Selection on One Attribute

Selection

Varieties of Selection

Selection is a relational algebra operation that ...

- filters a subset of tuples from one relation
- based on a condition 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

One-dimensional Selection

Operations for 1d Select

One-dimensional select queries = condition on single attribute.

- *one* = single tuple access, e.g.

  ```
  select * from Employees where id = 15567;
  ```

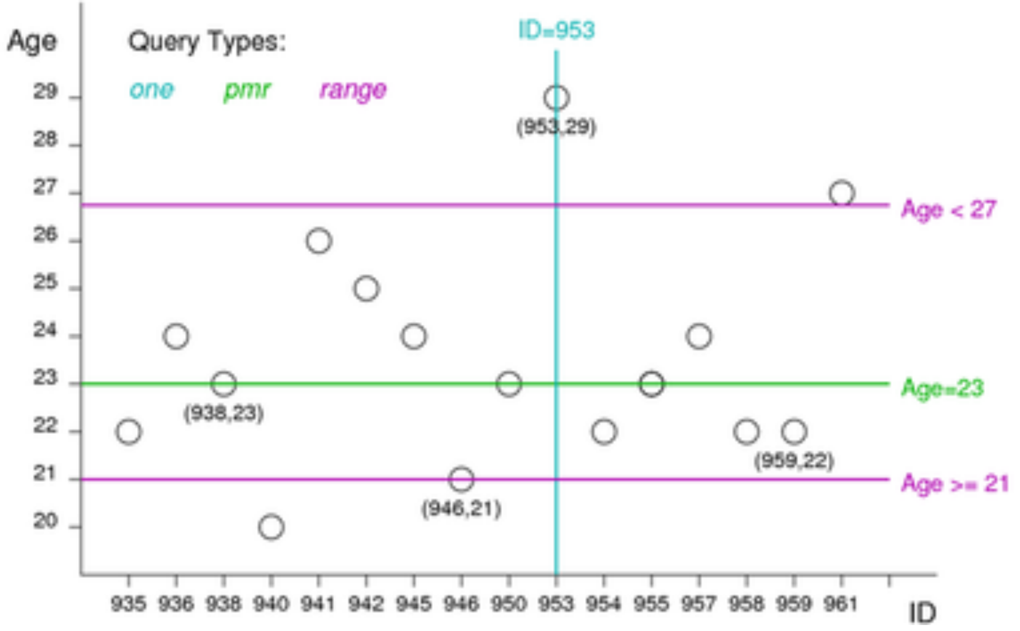
- *pmr* = multiple tuple access, e.g.

  ```
  select * from Employees where age = 25;
  ```

- *range* = range queries, e.g.

  ```
  select * from Employees where age>20 and age<50;
  ```

... Operations for 1d Select



... Operations for 1d Select

Other operations 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';
```

... Operations for 1d Select

In the algorithms below, we assume:

- *one* queries are of the form

```
select * from R where k = val
```

  where `k` is a `unique` attribute and `val` is a constant

- *pmr* queries are of the form

```
select * from R where k = val
```

  where `k` is a non-`unique` attribute and `val` is a constant

- *range* queries are of the form

