# Week 04 Lectures

## Exercise 1: PostgreSQL Tuple Visibility

1/110

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 determine whether a tuple is visible?

Assume: multiple concurrent transactions on tables.

tuple = (oid, xmin, xmax, cmin, cmax, infomask, ...rest of data...)

For all of the details:

- PG_SRC/include/access/htup.h ... tuple data structure
- PG_SRC/include/utils/snapshot.h ... "snapshot" data
- PG_SRC/backend/utils/time/tqual.c ... visibility checks

---

## Scanning in PostgreSQL

2/110

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(tup, desc, keys, ...)`** ... implements key match test

---

### ... Scanning in PostgreSQL

3/110

```
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

4/110

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

---

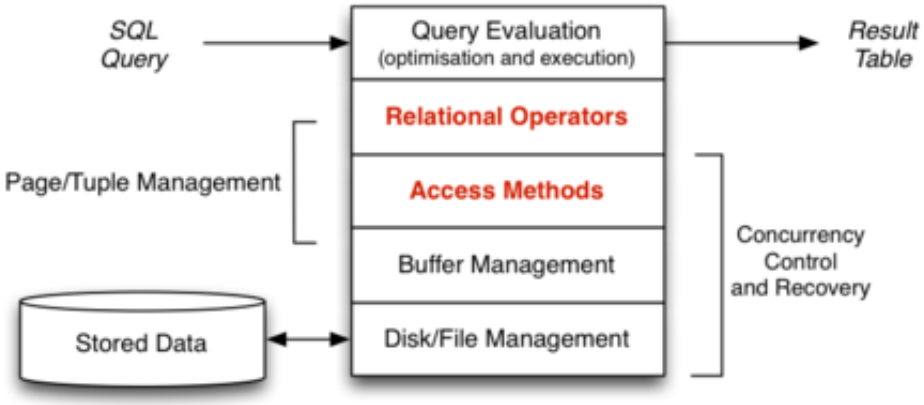
# Implementing Relational Operations

---

## Implementing Relational Operators

6/110

Implementation of relational operations in DBMS:




---

### ... Implementing Relational Operators

7/110

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

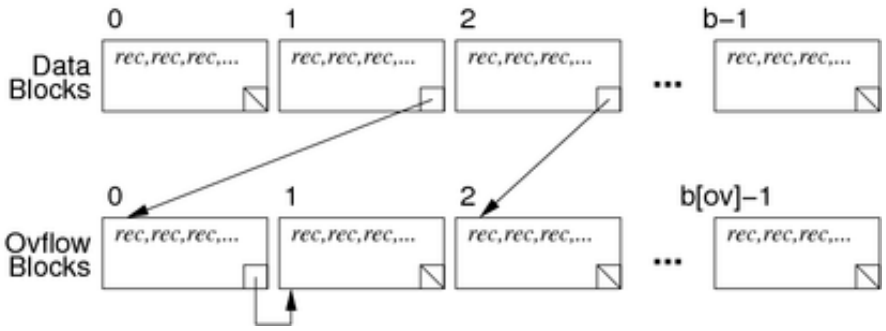
---

### ... Implementing Relational Operators

8/110

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*




---

## Reminder on Cost Analyses

9/110

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

11/110

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.

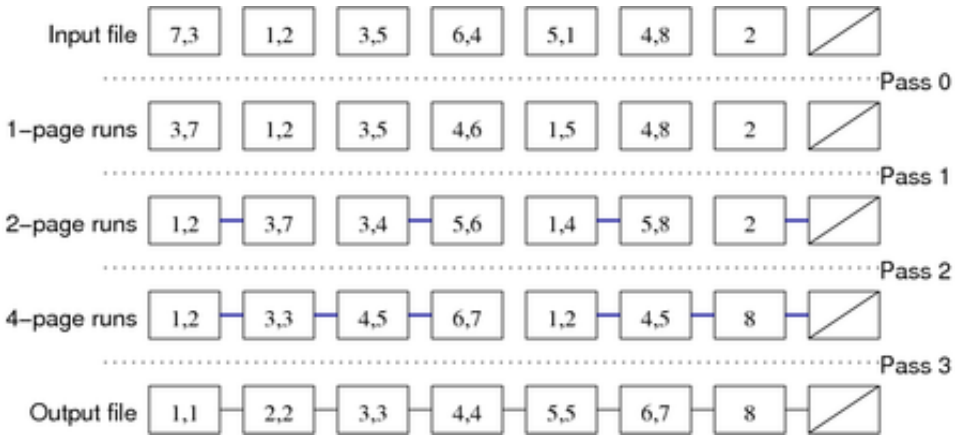
For large data on disks, use external sorts such as *merge sort*.

---

## Two-way Merge Sort

12/110

Example:

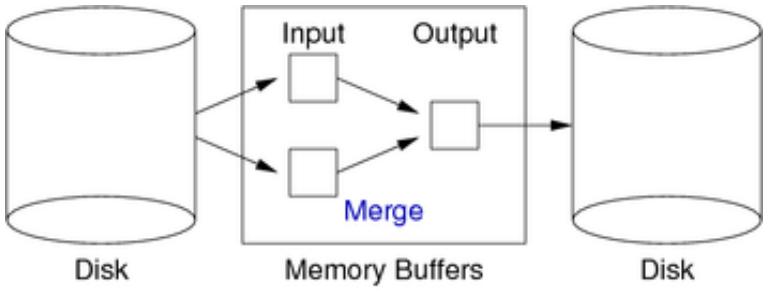



---

### ... Two-way Merge Sort

13/110

Requires three in-memory buffers:




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

---

## Comparison for Sorting

14/110

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 < and > are overloaded for all attribute types.

---

### ... Comparison for Sorting



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
```

