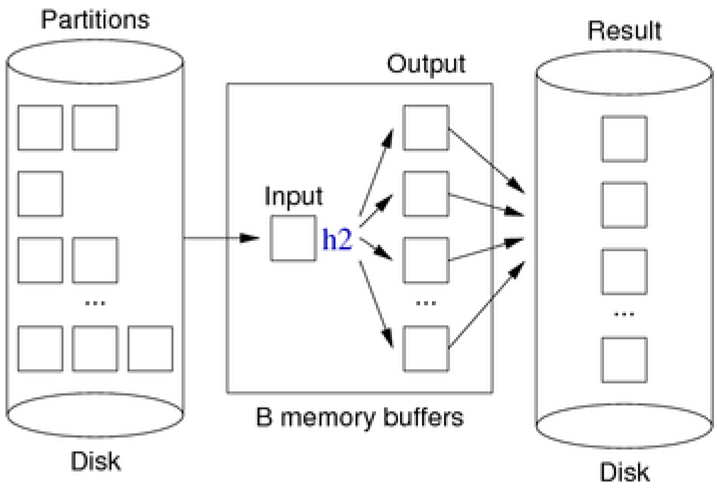
Partitioning phase:



... Hash-based Projection

32/106

Duplicate elimination phase:



... Hash-based Projection

33/106

Algorithm for both phases:

```
for each tuple T in relation Rel {
    T' = mkTuple([attrs],T)
    H = h1(T', n)
    B = buffer for partition[H]
    if (B full) write and clear B
    insert T' into 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 4: Cost of Hash-based Projection

34/106

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

35/106

The total cost is the sum of the following:

- scanning original relation `R`: 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 $sqrt(b_R)+1$ buffers
- if insufficient buffers, significant re-reading overhead

Projection on Primary Key

36/106

No duplicates, so the above approaches are not required.

Method:

```
bR = nPages(Rel)
for i in 0 .. bR-1 {
   P = read page i
   for j in 0 .. nTuples(P) {
```

```
        T = getTuple(P,j)
        T' = mkTuple(pk, T)
        if (outBuf is full) write and clear
        append T' to outBuf
    }
}
if (nTuples(outBuf) > 0) write
```

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

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

Code for projection forms part of execution iterators:

- backend/executor/execQual.c

Functions involved with projection:

- **`ExecProject(projInfo,...)`** ... extracts projected data
- **`check_sql_fn_retval(...)`** ... makes new tuple via TargetList
- **`ExecStoreTuple(newTuple,...)`** ... save tuple in buffer

plus many many others ...

Implementing Selection

Varieties of Selection

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

Examples of different selection types:

- *one*: `select * from` *R* `where id = 1234`
- *pmr*: `select * from` *R* `where age=65` (1-d)

  `select * from` *R* `where age=65 and gender='m'` (n-d)
- *rng*: `select * from` *R* `where age≥18 and age≤21` (1-d)

  `select * from` *R* `where age between 18 and 21` (n-d)
  `and height between 160 and 190`

  note: *rng* = *range*

Exercise 5: Query Types

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

44/106

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.

Selection in Heaps

46/106

For all selection queries, the only possible strategy is:

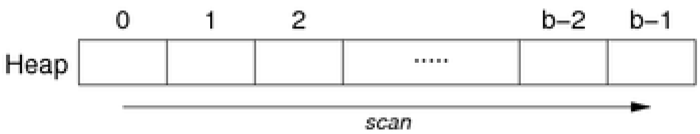
```
// select * from R where C
for each page P in file of relation R {
    for each tuple t in page P {
        if (t satisfies C)
            add tuple t to result set
    }
}
```

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

... Selection in Heaps

47/106

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

48/106

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

```
rel = openRelation("R", READ|WRITE);
pid = nPages(rel)-1;
get_page(rel, pid, buf);
if (size(newTup) > size(buf))
   { deal with oversize tuple }
else {
   if (!hasSpace(buf,newTup))
      { pid++; nPages(rel)++; clear(buf); }
   insert_record(buf,newTup);
   put_page(rel, pid, buf);
}
```

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

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

... 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

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

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

Implementation of deletion:

