course 4 Selection on One Attribute

1. Selection

we consider three distinct styles of selection:

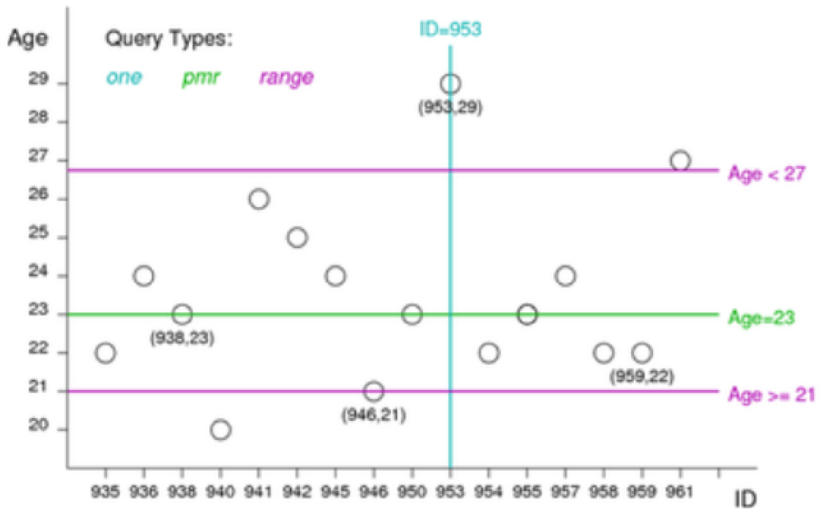
- 1-d
- n-d
- similarity (i don't know, it means approximate matching, with ranking)

each style has several possible file-structure/techniques

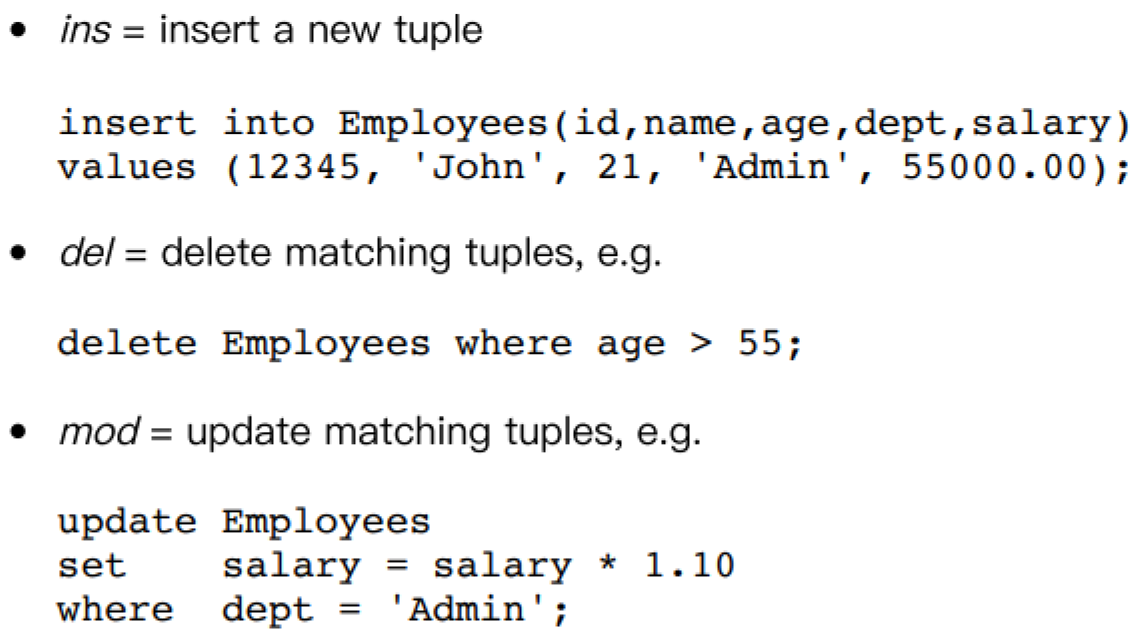
2. One-dimensional selection

there are three kinds of 1-d select queries:

- one
- pmr
- range



other operation on relations:



- *ins* = insert a new tuple

```
insert into Employees(id,name,age,dept,salary)
values (12345, 'John', 21, 'Admin', 55000.00);
```

- *del* = delete matching tuples, e.g.

```
delete Employees where age > 55;
```

- *mod* = update matching tuples, e.g.

```
update Employees
set    salary = salary * 1.10
where  dept = 'Admin';
```

2.1 implementing select efficiently

there are two basic approaches to make select operation more efficiently:

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

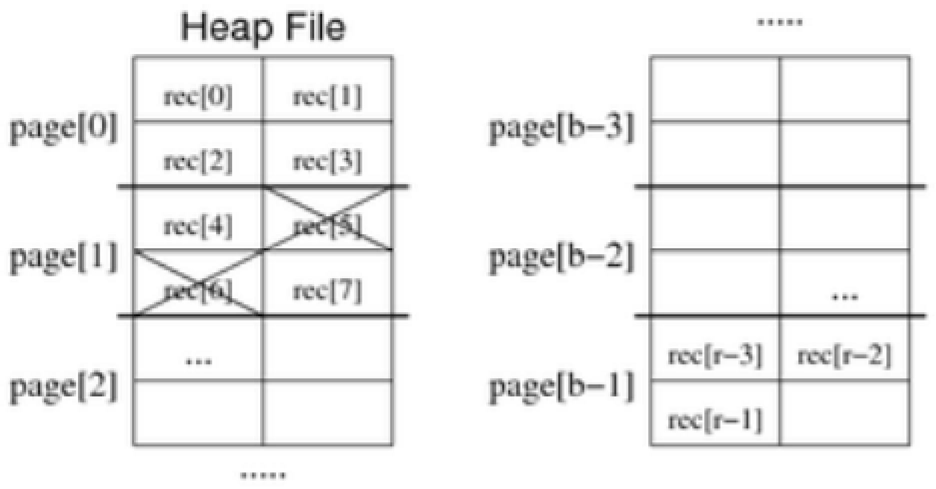
and our analyse base on an assumption: **1 input buffer for each relation**

cause if more buffer avaliable, most methods benefit

3. Heap file

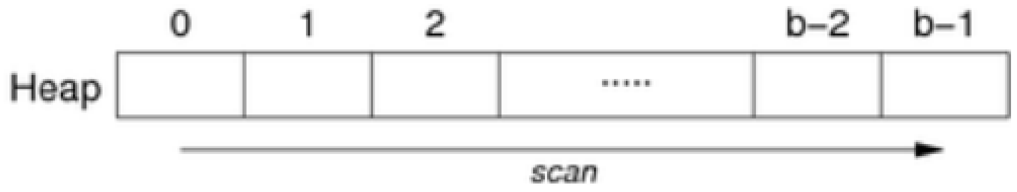
notes: “heap files” is simply an unordered collection of tuples in file

new tuples inserted at end of file, tuples deleted by marking



3.1 selection in heap file

for all selection queries, the only possible strategy is linear scan



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

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*

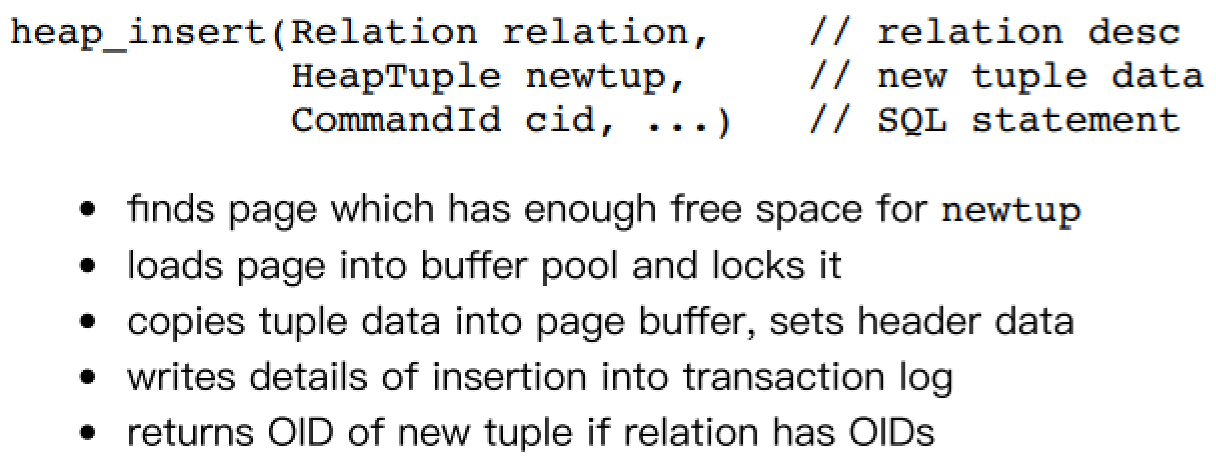
3.2 insertion in heaps

the cost of insertion = 1_r + 1_w (by scan)

and the strategy is :

- find any page in relation with enough space
- preferably a page already loaded into mem buffer

and the PostgreSQL's tuple insertion is shown below:

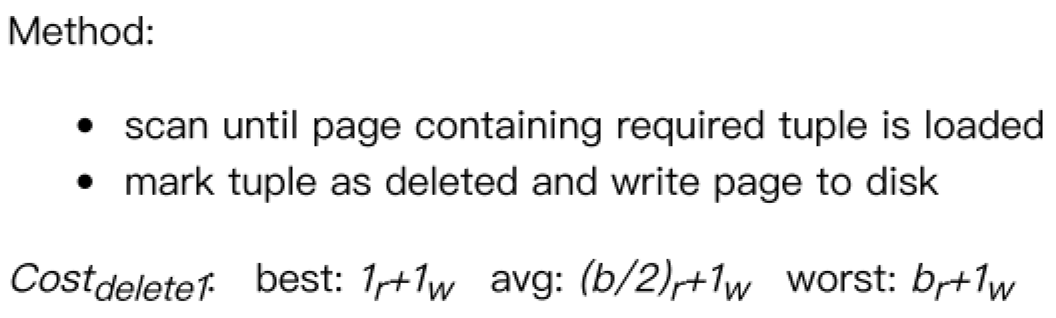


```
heap_insert(Relation relation,     // relation desc
            HeapTuple newtup,      // new tuple data
            CommandId cid, ...)    // SQL statement
```