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



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*

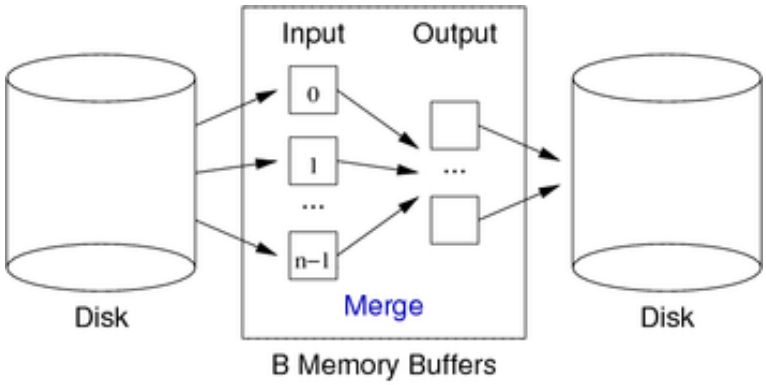
Reads/writes entire file 11 times! Can we do better?

---

## n-Way Merge Sort



Merge passes use: *B* memory buffers, *n* input buffers, *B-n* output buffers



Typically, consider only one output buffer, i.e. $B = n + 1$

---

### ... n-Way Merge Sort



Method:

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

---

### ... n-Way Merge Sort



Method for merging n runs (n input buffers, 1 output buffer):

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

---

## Cost of n-Way Merge Sort



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

- pass 0 produces 256 × 16-page sorted runs
- pass 1 produces 18 × 240-page sorted runs
- pass 2 produces 2 × 3600-page sorted run
- pass 3 produces 1 × 4096-page sorted run

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

For $b$ data pages and $n=15$ input buffers (15-way merge)

- 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 ($2.b$)

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

---

## Exercise 2: Cost of n-Way Merge Sort



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:

- 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



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



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 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.

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

---

# Implementing Projection

---

## The Projection Operation

26/110

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

27/110

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

28/110

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

29/110

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

30/110

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 $b_0 = ceil(b_T/(n+1))$
- scanning `Temp`, removing duplicates: $b_T$
- writing the result relation: $b_{Out}$ (maybe less tuples)

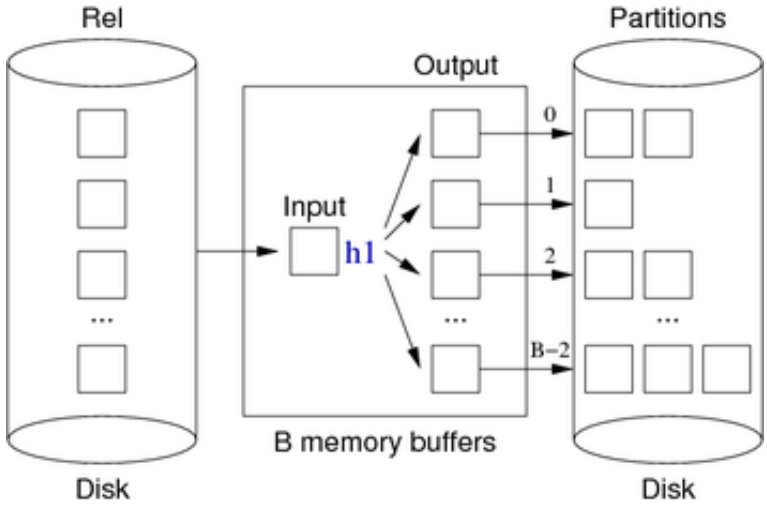
Cost = sum of above = $b_R + b_T + 2.b_T.ceil(\log_n b_0) + b_T + b_{Out}$