```
select * from R where k between lo and hi
```

  where `k` is any attribute and `lo` and `hi` are constants

- `R.k` gives the value of attribute `k` in table `R`

Implementing Select Efficiently

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.

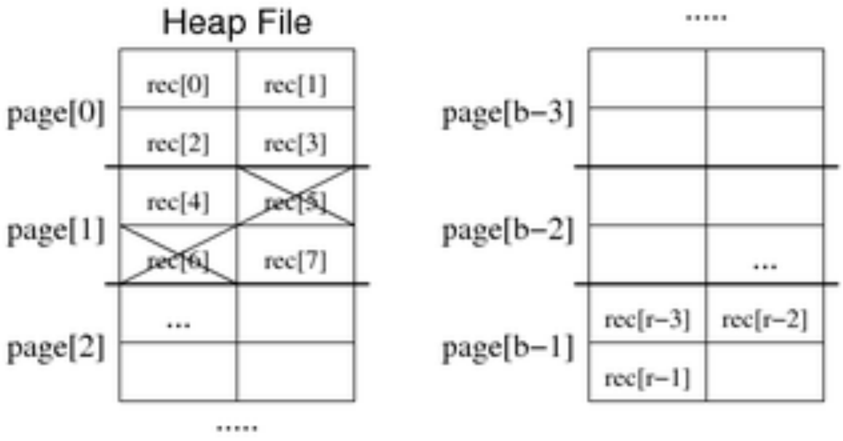
Heap Files

> Note: this is **not** "heap" as in the top-to-bottom ordered tree.
> It means simply an unordered collection of tuples in a file.

Heap File Structure

The simplest possible file organisation.

New tuples inserted at end of file; tuples deleted by marking.



Selection in Heaps

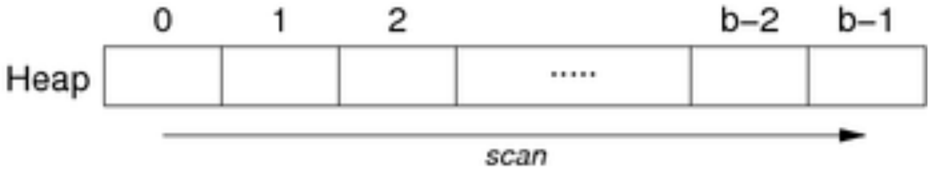
For all selection queries, the only possible strategy is:

```
// 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

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

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 recs)

... 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`
- 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

Deletion in Heaps

Deletion for *one* conditions, e.g.

```
delete from Employees where id = 12345
```

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

... Deletion in Heaps

Deletion for *pmr* or *range* conditions, e.g.

```
delete from Employees where dept = 'Marketing'
delete from Employees where 40 <= age and age < 50
```

Method:

- scan all pages
- mark matching tuples as deleted (e.g. update slot directory)
- write affected (modified) pages to disk

$Cost_{delete} = b_r + b_{qw}$ (where b_q is #pages with matches)

... Deletion in Heaps

Implementation of deletion:

```
// Deletion in heaps ... delete from R where C
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 C)
            { ndels++; delete_record(buf,i); }
    }
    if (ndels > 0) put_page(rel, p, buf);
    if (ndels > 0 && unique) break;
}
```

... Deletion in Heaps

How to deal with free slots:

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

Insertion with free slots:

```
rel = openRelation("R",READ|WRITE);
for (p = 0; p < nPages(f); p++) {
    get_page(rel, p, buf);
    if (hasSpace(buf,newTup))
        { insert_record(buf,newTup); break; }
}
if (p == nPages(f)) // not yet added
    { clear(buf); insert_record(buf,newTup); }
put_page(rel, p, buf);
```

... Deletion in Heaps

21/164

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 (at commit time)

Updates in Heaps

22/164

Example: `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

$Cost_{update} = b_r + b_{qw}$

A complication: new version of tuple may not fit in page.

Solution: delete and then insert
(Cost harder to analyse; depends on how many "oversize" tuples are produced)

... Updates in Heaps

23/164

```
// 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);
}
```

... Updates in Heaps

24/164

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

25/164

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 files" use ≥ 3 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 68 in PostgreSQL v11 documentation

... Heaps in PostgreSQL

Free space map

- contains data on free-space-per-data page
- organised as a tree of free-space values (one byte per page)
- fast to discover: no page has space for new tuple
- fast to discover: page which has enough space

Visibility map

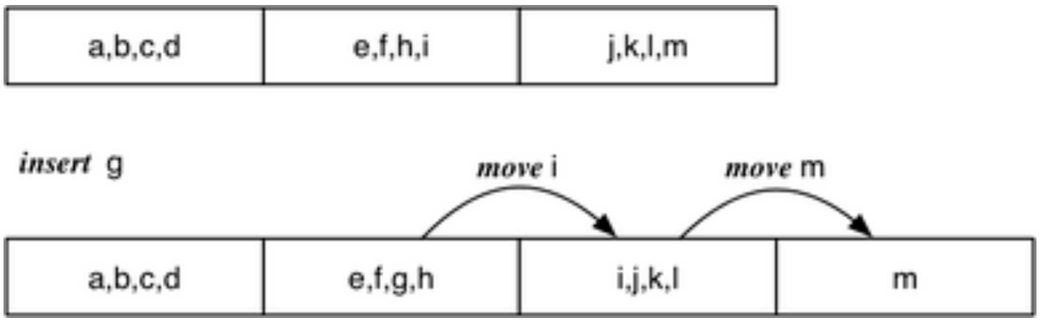
- keeps track of pages with all "live" tuples
  (i.e. all tuples in page are visible to all active transactions)
- organised as a bit-map (one bit per page)
- makes vacuum faster (can ignore all-live pages)
- may help to sped up tuple visibility checks in future