- finds page which has enough free space for `newtup`
- loads page into buffer pool and locks it
- copies tuple data into page buffer, sets header data
- writes details of insertion into transaction log
- returns OID of new tuple if relation has OIDs

3.3 deletion in heaps

for one condition:

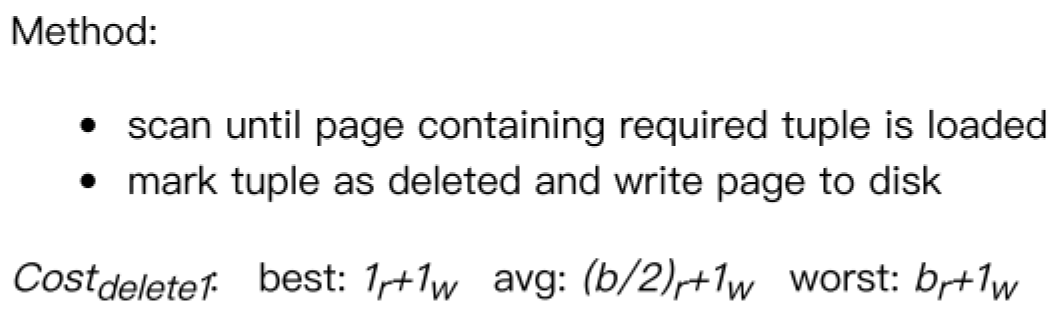


Method:

- scan until page containing required tuple is loaded
- mark tuple as deleted and write page to disk

$Cost_{delete1}$: best: 1_r+1_w avg: $(b/2)_r+1_w$ worst: b_r+1_w

for pmr or range condition:


Method:

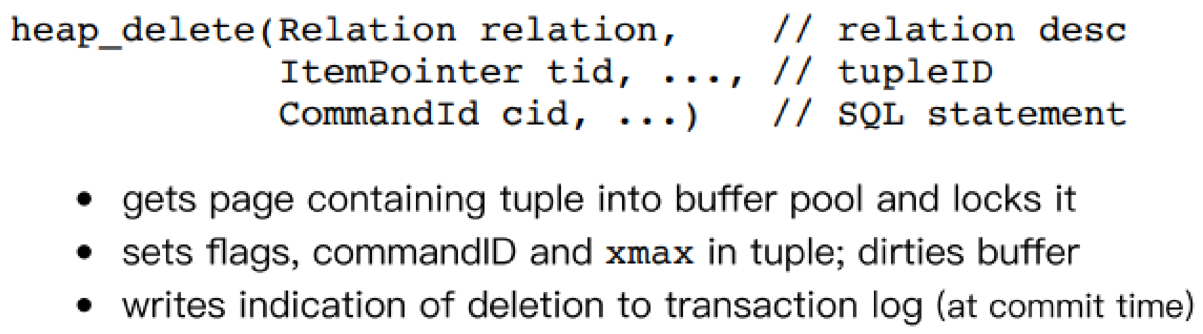
- scan until page containing required tuple is loaded
- mark tuple as deleted and write page to disk

$Cost_{delete1}$: best: 1_r+1_w avg: $(b/2)_r+1_w$ worst: b_r+1_w

and after deletion, free slots will apper, and how should we deal with them?

- re-use on subsequent insertion (requires modified insertion)
- periodic file reorganisation (requires full table locking for extended time)

deletion in postgreSQL:


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

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

3.4 updates in heaps

queries like:

```
update R set F=val where Condition;
```

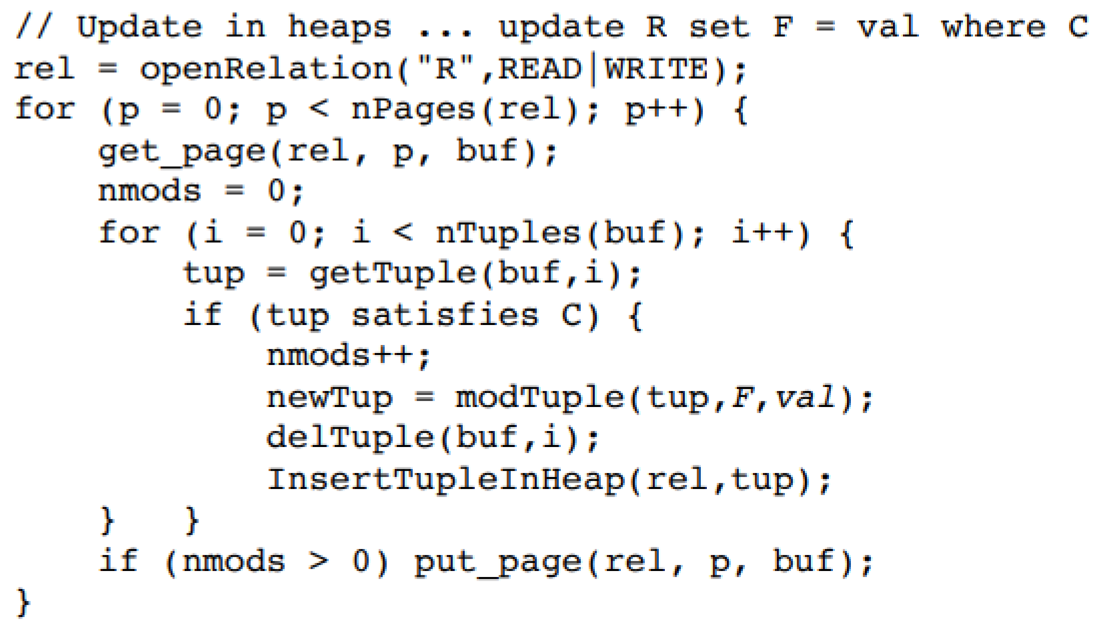
analysis for update is similar to that for deletion

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

Cost of update = b_r + b_qw

but updation may bring some complication like: new version of tuples may not fit in page

solution is delete this tuple and then insert it



```
// Update in heaps ... update R set F = val where C
rel = openRelation("R",READ|WRITE);
for (p = 0; p < nPages(rel); p++) {
    get_page(rel, p, buf);
    nmods = 0;
    for (i = 0; i < nTuples(buf); i++) {
        tup = getTuple(buf,i);
        if (tup satisfies C) {
            nmods++;
            newTup = modTuple(tup,F,val);
            delTuple(buf,i);
            InsertTupleInHeap(rel,tup);
    }   }
    if (nmods > 0) put_page(rel, p, buf);
}
```

3.5 heap files in PostgreSQL

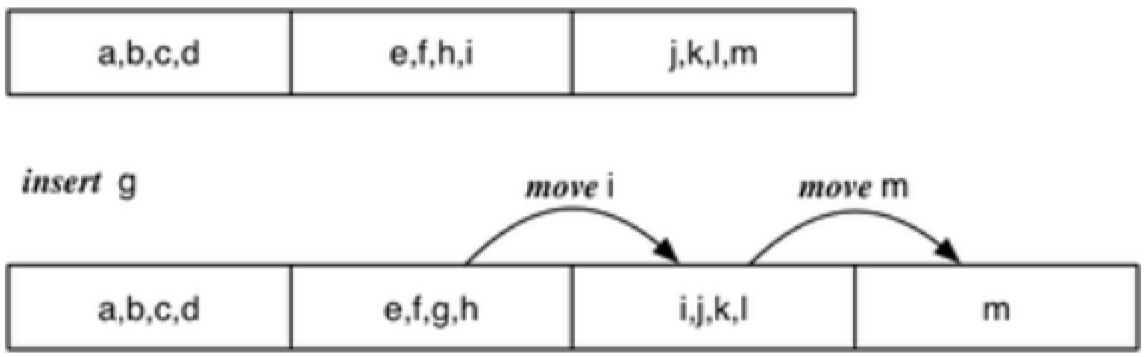
PostgreSQL store all table data in heap files (by default)

and a “heap files” use ≥ 3 physical files:

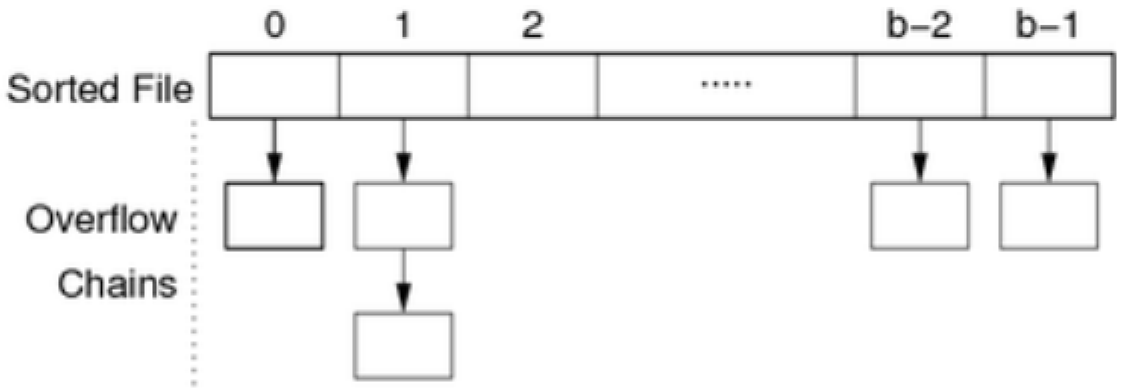
- first data file is called simply OID
  - if size exceeds 1GB, create a fork called OID.1
  - add more fork as data size grows (1 fork for 1 GB)
- second data file is free space map (OID_fsm)
- the thrid one is visibility map (OID_vm)
- and, optionally, TOAST file (if table has varlen attributes)

4. Sorted Files

sorted files make searching more efficient, but make insertion less efficient



so we design a sorted file-structure with overflow chains:(conceptual view is shown below)



Total number of overflow blocks = b_{OV}.