---

## Hash-based Projection

31/110

Partitioning phase:

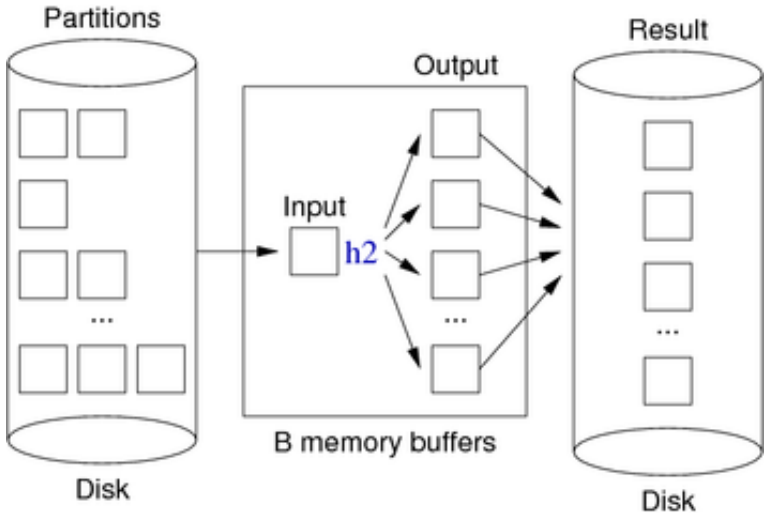



---

## ... Hash-based Projection

32/110

Duplicate elimination phase:




---

## ... Hash-based Projection

33/110

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

34/110

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

35/110

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

36/110

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

37/110

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

38/110

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

39/110

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

41/110

*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



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

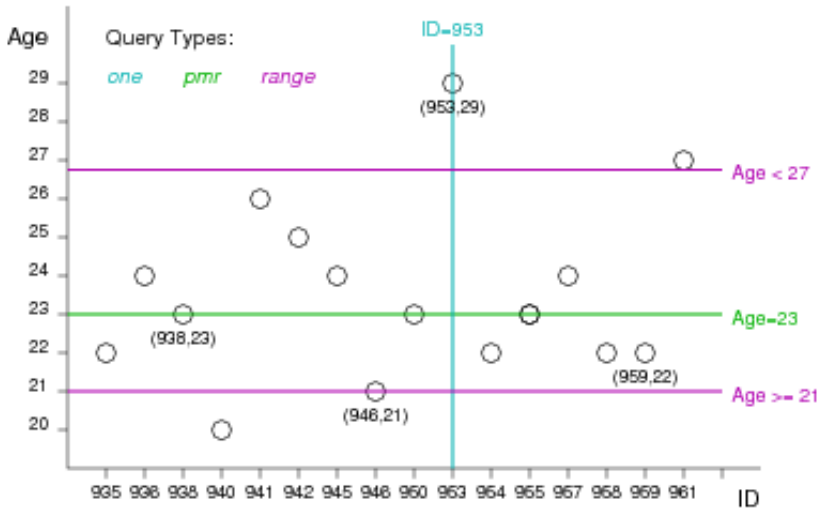
- assume relation *R* with attributes $a_1,...,a_n$
- attribute domains of *R* specify a *n*-dimensional space
- each tuple $(v_1,v_2,...,v_n) \in R$ is a point in that space
- queries specify values/ranges on $N \geq 1$ dimensions
- a query defines a point/line/plane/region of the *n*-d space
- results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






---

## ... Varieties of Selection



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 5: Query Types

45/110

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

46/110

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.