```
rel = openRelation("R",READ|WRITE);
for (p = 0; p < nPages(rel); p++) {
    get_page(rel, p, buf);
    ndels = 0;
    for (i = 0; i < nTuples(buf); i++) {
        tup = get_record(buf,i);
        if (tup satisfies Condition)
            { ndels++; delete_record(buf,i); }
    }
    if (ndels > 0) put_page(rel, p, buf);
    if (ndels > 0 && unique) break;
}
```

Exercise 6: Cost of Deletion in Heaps

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

PostgreSQL tuple deletion:

lock

```
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

Vacuuming eventually compacts space in each page.

Updates in Heaps

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 tuple larger than old version (too big for page)

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

... Updates in Heaps

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

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

table data, index file 在 heap

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

PostgreSQL "heap file" may use multiple physical files

如果过大 会create a fork

- 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 68 in PostgreSQL v11 documentation

Sorted Files

Sorted Files

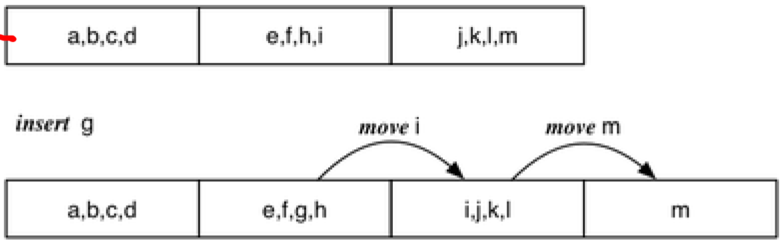
上下文

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

Makes searching more efficient; makes insertion less efficient

E.g. assume $c = 4$

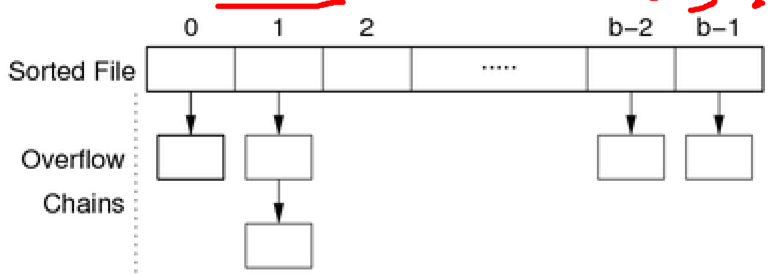
好多文件



... Sorted Files

In order to mitigate insertion costs, use overflow blocks.

防止大 insert



Total number of overflow blocks = 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 = b-1
while (lo <= hi) {
    mid = (lo+hi) / 2;  // int division with truncation
    (tup,loVal,hiVal) = searchBucket(f,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 `f` is file for relation, `mid,lo,hi` are page indexes,
`k` is a field/attr, `val,loVal,hiVal` are values for `k`