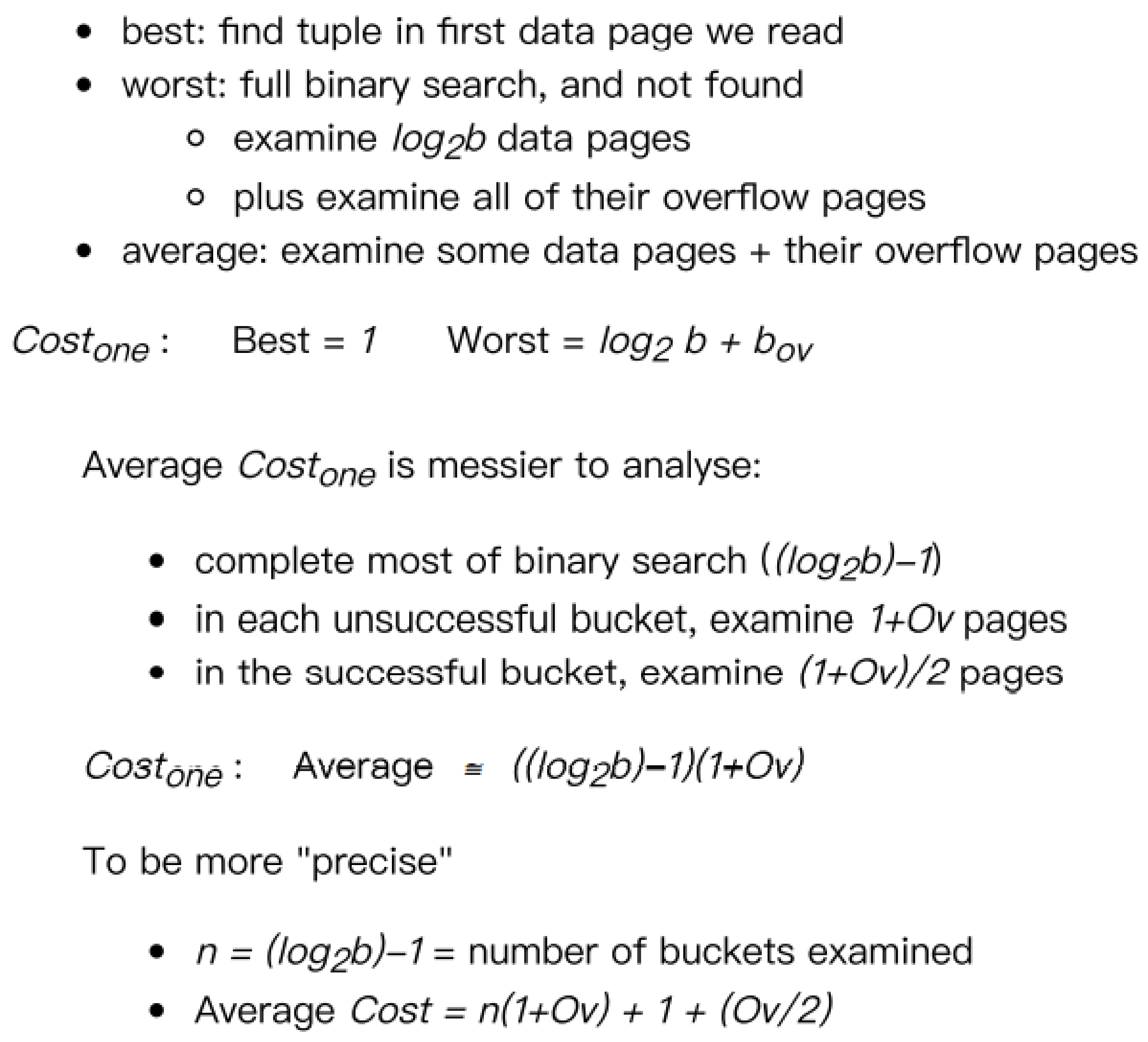
Average overflow chain length = $Ov = b_{OV}/b$.

Bucket = data page + its overflow page(s)

4.1 selection in sorted file

- for one queries on sort key, use binary search;

we treats each bucket as a single large page



- best: find tuple in first data page we read
- worst: full binary search, and not found
  - examine log_2b 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 $Cost_{one}$ is messier to analyse:

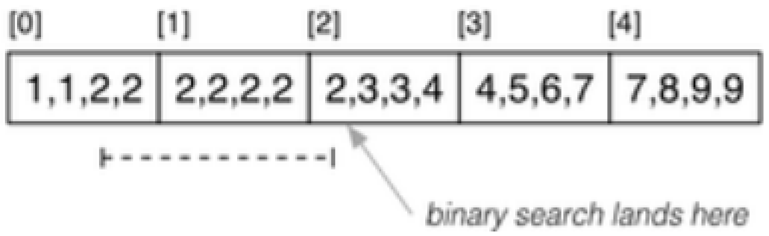
- complete most of binary search ($(log_2b)-1$)
- in each unsuccessful bucket, examine $1+Ov$ pages
- in the successful bucket, examine $(1+Ov)/2$ pages

$Cost_{one}$: Average $\cong$ $((log_2b)-1)(1+Ov)$

To be more "precise"

- $n = (log_2b)-1$ = number of buckets examined
- Average $Cost = n(1+Ov) + 1 + (Ov/2)$

- for pmr queries, on non-unique attribute k
  - assume file is sorted on k
  - tuples contain k may appear in several pages



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

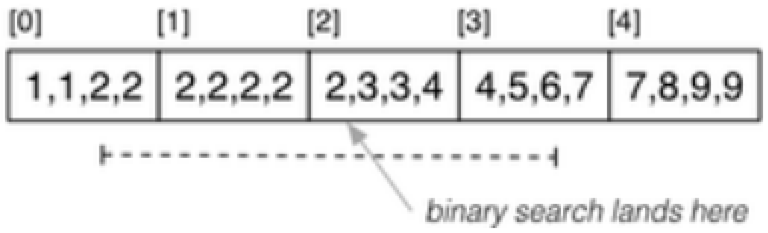
Scan backwards and forwards from *p* to find matches.

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

- for range queries on unique sort key, like primary key
  - use binary search to find the lower bound
  - read sequentially util reach the upper bound

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

If secondary key, similar method to *pmr*.



so far, we have dicuss condition involves sort key k, so what will happen if the attribute we select is not the sort key?

the ans is: it just like selection in heap file

4.2 insertion in sorted files

Insertion is more complex than for Heaps.

- 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 $\delta = 1 \text{ or } 2$, depending on whether we need to create a new overflow block

4.3 deletion in sorted files

Deletion involves:

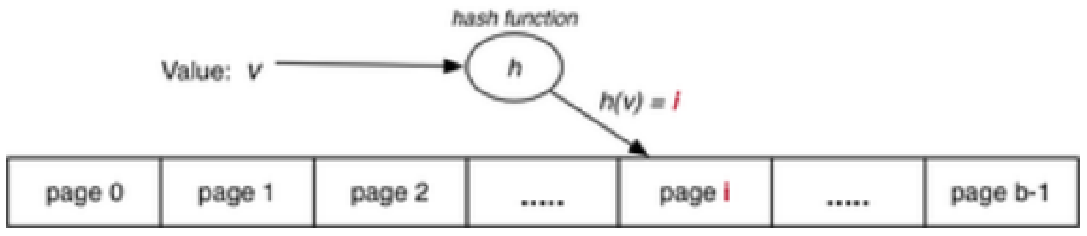
- find matching tuple(s)
- mark them as deleted

Cost depends on selection criteria (i.e. on how many matching tuples there are)

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

5. Hashed Files

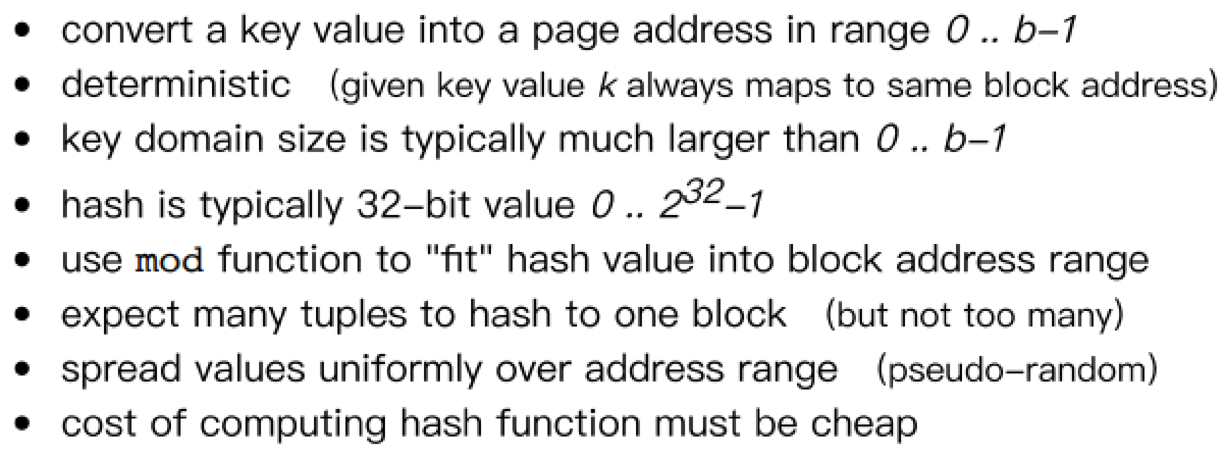
basic idea of hashing is: use key value to compute page address of tuple



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

Requires: hash function $h(v)$ that maps $KeyDomain \rightarrow [0..b-1]$.

there are several points to note:



- convert a key value into a page address in range *0 .. b−1*
- deterministic (given key value *k* always maps to same block address)
- key domain size is typically much larger than *0 .. b−1*
- hash is typically 32-bit value $0 .. 2^{32}-1$
- use `mod` function to "fit" hash value into block address range
- expect many tuples to hash to one block (but not too many)
- spread values uniformly over address range (pseudo-random)
- cost of computing hash function must be cheap