---

## Recap on Implementing Selection

47/110

*Selection* = `select * from` *R* `where` *C*

- yields a subset of *R* tuples satisfying condition *C*
- a very important (frequent) operation in relational databases

Types of selection determined by type of condition

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

---

### ... Recap on Implementing Selection



Strategies for implementing selection efficiently

- arrangement of tuples in file (e.g. sorting, hashing)
- auxiliary data structures (e.g. indexes, signatures)

Interested in cost for `select`, `delete`, `update`, and `insert`

- for `select`, simply count number of pages read $n_r$
- for others, use $n_r$ and $n_w$ to distinguish reads/writes

Typical file structure has

- $b$ main data pages, $b_{Ov}$ overflow pages, $c$ tuples per page
- auxiliary files with e.g. oversized values, index entries

---

# 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



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
    }
}
```

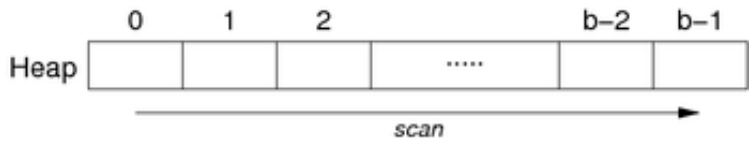
i.e. linear scan through file searching for matching tuples

---

### ... 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

52/110

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); }  // add a new page
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

53/110

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

54/110

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

55/110

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 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:

```
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 version of 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).

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

- 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

---

# Sorted Files

---

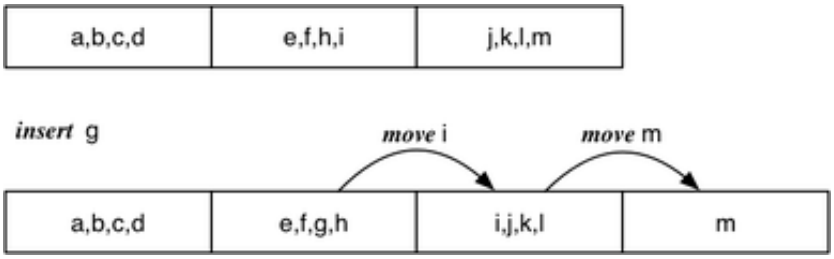
## Sorted Files



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*

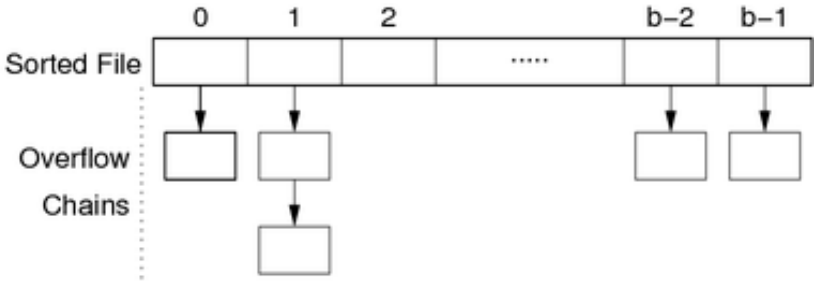



---

### ... Sorted Files

64/110

In order to mitigate insertion costs, use overflow blocks.




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

65/110

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,k,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

66/110

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

---

### ... Selection in Sorted Files

67/110

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

68/110

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 relies on assumptions (e.g. data distribution)