... Selection in Sorted Files

Search a page and its overflow chain for a key value

```
searchBucket(f,p,k,val)
{
    buf = getPage(f,p);
    (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) {
        buf = getPage(ovf,ovp);
        (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

Note: `getPage(f,pid) = { read_page(relOf(f),pid,buf); return buf; }`

... Selection in Sorted Files

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 = getTuple(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

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

Cases:

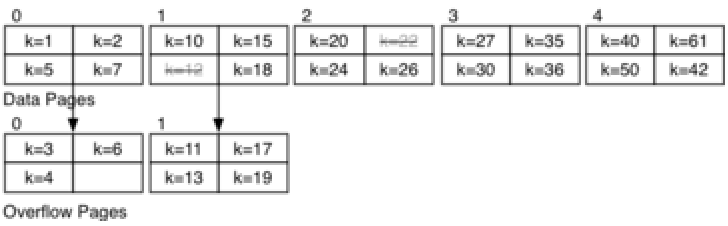
- best: find tuple in first data page we read
- worst: full binary search, and not found
    - examine $log_2 b$ data pages
    - plus examine all of their overflow pages
- average: examine some data pages + their overflow pages

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

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

Exercise 7: Searching in Sorted File

Consider this sorted file with overflows (*b=5, c=4*):



Compute the cost for answering each of the following:

- `select * from R where k = 24`
- `select * from R where k = 3`
- `select * from R where k = 14`
- `select max(k) from R`

Exercise 8: Optimising Sorted-file Search

66/106

The `searchBucket(f,p,k,val)` function requires:

- read the p^{th} page from data file
- scan it to find a match and min/max k values in page
- while no match, repeat the above for each overflow page
- if we find a match in any page, return it
- otherwise, remember min/max over all pages in bucket

Suggest an optimisation that would improve `searchBucket()` performance for most buckets.

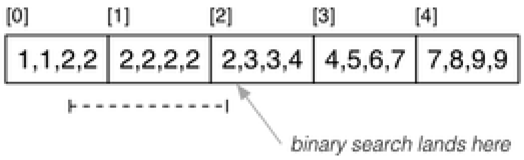
... Selection in Sorted Files

67/106

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

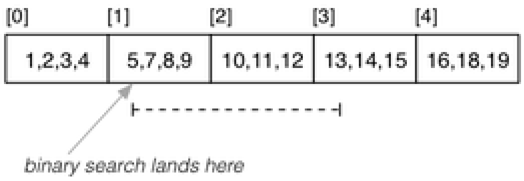
... Selection in Sorted Files

68/106

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`



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

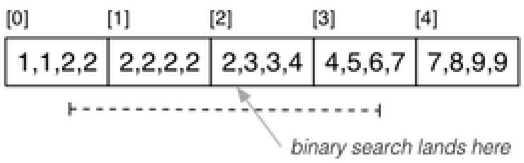
... Selection in Sorted Files

69/106

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

70/106

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

71/106

Insertion approach:

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

[Diagram:Pics/file-struct/sorted-file1-small.png]

Deletion from Sorted Files

72/106

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)

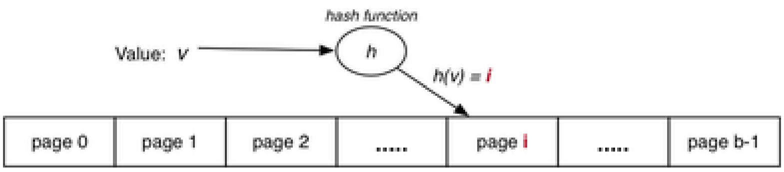
Thus, $Cost_{delete} = Cost_{select} + b_{qw}$

Hashed Files

Hashing

74/106

Basic idea: use key value to compute page address of tuple.



e.g. tuple with key = *v* is stored in page *i*

Requires: hash function *h(v)* that maps *KeyDomain* → *[0..b-1]*.

- hashing converts key value (any type) into integer value
- integer value is then mapped to page index
- note: can view integer value as a bit-string

... Hashing

75/106

PostgreSQL hash function (simplified):

```
Datum hash_any(unsigned char *k, register int keylen)
{
   register uint32 a, b, c, len;
   /* Set up the internal state */
   len = keylen;  a = b = c = 0x9e3779b9 + len + 3923095;
   /* handle most of the key */
   while (len >= 12) {
      a += ka[0]; b += ka[1]; c += ka[2];
      mix(a, b, c);
      ka += 3;  len -= 12;
   }
   /* collect any data from last 11 bytes into a,b,c */
   mix(a, b, c);
   return UInt32GetDatum(c);
}
```

See **`backend/access/hash/hashfunc.c`** for details (incl `mix()`)

... Hashing

76/106

`hash_any()` gives hash value as 32-bit quantity (`uint32`).

Two ways to map raw hash value into a page address:

- if $b = 2^k$, bitwise AND with *k* low-order bits set to one

  ```
  uint32 hashToPageNum(uint32 hval) {
      uint32 mask = 0xFFFFFFFF;
      return (hval & (mask >> (32-k)));
  }
  ```

- otherwise, use *mod* to produce value in range *0..b-1*