5.1 hash function

usual approach in hash function:

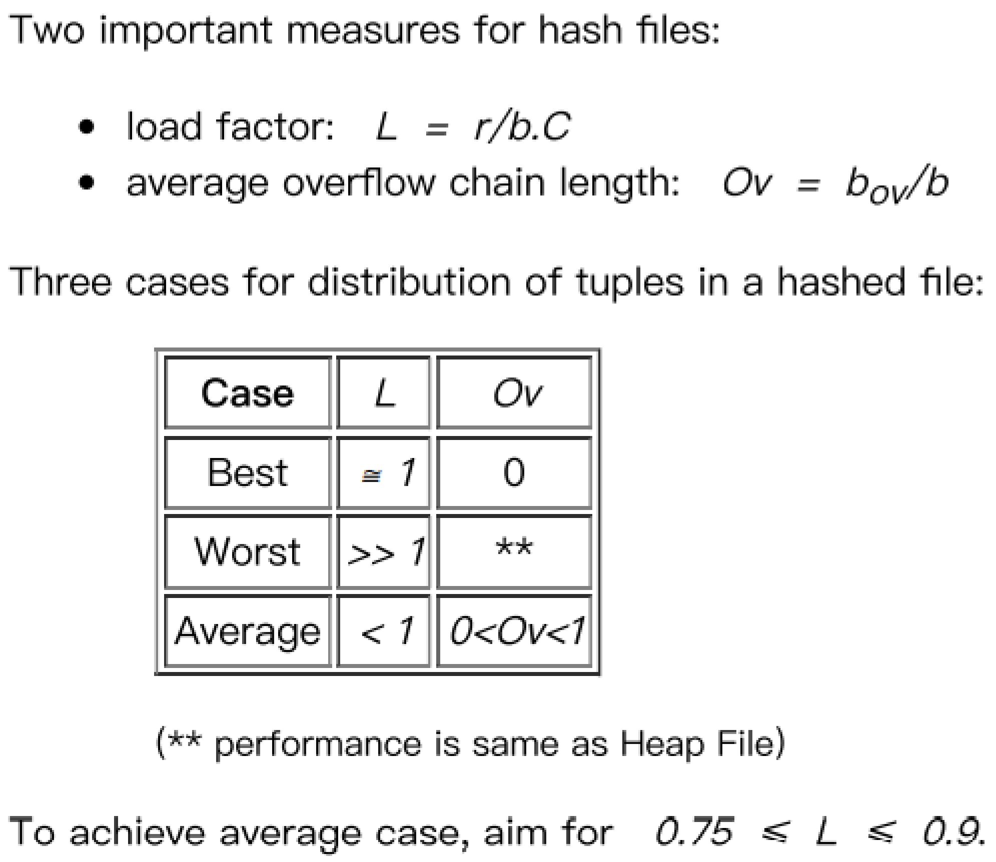
- convert other types of value into numeric value
- numeric value can be viewed as arbitrary-length bit-string
- and bit-string can be mapped into page address

5.2 hashing performance

the performance of hash function depends on the data distribution, since if data distribution not uniform, block address cannot be uniform

if the data is non-uniform or too many, we can add a overflow blocks

and there are two important measures for hash files



Two important measures for hash files:

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

Three cases for distribution of tuples in a hashed file:

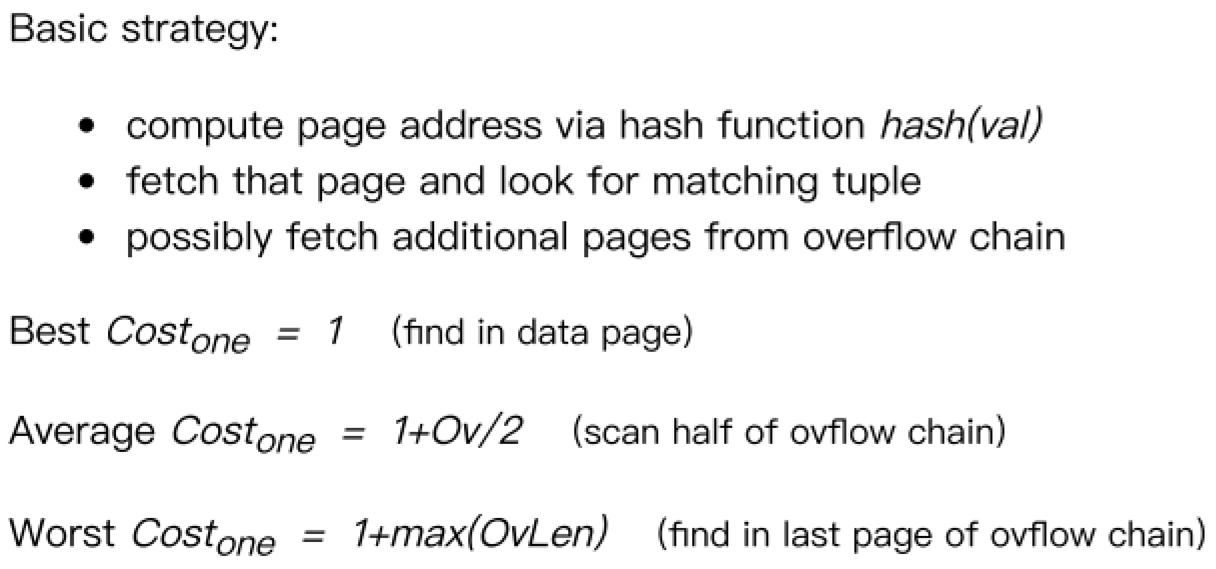
Case	*L*	*Ov*
Best	$\cong 1$	0
Worst	$>> 1$	**
Average	< 1	$0<Ov<1$

(** performance is same as Heap File)

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

5.3 selection with hashing

- for one queries, hashing has the best performance



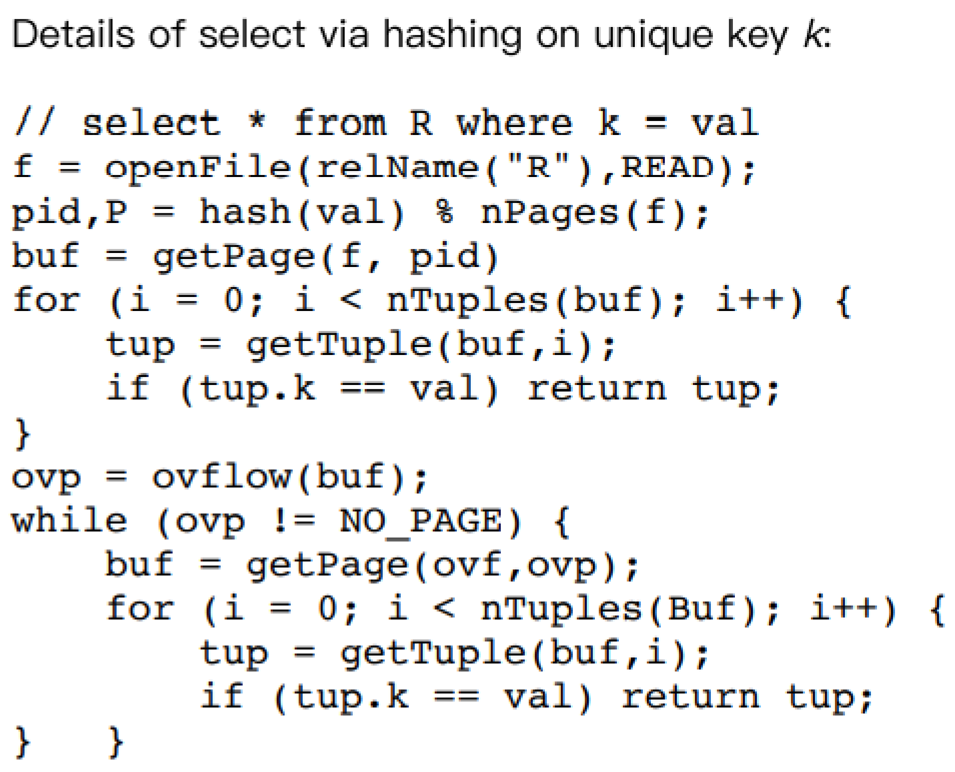
Basic strategy:

- 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)


Details of select via hashing on unique key *k*:

```
// select * from R where k = val
f = openFile(relName("R"),READ);
pid,P = hash(val) % nPages(f);
buf = getPage(f, pid)
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;
}   }
```

- for non-unique hash key nk (pmr)

operation is similar to one queries, scanning each tuple in Page(address calculated by hash function), and then scanning each tuple in overflow blocks

```
// select * from R where nk = val
pid,P = hash(val) % nPages(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
```

$Cost_{pmr} = 1 + Ov$

- for range queries

(unless the hash functionis order-preserving, hashing doesn’t help with range queries)

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

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

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

5.4 insertion with hashing

insertion with hashing easy, first find the corresponding page, and then iterating all space in page and in overflow blocks, if have space then insert in it, or create a new overflow block

Variation in costs determined by iteration on overflow chain.