---

## Exercise 7: Searching in Sorted File

69/110

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



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



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

---

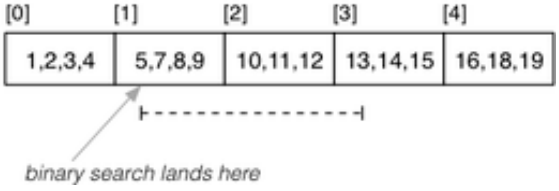
### ... Selection in Sorted Files



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`



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

---

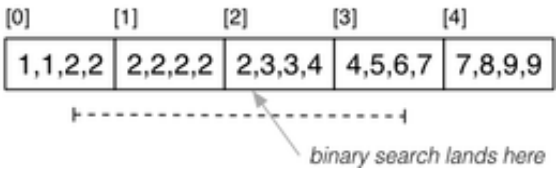
### ... Selection in Sorted Files



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`




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

---

### ... Selection in Sorted Files



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

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

---

## Updates to Sorted Files



**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 $\delta_w = 1$ or 2)

**Deletion** strategy:

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

Cost depends on *selectivity* of selection condition

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

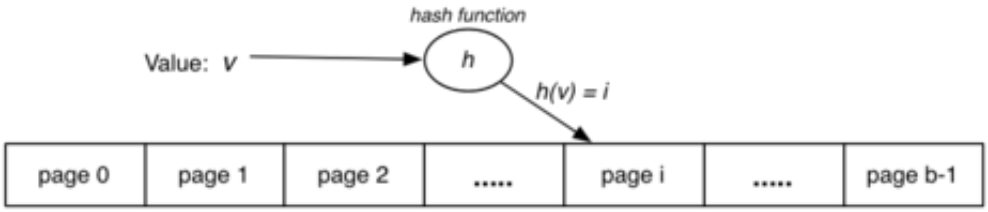
---

# Hashed Files

---

## Hashing

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

78/110

PostgreSQL hash function (simplified):

```
uint32 hash_any(unsigned char *k, register int keylen)
{
   register uint32 a, b, c, len;
   /* Set up the internal state */
   len = keylen;  a = b = 0x9e3779b9;  c = 3923095;
   /* handle most of the key */
   while (len >= 12) {
      a += (k[0] + (k[1]<<8) + (k[2]<<16) + (k[3]<<24));
      b += (k[4] + (k[5]<<8) + (k[6]<<16) + (k[7]<<24));
      c += (k[8] + (k[9]<<8) + (k[10]<<16) + (k[11]<<24));
      mix(a, b, c);   k += 12; len -= 12;
   }
   /* collect any data from last 11 bytes into a,b,c */
   mix(a, b, c);
   return c;
}
```

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

---

## ... Hashing

79/110

`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



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



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 $0.75 \leq L \leq 0.9$.

---

# Selection with Hashing



Best performance occurs for *one* queries on hash key field.

Basic strategy:

- compute page address via hash function *hash(val)*
- fetch that page and look for matching tuple
- possibly fetch additional pages from overflow chain

Best $Cost_{one} = 1$ (find in data page)

Average $Cost_{one} = 1+Ov/2$ (scan half of ovflow chain)

Worst $Cost_{one} = 1+max(OvLen)$ (find in last page of ovflow chain)

---

## ... Selection with Hashing

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

```
// select * from R where k = val
f = openFile(relName("R"),READ);
p = hash(val) % nPages(f);
buf = getPage(f, p)
for (i = 0; i < nTuples(buf); i++) {
    tup = getTuple(buf,i);
    if (tup.k == val) return tup;
}
ovp = ovflow(buf);
while (ovp != NO_PAGE) {
    buf = getPage(ovf,ovp);
    for (i = 0; i < nTuples(Buf); i++) {
        tup = getTuple(buf,i);
        if (tup.k == val) return tup;
}   }
```

---

### ... Selection with Hashing

84/110

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

```
// select * from R where k = val
f = openFile(relName("R"),READ);
p = hash(val) % nPages(f);
buf = getPage(f, p)
for (i = 0; i < nTuples(buf); i++) {
    tup = getTuple(buf,i);
    if (tup.k == val) append tup to results
}
ovp = ovflow(buf);
while (ovp != NO_PAGE) {
    buf = getPage(ovf,ovp);
    for (i = 0; i < nTuples(Buf); i++) {
        tup = getTuple(buf,i);
        if (tup.k == val) append tup to results
}   }
```

$Cost_{pmr} = 1 + Ov$

---

### ... Selection with Hashing

85/110

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

86/110

Insertion uses similar process to *one* queries.

```
// insert tuple t with key=val into rel R
// f = data file ... ovf = ovflow file
p = hash(val) % nPages(R)
P = getPage(f,p)
if (tup fits in page P)
    { insert t into P; return }
for each overflow page Q of P {
    if (tup fits in page Q)
        { insert t into Q; return }
}
add new overflow page Q
link Q to previous overflow 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

87/110

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

88/110

Similar performance to select:

```
// delete from R where k = val
// f = data file ... ovf = ovflow file
p = hash(val) % nPages(R)
buf = getPage(f,p)
ndel = delTuples(buf,k,val)
if (ndel > 0) putPage(f,buf,p)
p = ovFlow(buf)
while (p != NO_PAGE) {
    buf = getPage(ovf,p)
    ndel = delTuples(buf,k,val)
    if (ndel > 0) putPage(ovf,buf,p)
    p = ovFlow(buf)
}
```

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

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

---

## Problem with Hashing...



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

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...



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



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`

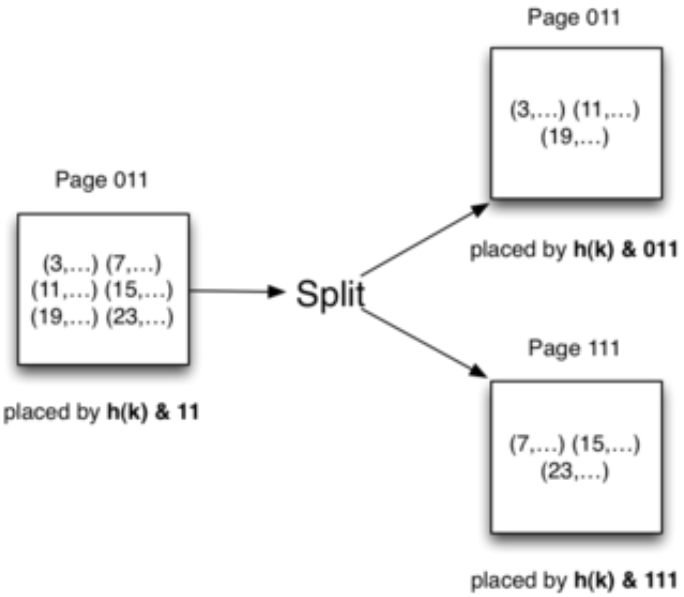
Result: expandable data file, never requiring a complete file rebuild

---

### ... Problem with Hashing...

93/110

Example of splitting:




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

---

## Linear Hashing

94/110

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 (splits don't occur on full pages)

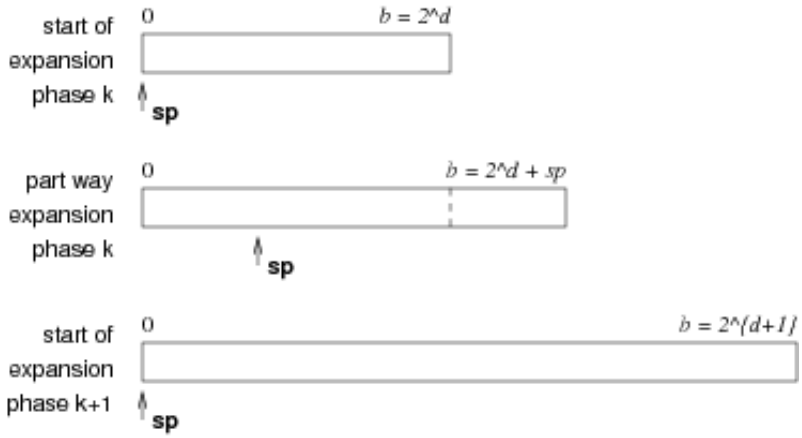
---

### ... Linear Hashing

95/110

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

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




---

## Selection with Lin.Hashing

96/110

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;
}
```

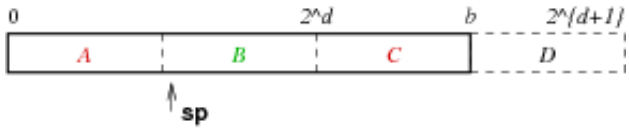
Average $Cost_{one} = 1+Ov$

---

### ... Selection with Lin.Hashing

97/110

If $b != 2^d$, 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

98/110

Modified search algorithm:

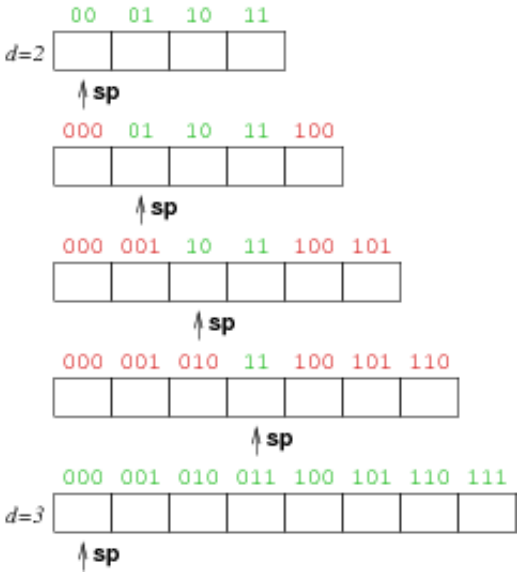
```
// select * from R where k = val
h = hash(val);
P = bits(d,h);
if (P < sp) { P = bits(d+1,h); }
for each tuple t in page P
         and its overflow blocks {
    if (t.k == val) return R;
```

```
}
```