  ```
  uint32 hashToPageNum(uint32 hval) {
      return (hval % b);
  }
  ```

Hashing Performance

77/106

Aims:

- distribute tuples evenly amongst buckets
- have most buckets nearly full (attempt to minimise wasted space)

Note: if data distribution not uniform, address distribution can't be uniform.

Best case: every bucket contains same number of tuples.

Worst case: every tuple hashes to same bucket.

Average case: some buckets have more tuples than others.

Use overflow pages to handle "overfull" buckets (cf. sorted files)

All tuples in each bucket must have same hash value.

... Hashing Performance

78/106

Two important measures for hash files:

- load factor: $L = r / bc$
- average overflow chain length: $Ov = b_{ov} / b$

Three cases for distribution of tuples in a hashed file:

Case	L	Ov
Best	$\cong 1$	0
Worst	$>> 1$	**
Average	< 1	$0<Ov<1$

(** performance is same as Heap File)

To achieve average case, aim for $0.75 \le L \le 0.9$.

Selection with Hashing

79/106

Select via hashing on unique key *k* (*one*)

```
// select * from R where k = val
pid,P = getPageViaHash(val,R)
for each tuple t in page P {
    if (t.k == val) return t
}
for each overflow page Q of P {
    for each tuple t in page Q {
        if (t.k == val) return t
}   }
```

$Cost_{one}$: Best = 1, Avg = $1+Ov/2$ Worst = $1+max(OvLen)$

... Selection with Hashing

80/106

Select via hashing on non-unique hash key *nk* (*pmr*)

```
// select * from R where nk = val
pid,P = getPageViaHash(val,R)
for each tuple t in page P {
    if (t.nk == val) add t to results
}
for each overflow page Q of P {
    for each tuple t in page Q {
        if (t.nk == val) add t to results
}   }
return results
```

$Cost_{pmr} = 1 + Ov$

... Selection with Hashing

81/106

Hashing does not help with *range* queries** ...

$Cost_{range} = b + b_{ov}$

Selection on attribute *j* which is not hash key ...

$Cost_{one}$, $Cost_{range}$, $Cost_{pmr} = b + b_{ov}$

** unless the hash function is order-preserving (and most aren't)

Insertion with Hashing

82/106

Insertion uses similar process to *one* queries.

```
// insert tuple t with key=val into rel R
pid,P = getPageViaHash(val,R)
```

```
if room in page P {
    insert t into P; return
}
for each overflow page Q of P {
    if room in page Q {
        insert t into Q; return
}   }
add new overflow page Q
link Q to previous page
insert t into Q
```

$Cost_{insert}$: Best: $1_r + 1_w$ Worst: $1+max(OvLen))_r + 2_w$

Exercise 9: Insertion into Static Hashed File

83/106

Consider a file with *b=4*, *c=3*, *d=2*, *h(x) = bits(d,hash(x))*

Insert tuples in alpha order with the following keys and hashes:

k	hash(k)		k	hash(k)		k	hash(k)		k	hash(k)
a	10001		g	00000		m	11001		s	01110
b	11010		h	00000		n	01000		t	10011
c	01111		i	10010		o	00110		u	00010
d	01111		j	10110		p	11101		v	11111
e	01100		k	00101		q	00010		w	10000
f	00010		l	00101		r	00000		x	00111

The hash values are the 5 lower-order bits from the full 32-bit hash.

Deletion with Hashing

84/106

Similar performance to select on non-unique key:

```
// delete from R where k = val
// f = data file ... ovf = ovflow file
pid,P = getPageViaHash(val,R)
ndel = delTuples(P,k,val)
if (ndel > 0) putPage(f,P,pid)
for each overflow page qid,Q of P {
    ndel = delTuples(Q,k,val)
    if (ndel > 0) putPage(ovf,Q,qid)
}
```

Extra cost over select is cost of writing back modified blocks.

Method works for both unique and non-unique hash keys.

Problem with Hashing...

85/106

So far, discussion of hashing has assumed a fixed file size (*b*). ← fixed

What size file to use?

- the size we need right now (performance degrades as file overflows)
- the maximum size we might ever need (signifcant waste of space)

Change file size ⇒ change hash function ⇒ rebuild file

Methods for hashing with dynamic files:

- extendible hashing, dynamic hashing (need a directory, no overflows)
- *linear hashing* (expands file "sytematically", no directory, has overflows)

... Problem with Hashing...

← / 针对 flexible

86/106

All flexible hashing methods ...

- treat hash as 32-bit bit-string
- adjust hashing by using more/less bits

Start with hash function to convert value to bit-string:

`uint32 hash(unsigned char *val)`

Require a function to extract *d* bits from bit-string:

`unit32 bits(int d, uint32 val)`

Use result of `bits()` as page address.

Exercise 10: Bit Manipulation

87/106

1. Write a function to display `uint32` values as `01010110...`