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

Average $Cost_{insert} = (1+Ov)_r + 1_w$

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

5.5 deletion with hashing

operation of deletion is similar to selection, extra cost over selection is cost of writing back modified blocks, methods works for both unique and non-unique hahs keys

5.6 flexible hashing

recently what we talk about is based on static file-size, for dynamic file, everytime size of file changed, we need to use a new hashing function, and requires complete re-organisation of the file

so there are several hashing schemes have been proposed to deal with thisi problem:

- two methods access blocks via directory
    - extendible hahsing, dynamic hashing
- expands files systematicaly, so needs no directory
    - linear hashing

and for all methods, all treat hash value as bit-string

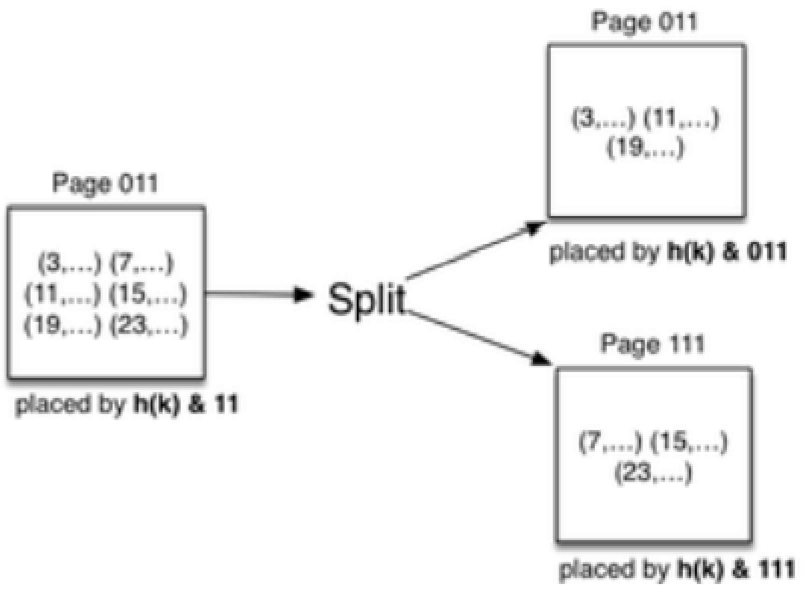
5.6.1 splitting

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`

Aim: expandable data file, requiring minimal large reorganisation

Example of splitting:

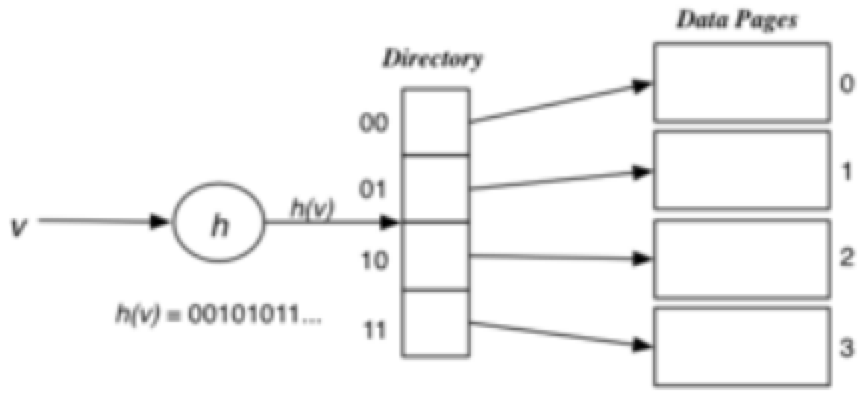


Tuples only show key value; assume *h(val) = val*

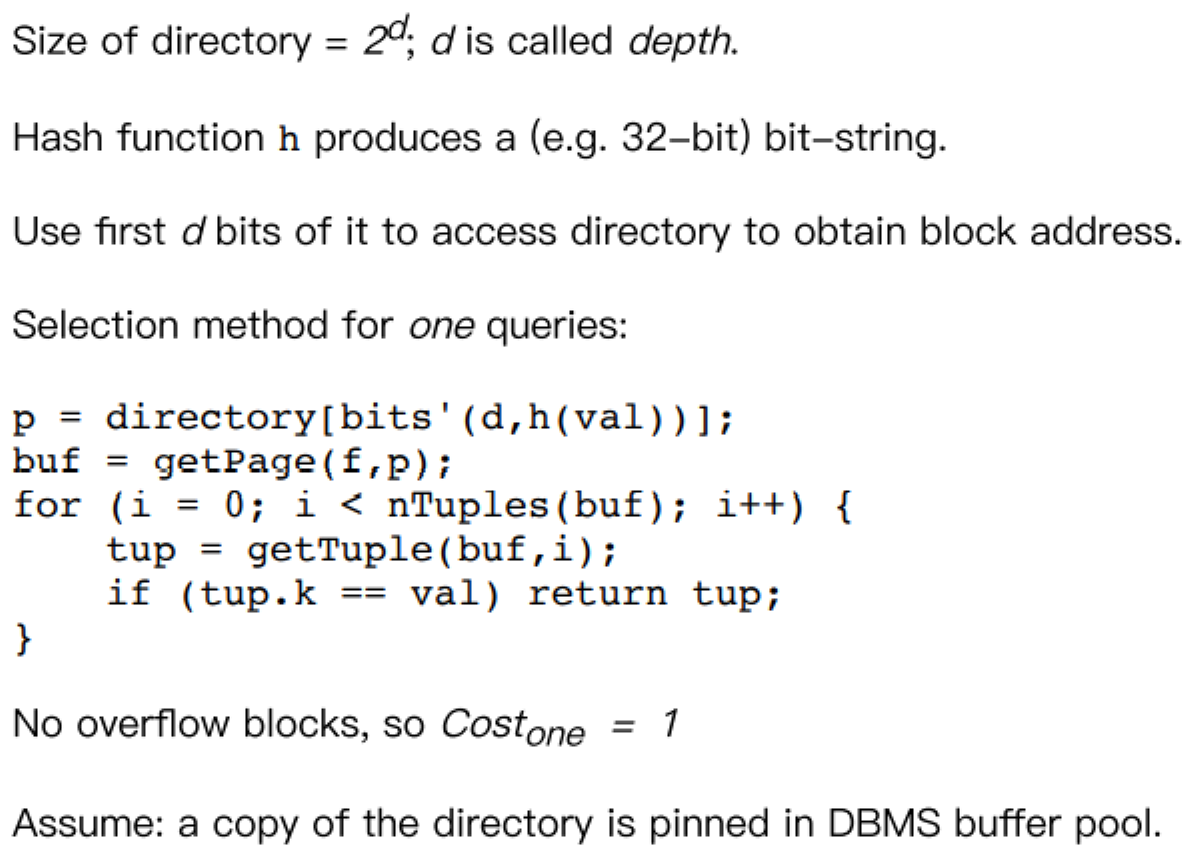
5.6.2 extendible hashing

- file of primary data blocks
- directory containing block addresses
- directory indexed using first/last *d* bits from hash value
  (can make use of bits either high→low or low→high bits
- data file and directory expand as more tuples added
  (expansion occurs whenever pages fill, so there are **no overflow pages**)

Directory could live in page 0 and be cached in memory buffer.



5.6.2.1 selection with Ext.Hashing



Size of directory = 2^d; *d* is called *depth*.

Hash function `h` produces a (e.g. 32-bit) bit-string.

Use first *d* bits of it to access directory to obtain block address.

Selection method for *one* queries:

```
p = directory[bits'(d,h(val))];
buf = getPage(f,p);
for (i = 0; i < nTuples(buf); i++) {
    tup = getTuple(buf,i);
    if (tup.k == val) return tup;
}
```

No overflow blocks, so $Cost_{one} = 1$

Assume: a copy of the directory is pinned in DBMS buffer pool.

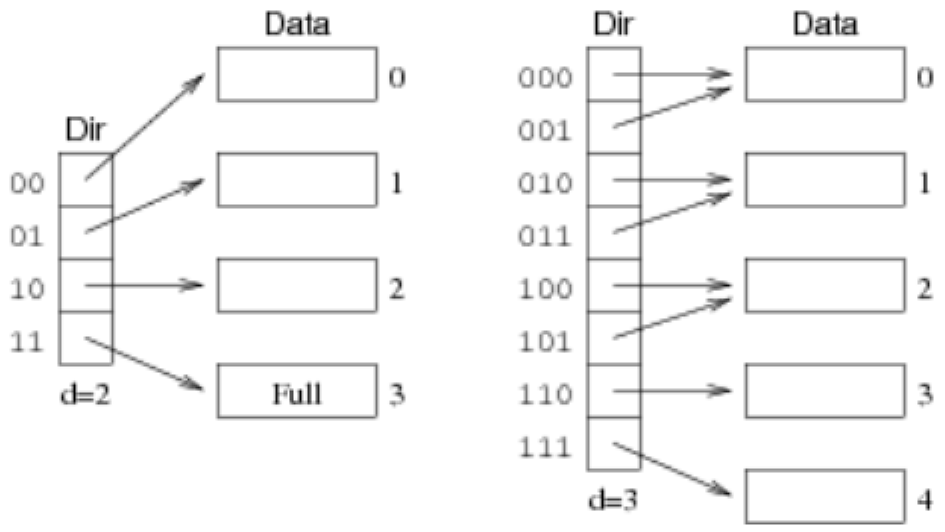
5.6.2.2 insertion with Ext.Hashing

you know we have a directory, so what can we do if a new tuple arrive(1xxxxx…), but directory 1 is full?