Sorted Files

Sorted Files

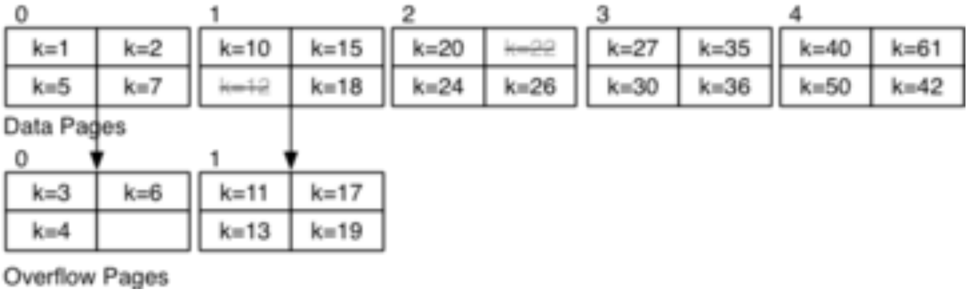
Records stored in file in order of some field *k* (the sort key).

Makes searching more efficient; makes insertion less efficient



... Sorted Files

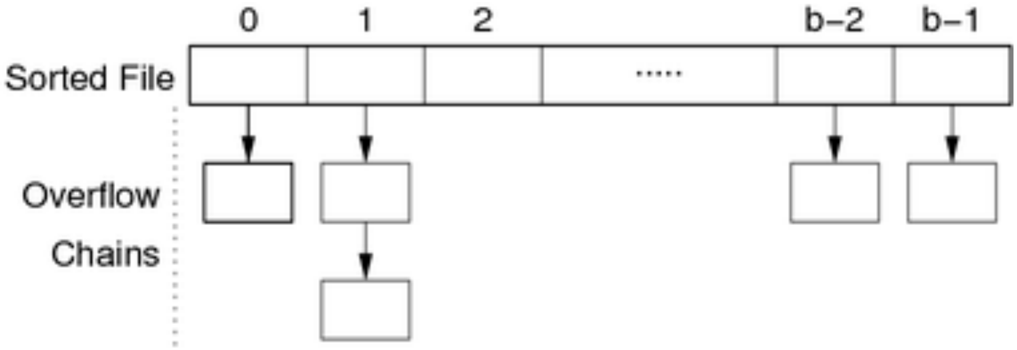
In order to simplify insertion, use overflow blocks.



Large-scale file re-arrangement occurs less frequently.

... Sorted Files

Conceptual view of sorted file structure:



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) div 2;
    buf = getPage(f,mid);
    (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;
```

... 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

34/164

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

```
searchPage(buf,k,val,min,max)
{
    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

35/164

The above method treats each bucket as 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 = $log_2\ b + b_{ov}$

... Selection in Sorted Files

36/164

Average $Cost_{one}$ is messier to analyse:

- complete most of binary search ($(log_2 b)-1$)
- in each unsuccessful bucket, examine $1+Ov$ pages
- in the successful bucket, examine $(1+Ov)/2$ pages

$Cost_{one}$: Average $\cong$ $((log_2 b)-1)(1+Ov)$

To be more "precise"

- $n = (log_2 b)-1$ = number of buckets examined
- Average $Cost = n(1+Ov) + 1 + (Ov/2)$

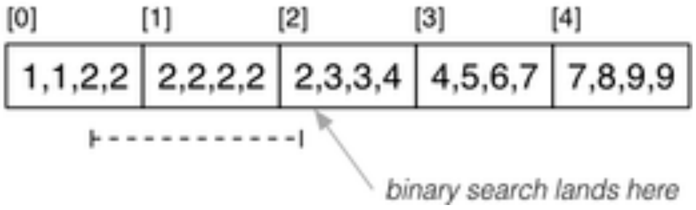
In general, average case is difficult to analyse since dependent on other factors such as data distribution.