```
char *showBits(uint32 val, char *buf);
```

Analogous to `gets()` (assumes supplied buffer large enough)

2. Write a function to extract the *d* bits of a `uint32`

```
uint32 bits(int d, uint32 val);
```

If $d > 0$, gives low-order bits; if $d < 0$, gives high-order bits

---

... Problem with Hashing...

Important concept for flexible hashing: *splitting*

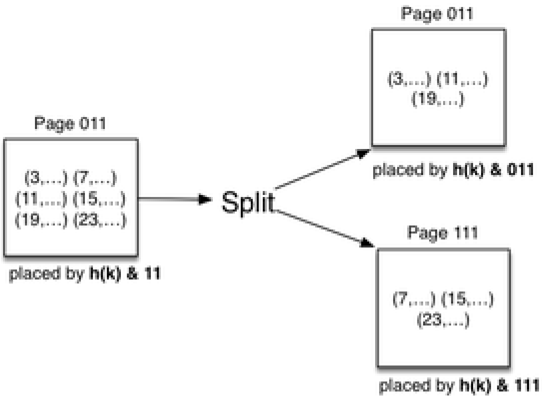
- consider one page (all tuples have same hash value)
- recompute page numbers by considering one extra bit
- if current page is `101`, new pages have hashes `0101` and `1101`
- some tuples stay in page `0101` (was `101`)
- some tuples move to page `1101` (new page)
- also, rehash any tuples in overflow pages of page `101`

Result: expandable data file, never requiring a complete file rebuild

---

... Problem with Hashing...

Example of splitting:



Tuples only show key value; assume *h(val) = val*

Linear Hashing

File organisation:

- file of primary data blocks
- file of overflow data blocks
- a register called the *split pointer* (sp)

Uses systematic method of growing data file ...

- hash function "adapts" to changing address range
- systematic splitting controls length of overflow chains

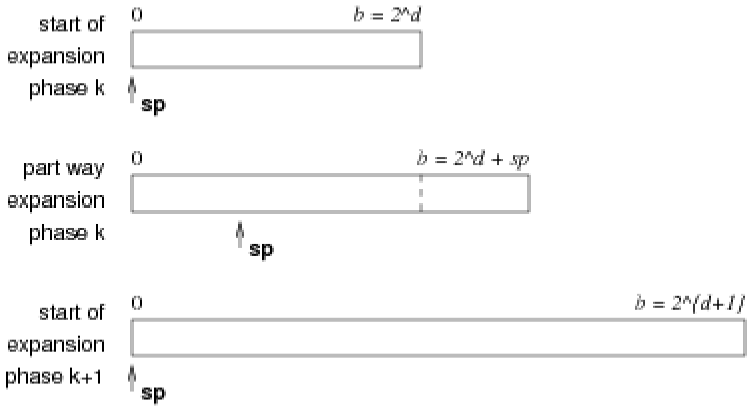
Advantage: does *not* require auxiliary storage for a directory

Disadvantage: requires overflow pages (don't split on full pages)

... Linear Hashing

File grows linearly (one block at a time, at regular intervals).

Has "phases" of expansion; over each phase, *b* doubles.



Selection with Lin.Hashing

If $b=2^d$, the file behaves exactly like standard hashing.

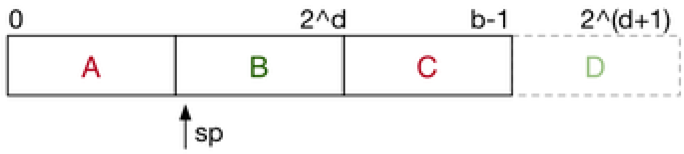
Use *d* bits of hash to compute block address.