we can use method like splitting

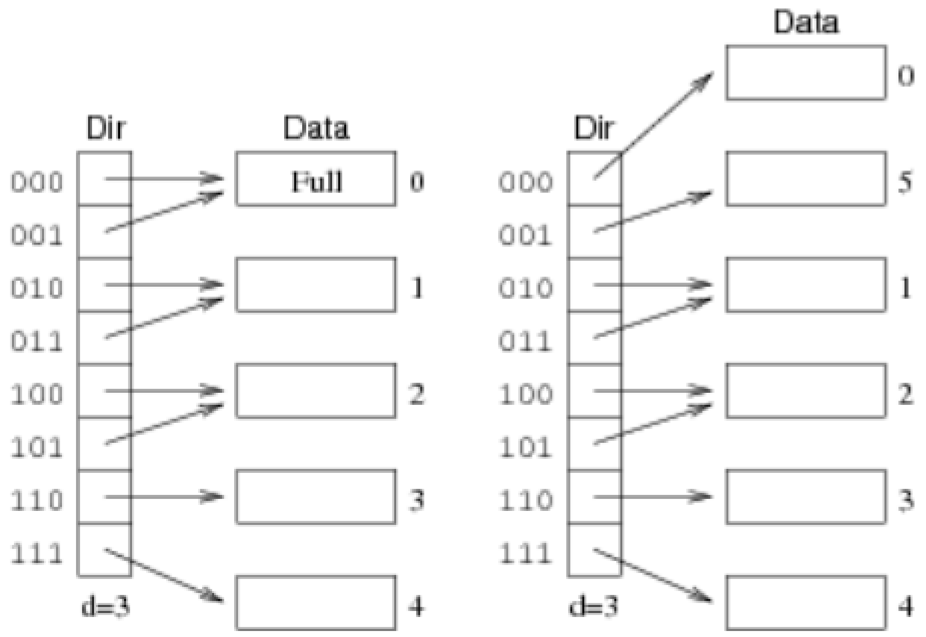
doubling directory size and adding one block creates many empty directory slots

and we can make them point to existing pages



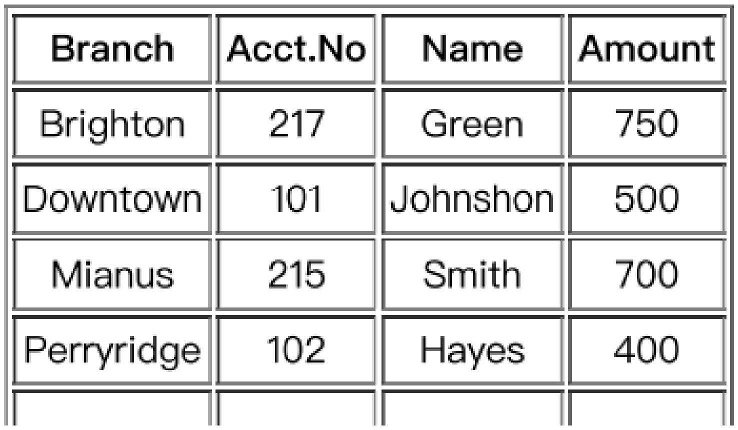
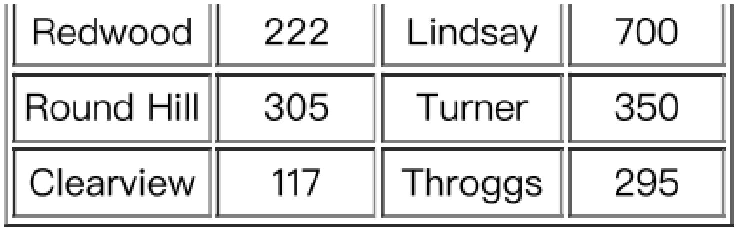
the idea: some parts are hashed effectively using d bits, others using d+1 bits

and if we split a block with two pointer to it, no need to make directory larger



let’s explain this with an example:

Consider the following set of tuples describing bank deposits:

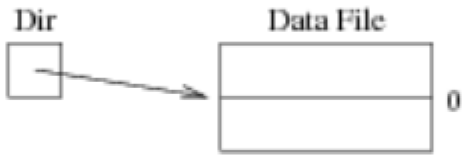
Branch	Acct.No	Name	Amount
Brighton	217	Green	750
Downtown	101	Johnshon	500
Mianus	215	Smith	700
Perryridge	102	Hayes	400
Redwood	222	Lindsay	700
Round Hill	305	Turner	350
Clearview	117	Throggs	295

We hash on the branch name, with the following hash function:

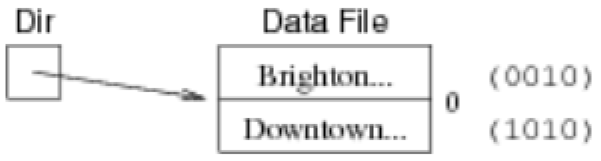
Branch	Hash Value
Brighton	`0010 1101 1111 1011`
Clearview	`1101 0101 1101 1110`
Downtown	`1010 0011 1010 0000`
Mianus	`1000 0111 1110 1101`
Perryridge	`1111 0001 0010 0100`
Redwood	`1011 0101 1010 0110`
Round Hill	`0101 1000 0011 1111`

Assume we have a file with *c=2*.

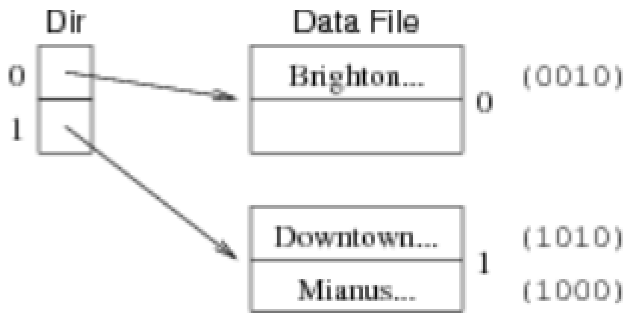
Start with an initially empty file:



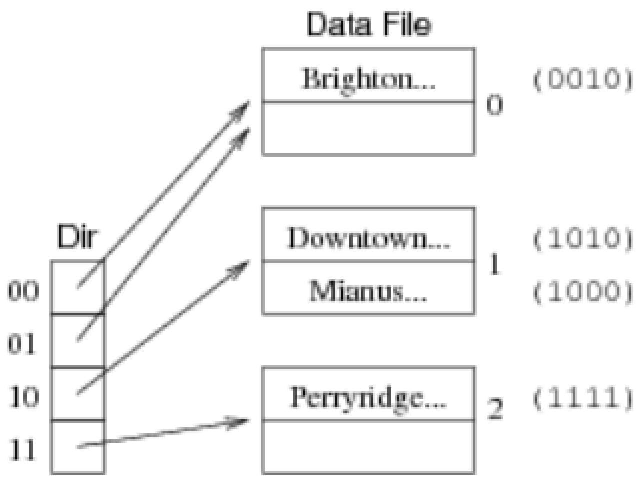
Add `Brighton...` tuple (hash=`0010...`).
Add `Downtown...` tuple (hash=`1010...`).



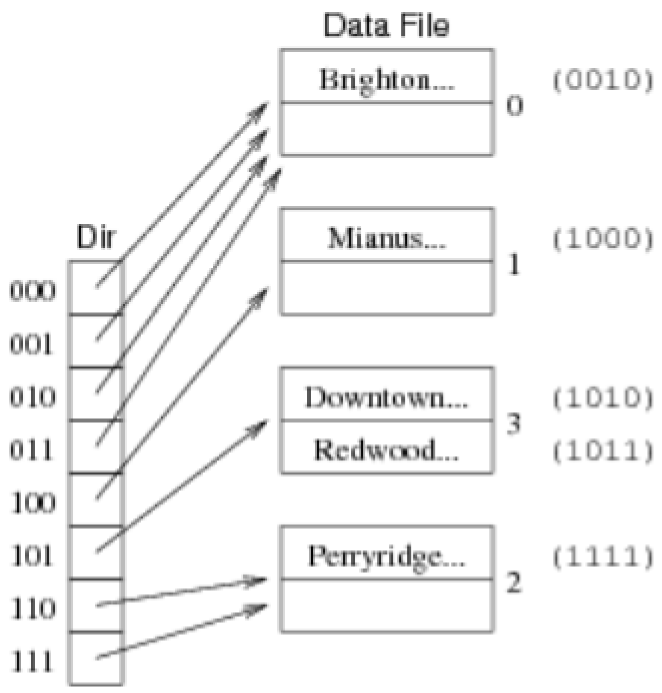
Add `Mianus...` tuple (hash=`1000...`).



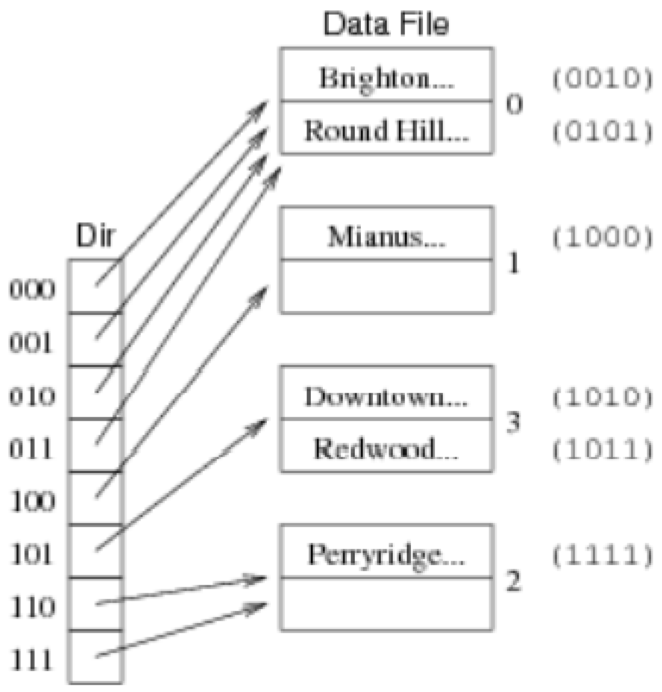
Add `Perryridge...` tuple (hash=`1111...`).



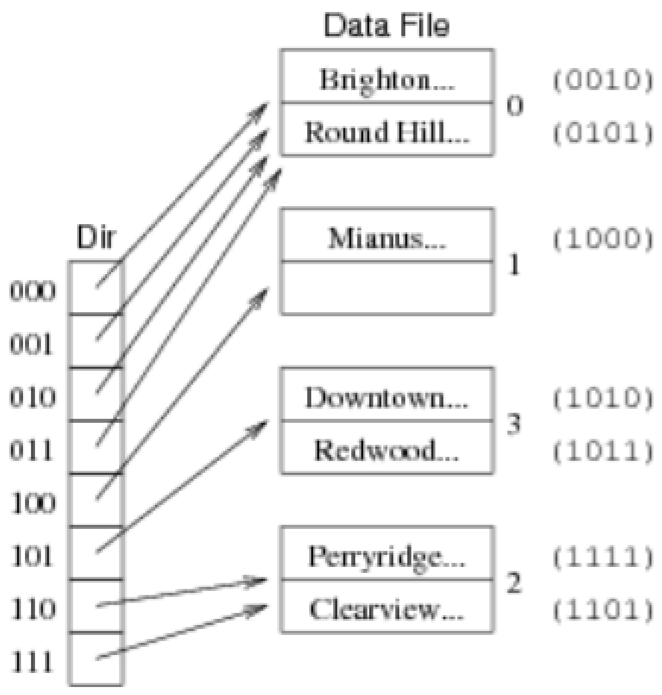
Add `Redwood...` tuple (hash=`1011...`).