... Selection in Sorted Files

37/164

For *pmr* query, on non-unique attribute *k*

- assume file is sorted on *k*
- tuples containing *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, $Cost_{pmr} = Cost_{one} + (b_q\text{-}1).(1+Ov)$

... Selection in Sorted Files

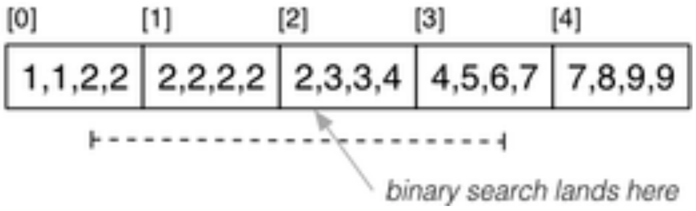
38/164

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

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

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

If secondary key, similar method to *pmr*.



... Selection in Sorted Files

39/164

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

Insertion in Sorted Files

40/164

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

Deletion in Sorted Files

41/164

Deletion involves:

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

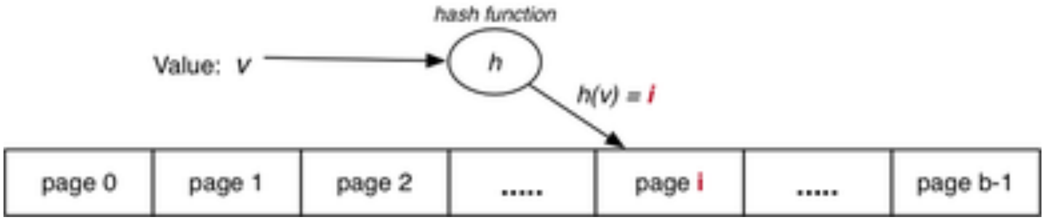
Cost depends on selection criteria   (i.e. on how many matching tuples there are)

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

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]*.

Hash Functions

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

... Hash Functions

Usual approach in hash function:

- convert key into numeric value (how to do this depends on key type)
- numeric value can be viewed as an arbitrary-length bit-string
- value is mapped into page address space

May need separate hash function for each data type

Or, convert each data value to a string and hash that.

... Hash Functions

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);
}
```

... Hash Functions

Where `mix` is defined as:

```
#define mix(a,b,c) \
{ \
  a -= b; a -= c; a ^= (c>>13); \
  b -= c; b -= a; b ^= (a<<8);  \
  c -= a; c -= b; c ^= (b>>13); \
  a -= b; a -= c; a ^= (c>>12); \
  b -= c; b -= a; b ^= (a<<16); \
  c -= a; c -= b; c ^= (b>>5);  \
  a -= b; a -= c; a ^= (c>>3);  \
  b -= c; b -= a; b ^= (a<<10); \
  c -= a; c -= b; c ^= (b>>15); \
}
```

i.e. scrambles all of the bits from the bytes of the key value

See **`backend/access/hash/hashfunc.c`** for details.

... Hash Functions

Above functions give 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 blocks
- have most blocks nearly full (attempt to minimise wasted space)

Note: if data distribution not uniform, block address distribution cannot be uniform.

Best case: every block contains same number of tuples.

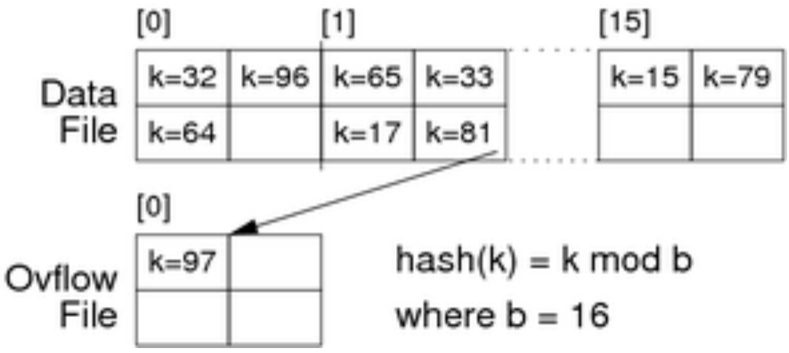
Worst case: every tuple hashes to same block.

Average case: some blocks have more tuples than others.

... Hashing Performance

With non-uniform distribution or too many tuples, some blocks will fill.

Use overflow blocks to contain "excess" tuples; one chain of overflow blocks for each primary block (containing all tuples with same hash value).



... Hashing Performance

Trade-off (assuming reasonably uniform hash function):

- adding more pages to data file reduces collisions (⇒ overflows)

  effect: reduces long overflow chains which cause query-time overhead

- adding more pages to data file results in more unused storage space

  effect: half-full blocks result in storage overhead

Want to minimise overflow chain length, yet keep blocks as full as possible.

... Hashing Performance

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	≃ *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

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 hash key *k* (*one*)

Overview:

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

... Selection with Hashing

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;
}   }
```