---

## File Expansion with Lin.Hashing

99/110




---

## Insertion with Lin.Hashing

100/110

Abstract view:

```
P = bits(d,hash(val));
if (P < sp) P = bits(d+1,hash(val));
// bucket P = page P + its overflow pages
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

101/110

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 ...

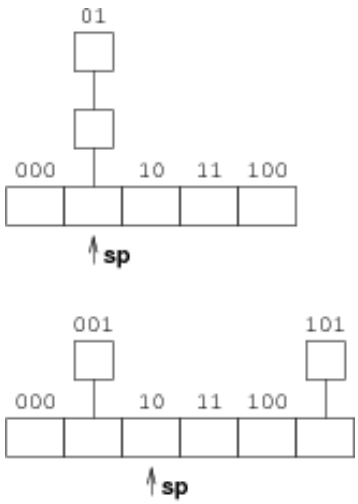
- eventually reduces length of every overflow chain
- helps to maintain short average overflow chain length

---

### ... 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



Detailed splitting algorithm:

```
// partitions tuples between two buckets
newp = sp + 2^d; oldp = sp;
buf = getPage(f,sp);
clear(oldBuf); clear(newBuf);
```

```
for (i = 0; i < nTuples(buf); i++) {
    tup = getTuple(buf,i);
    p = bits(d+1,hash(tup.k));
    if (p == newp)
        addTuple(newBuf,tup);
    else
        addTuple(oldBuf,tup);
}
p = ovflow(buf);  oldOv = newOv = 0;
while (p != NO_PAGE) {
    ovbuf = getPage(ovf,p);
    for (i = 0; i < nTuples(ovbuf); i++) {
        tup = getTuple(buf,i);
        p = bits(d+1,hash(tup.k));
        if (p == newp) {
            if (isFull(newBuf)) {
                nextp = nextFree(ovf);
                ovflow(newBuf) = nextp;
                outf = newOv ? f : ovf;
                writePage(outf, newp, newBuf);
                newOv++; newp = nextp; clear(newBuf);
            }
            addTuple(newBuf, tup);
        }
        else {
            if (isFull(oldBuf)) {
                nextp = nextFree(ovf);
                ovflow(oldBuf) = nextp;
                outf = oldOv ? f : ovf;
                writePage(outf, oldp, oldBuf);
                oldOv++; oldp = nextp; clear(oldBuf);
            }
            addTuple(oldBuf, tup);
        }
    }
    addToFreeList(ovf,p);
    p = ovflow(buf);
}
sp++;
if (sp == 2^d) { d++; sp = 0; }
```

---

## Insertion Cost

105/110

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

106/110

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).

Involves a coalesce procedure which is an inverse split.

---

## 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

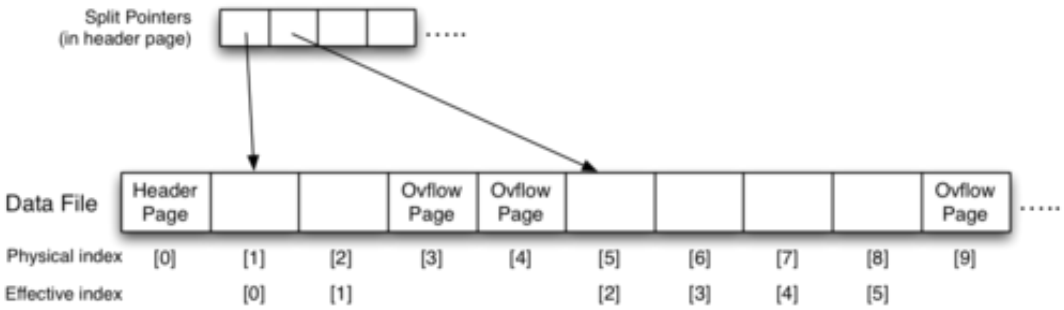
If overflow pages become empty, add to free list and re-use.

---

### ... 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;
}
```

---

Produced: 16 Aug 2018