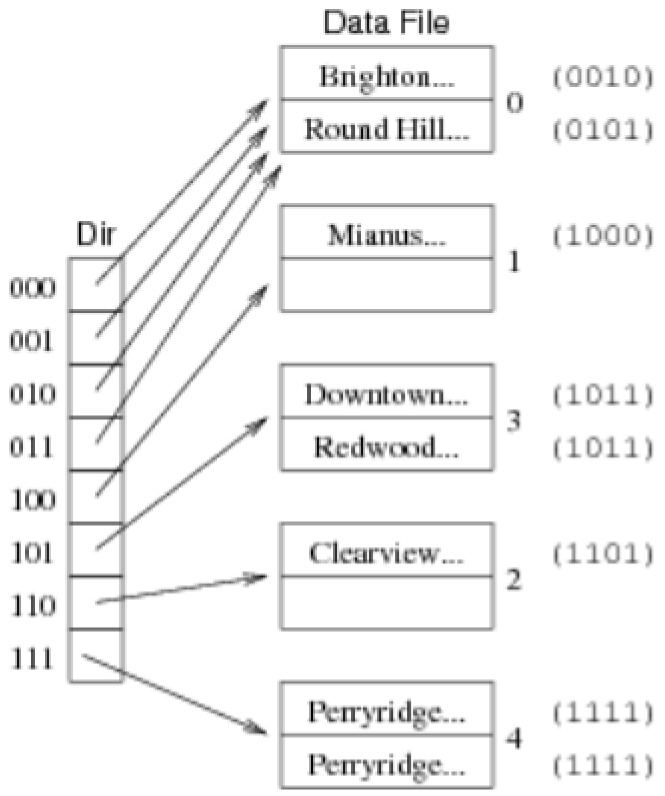
Add `Round Hill...` tuple (hash=`0101...`).



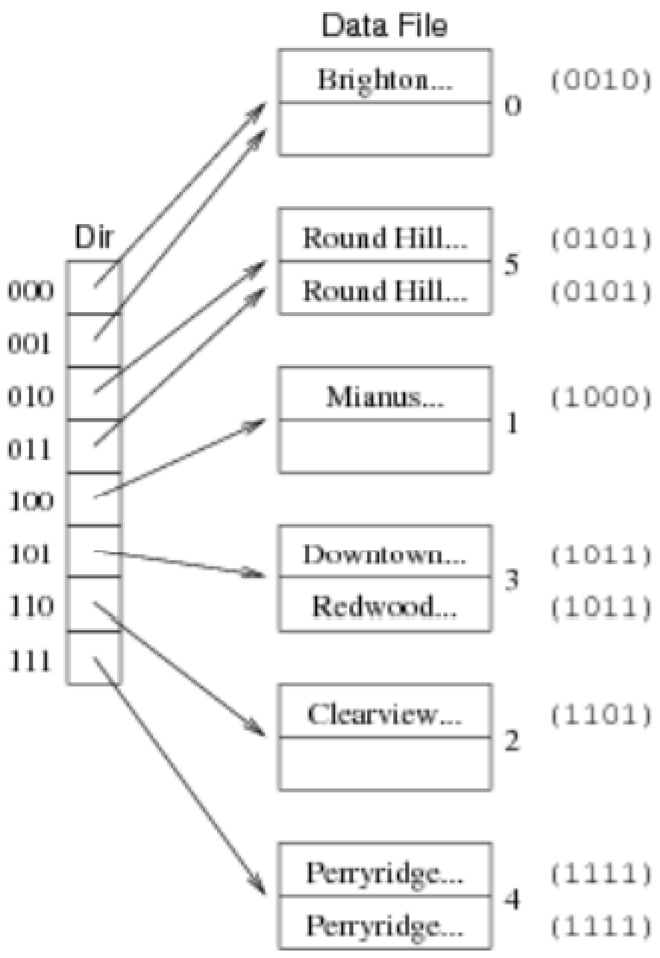
Add `Clearview...` tuple (hash=`1101...`).



Add another `Perryridge...` tuple (hash=`1111...`).



Add another `Round Hill...` tuple (hash=`0101...`).



the demostration above need asumption: directory is small enough to be stored in memory

On average, $Cost_{insert} = 1_r + (1+\delta)_w$

(δ is a small value depending on b, load, hash function,...)

but this methods will bring some potential problem:

- split might leave all tuple in a page(p1)
- new tuple hashes to page(p1) → no room

- need to split again until eventually differ in d+n bits

and there are a degenerate case:

- each block holds c tuples
- more than c tuples hash to exactly same value
- splitting never disabiguates tuples → directory explodes

5.6.2.3 deletion with Ext.Hashing

deletion with Ext.Hashing is similar to ordinary static hash file, that mark tuples as removed

but you know empty block might create hole in the middle of file

there are two methods to solve this problem:

- maintain a list of free pages, and re-use on next split
- shrink file: copy last page into “hole”, adjust directory

5.6.3 Linear Hashing

file organisation:

- file of primary data blocks
- file of overflow data blocks
- a register called the split pointer

advantage over other approaches:

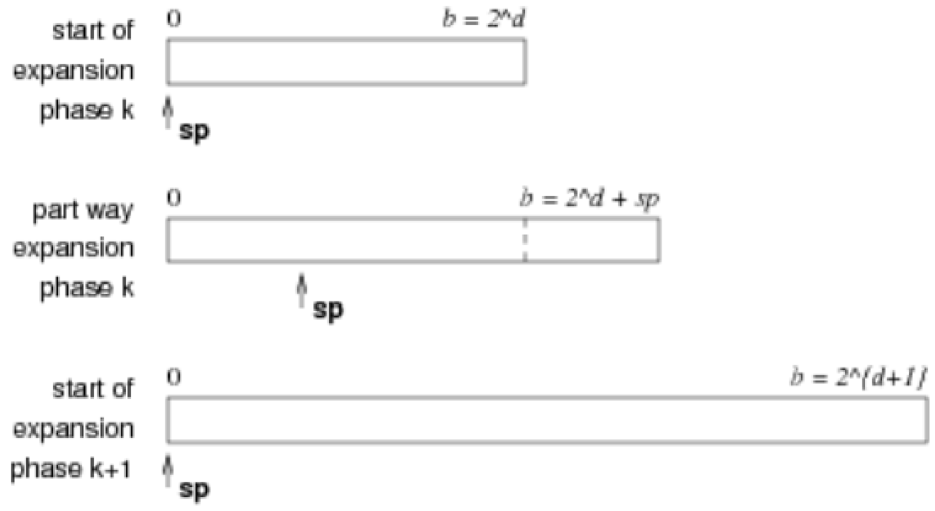
- does not require auxiliary storage for a directory
- dose not require periodic file reoragnisation

use systematic methods of growing data file:

- hash function “adapts” to changing address range
- systematic splitting controls the length of overflow chains

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

Can be viewed as having "phases" of expansion; during each phase, *b* doubles.



start of expansion phase k
0
b = 2^d
sp
part way expansion phase k
0
b = 2^d + sp
sp
start of expansion phase k+1
0
b = 2^{d+1}
sp

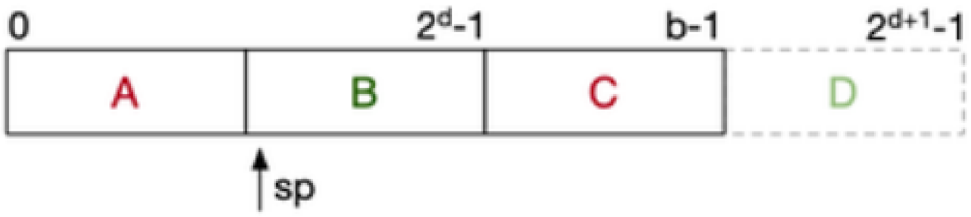
5.6.3.1 selection with linear Hashing

- if b=2^d, the file behaves exactly like standard hashing
    - use d bits of hash to compute block address

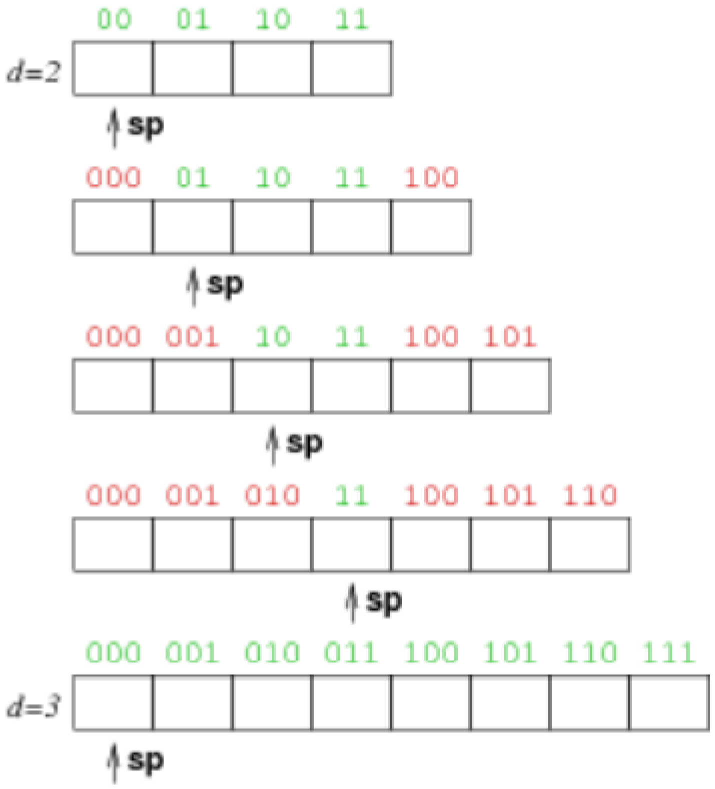
```
// select * from R where k = val
p = bits(d,hash(val));  --least sig bits
P = getPage(f,p)
for each tuple t in page P
         and its overflow pages {
    if (t.k == val) return t;
}
```

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

- if b≠2^d, treat different parts of the file differently



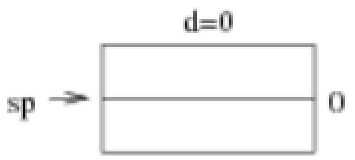
a illustration is shown below:



here is an example:

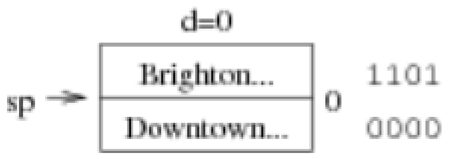
Assume we have a file with: *c=2*.