```
// select * from R where k = val
h = hash(val);
P = bits(d,h);  // lower-order bits
for each tuple t in page P
         and its overflow pages {
    if (t.k == val) return t;
}
```

Average $Cost_{one}$ = $1+Ov$

... Selection with Lin.Hashing

If $b \mathrel{!}= 2^d$, treat different parts of the file differently.



Parts *A* and *C* are treated as if part of a file of size 2^{d+1}.

Part *B* is treated as if part of a file of size 2^d.

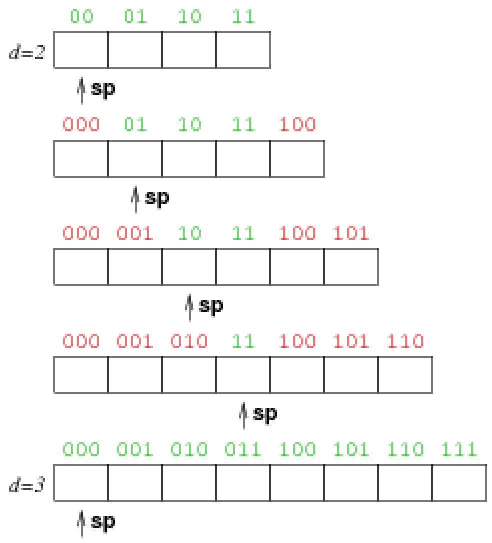
Part *D* does not yet exist (tuples in *B* may move into it).

... Selection with Lin.Hashing

Modified search algorithm:

```
// select * from R where k = val
h = hash(val);
p = bits(d,h);
if (P < sp) { p = bits(d+1,h); }
P = getPage(f, p)
for each tuple t in page P
         and its overflow blocks {
    if (t.k == val) return R;
}
```

File Expansion with Lin.Hashing



Insertion with Lin.Hashing

Abstract view:

```
p = bits(d,hash(val));
if (p < sp) P = bits(d+1,hash(val));
// bucket P = page P + its overflow pages
P = getPage(f,p)
for each page Q in bucket P {
    if (space in Q) {
        insert tuple into Q
        break
    }
}
if (no insertion) {
```

```
    add new ovflow page to bucket P
    insert tuple into new page
}
if (need to split) {
    partition tuples from bucket sp
          into buckets sp and sp+2^d
    sp++;
    if (sp == 2^d) { d++; sp = 0; }
}
```

Splitting

How to decide that we "need to split"?

Two approaches to triggering a split:

- split every time a tuple is inserted into full block
- split when load factor reaches threshold (every *k* inserts)

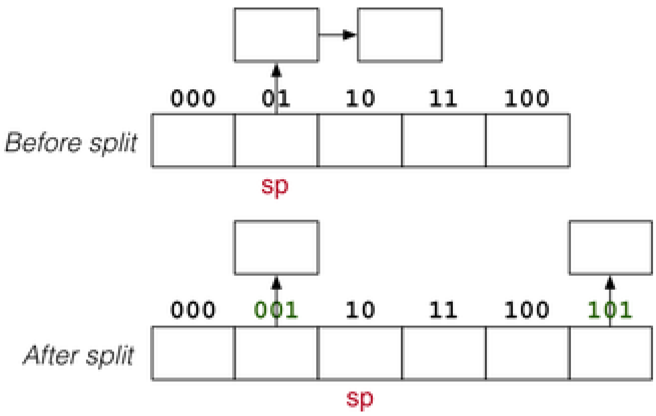
Note: always split block *sp*, even if not full/"current"

Systematic splitting like this ...

- eventually reduces length of every overflow chain
- helps to maintain short average overflow chain length

... Splitting

Splitting process for block *sp*=`01`:



Exercise 11: Insertion into Linear Hashed File

Consider a file with *b=4*, *c=3*, *d=2*, *sp=0*, *hash(x)* as above

Insert tuples in alpha order with the following keys and hashes:

k	*hash(k)*		*k*	*hash(k)*		*k*	*hash(k)*		*k*	*hash(k)*
a	10001		g	00000		m	11001		s	01110
b	11010		h	00000		n	01000		t	10011
c	01111		i	10010		o	00110		u	00010
d	01111		j	10110		p	11101		v	11111
e	01100		k	00101		q	00010		w	10000
f	00010		l	00101		r	00000		x	00111

The hash values are the 5 lower-order bits from the full 32-bit hash.

... Splitting

Splitting algorithm:

```
// partition tuples between two buckets
newp = sp + 2^d; oldp = sp;
for all tuples t in P[oldp] and its overflows {
    p = bits(d+1,hash(t.k));
    if (p == newp)
        add tuple t to bucket[newp]
    else
        add tuple t to bucket[oldp]
}
sp++;
if (sp == 2^d) { d++; sp = 0; }
```

Insertion Cost

If no split required, cost same as for standard hashing:

$Cost_{insert}$:   Best: $1_r + 1_w$,   Avg: $(1+Ov)_r + 1_w$,   Worst: $(1+max(Ov))_r + 2_w$

If split occurs, incur $Cost_{insert}$ plus cost of splitting:

- read block sp (plus all of its overflow blocks)
- write block sp (and its new overflow blocks)
- write block $sp+2^d$ (and its new overflow blocks)

On average, $Cost_{split} = (1+Ov)_r + (2+Ov)_w$

Deletion with Lin.Hashing

Deletion is similar to ordinary static hash file.

But might wish to contract file when enough tuples removed.

Rationale: r shrinks, b stays large ⇒ wasted space.

Method:

- remove last bucket in data file (contracts linearly).
- merge tuples from bucket with its buddy page (using d-1 hash bits)

Hash Files in PostgreSQL

PostgreSQL uses linear hashing on tables which have been:

`create index Ix on R using hash (k);`

Hash file implementation: **`backend/access/hash`**

- **`hashfunc.c`** ... a family of hash functions
- **`hashinsert.c`** ... insert, with overflows
- **`hashpage.c`** ... utilities + splitting
- **`hashsearch.c`** ... iterator for hash files

Based on "A New Hashing Package for Unix", Margo Seltzer, Winter Usenix 1991

... Hash Files in PostgreSQL

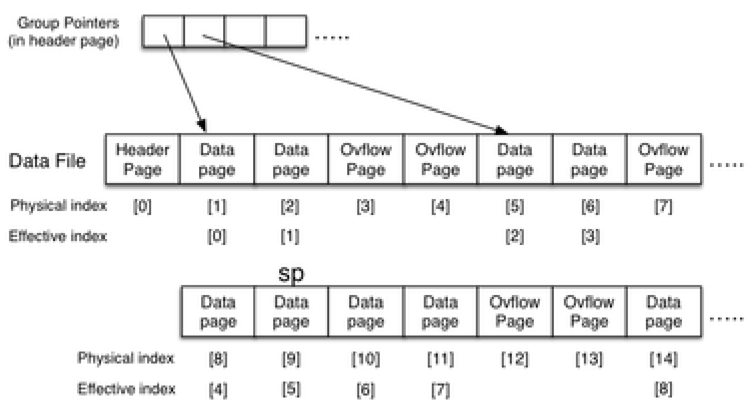
PostgreSQL uses slightly different file organisation ...

- has a single file containing main and overflow pages
- has groups of main pages of size 2^n
- in between groups, arbitrary number of overflow pages
- maintains collection of "split pointers" in header page
- each split pointer indicates start of main page group

If overflow pages become empty, add to free list and re-use.

... Hash Files in PostgreSQL

PostgreSQL hash file structure:



... Hash Files in PostgreSQL

Converting bucket # to page address:

```
// which page is primary page of bucket
uint bucket_to_page(headerp, B) {
   uint *splits = headerp->hashm_spares;
   uint chunk, base, offset, lg2(uint);
   chunk = (B<2) ? 0 : lg2(B+1)-1;
   base = splits[chunk];
   offset = (B<2) ? B : B-(1<<chunk);
   return (base + offset);
```

```
}
// returns ceil(log_2(n))
int lg2(uint n) {
        int i, v;
        for (i = 0, v = 1; v < n; v <<= 1) i++;
        return i;
}
```