Start with an initially empty file:

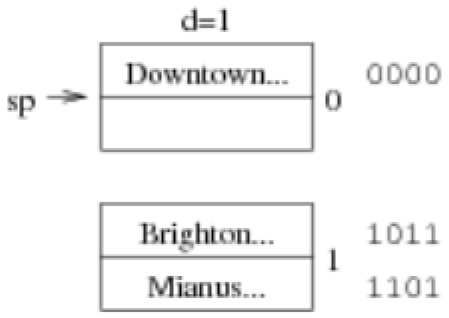


Add `Brighton...` tuple (hash=`...1011`).

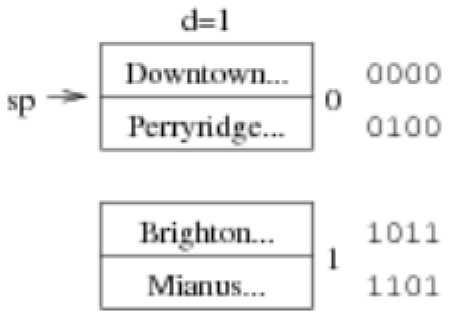
Add `Downtown...` tuple (hash=`...0000`).



Add `Mianus...` tuple (hash=`...1101`).



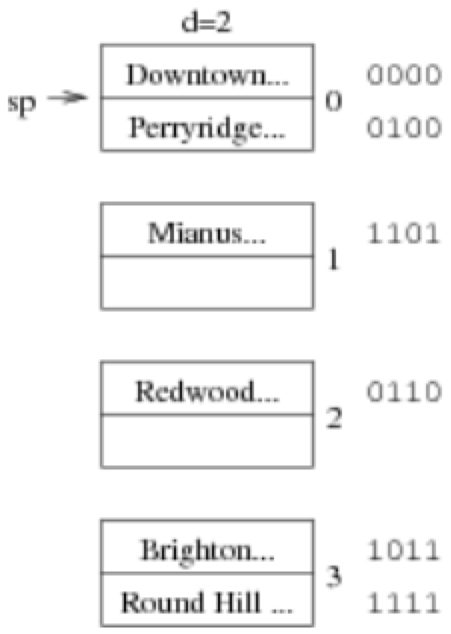
Add `Perryridge...` tuple (hash=`...0100`).



Add `Redwood...` tuple (hash=`...0110`).

... Lin.Hashing Example

Add `Round Hill...` tuple (hash=`...1111`).



5.6.3.2 splitting

how to decide that we “need to split”

- in extendible hashing, blocks were split on overflow
- in linear hashing, we always split block sp

and there are two approaches to triggering a split:

- split everytime a tuples was inserted into a full block
- split when load factor reaches threshhold

6. indexes

an indexs is a file of (keyVal, tupleID) pairs

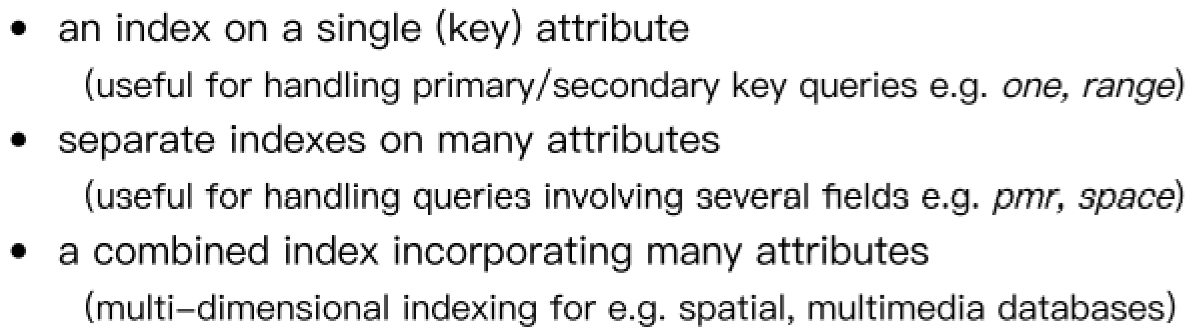
Taxonomy of index types, based on properties of index attribute:

- primary: index on unique field, file sorted on this filed
- clustering: index on non-unique fields, file sorted on this field
- secondary: file not sorted on this fields

and there are seveal index structures, which aims to minimise storage and search cost

- dense
- sparse
- single-level
- multi-level

a relation/file may have: (indexs are typically stored in separate files to data)



- an index on a single (key) attribute
  (useful for handling primary/secondary key queries e.g. *one, range*)
- separate indexes on many attributes
  (useful for handling queries involving several fields e.g. *pmr, space*)
- a combined index incorporating many attributes
  (multi-dimensional indexing for e.g. spatial, multimedia databases)

6.1 primary index

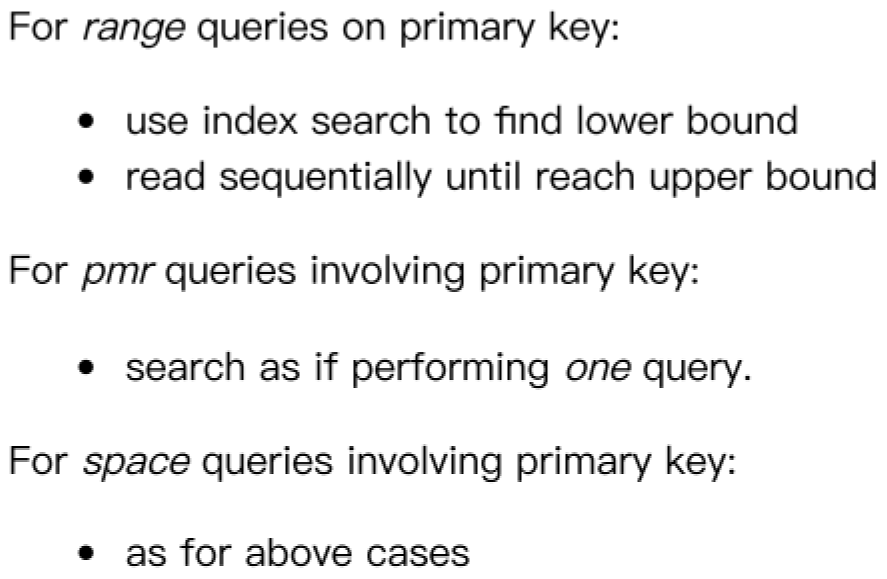
primary index is an ordered file whose elements have two fields

- key value: for primary key attribute
- tuple id: for tuple containing key value

there are two condition:

- Dense: one index tuple per data tuple (no requirement on data file ordering)
- Sparse: one index tuple per data page (requires primary key sorted data file)

6.1.1 selection with prim.index


For *range* queries on primary key:

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

For *pmr* queries involving primary key:

- search as if performing *one* query.

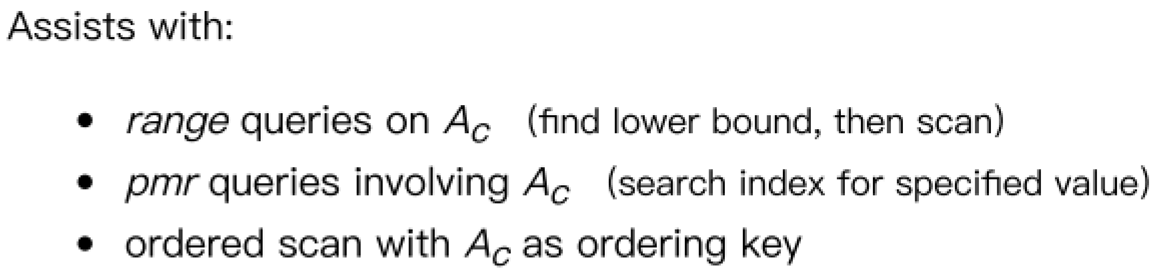
For *space* queries involving primary key:

- as for above cases

6.2 clustering index

index on non-unique ordering attribute

usually a sparse index, one pointer to first tuple containing value


Assists with:

- *range* queries on A_C (find lower bound, then scan)
- *pmr* queries involving A_C (search index for specified value)
- ordered scan with A_C as ordering key

Insertions are expensive: rearrange index file and data file.
This applies whether new value or new tuple containing known value.
One "optimisation": reserve whole block for one particular value.
Deletions relatively cheap (similar to primary index).
(Note: can't mark index entry for value X until all X tuples are deleted)

6.3 secondary index

generally, dense index on non-unique attribute

6.4 multi-level indexes

in indexes described so far, search begins with binary search to index all time

and it reuqires log2 i index block accesses

we can improve it by putting a second index on the first index

7. B Tree Index

B tree are MST with such properties:

- they are updated so as to remain balanced
- each node hash at least (n-1)/2 entries in it
- each tree node occupies an entire disk page

B tree insertion and deletion methods

- are moderately complicated to describe
- can be implemented very efficiently

advantage of B trees over general MST

- better storage utilisation
- better worst case performance

7.1 B+Tree

In database context, nodes are index blocks.

Two possible ways to structure them:

- B trees:
  - some entries in each node have (key, tid) pair
  - other entries (key,ixBlock) pairs (pointers within tree)
- B+ trees:
  - high-level entries contain (key, index block) pairs
  - first-level entries contain (key, data block) pairs

Higher levels of B+ tree have greater $c_i \quad \Rightarrow \quad$ B+ tree may have less levels.