... Selection with Hashing

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

```
// 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*

... Selection with Hashing

Details of selection via hashing on non-unique hash key *nk* (*pmr*)

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

Unless the hash function is order-preserving (and most aren't) hashing does not help with *range* queries.

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

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

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

Insertion with Hashing

Insertion uses similar process to *one* queries.

Overview:

```
pid,P = hash(val) % nPages(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
```

... Insertion with Hashing

Details of insertion into hashed file:

```
f = openFile(fileName("R"),READ|WRITE);
p = hash(tup.k) % nPages(R);
buf = getPage(f,p);
if (!isFull(buf) && addTuple(buf,tup) >= 0)
    { writePage(f, p, buf); return; }
ovf = openFile(ovFileName("R"),READ|WRITE);
```

```
p = ovFlow(buf); hasOvFlow = (p != NO_PAGE);
while (p != NO_PAGE) {
    buf = getPage(ovf,p);
    if (!isFull(buf) && addTuple(buf,tup) >= 0)
        { writePage(ovf, p, buf); return; }
    p = ovflow(buf);
}
ovflow(buf) = nPages(ovf);
writePage((hasOvFlow?ovf:f), p, buf);
clear(buf);
addTuple(buf, tup);
writePage(ovf, nPages(ovf), buf);
```

... Insertion with Hashing

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

Deletion with Hashing

Similar performance to select on non-unique key:

```
// delete from R where k = val
pid,P = hash(val) % nPages(R);
ndel = delTuples(P,k,val)
if (ndel > 0) putPage(f,buf,p)
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.

... Deletion with Hashing

More details on deletion with hashing

```
// 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)
pid = ovFlow(buf)
while (pid != NO_PAGE) {
    buf = getPage(ovf,pid);
    delTuples(ovf,buf,k,val);
    pid = ovFlow(buf);
}
```

... Deletion with Hashing

Function for deleting all matches in a buffer:

```
delTuples(outf, buf, k, val)
{
    int i;
    int ndels = 0;
    for (i = 0; i < nTuples(buf); i++) {
        tup = getTuple(buf,i);
        if (tup.k == val)
            { ndels++; delTuple(buf,i); }
    }
    if (ndels > 0)
        writePage(outf, p, buf);
}
```

Realistic deletion would also handle free-list of empty pages.

Another Problem with Hashing...

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

This is known as *static hashing*.

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)

... Another Problem with Hashing...

If we change the file size, by adding more blocks to accomodate more tuples, then we are effectively changing the hash function.

In order to be able to use the new hash function to find existing tuples, all tuples need to be removed and re-inserted.

In other words, this requires complete re-organisation of the file.

Obvious disadvantages:

- the cost of re-organisation (consider re-inserting 10^6 tuples)
- lack of access to the data during reorganisation.

Flexible Hashing

Several hashing schemes have been proposed to deal with dynamic files.

Two methods access blocks via a *directory*:

- *extendible hashing*, dynamic hashing

A third method expands files systematically, so needs no directory:

- *linear hashing*

All methods:

- treat hash value as bit-string (e.g. unsigned 32-bit int)
- adjust hash function by altering number of bits considered

... Flexible Hashing

Require a function to act on hash values to give *d* bits

```
#define HashSize 32
typedef unsigned int HashVal;

// return low-order d bits
HashVal bits(d int, h HashVal)
{
    HashVal mask;
    assert(d > 0 && d <= HashSize);
    mask = 0xffffffff >> (HashSize-d);
    return (h & mask);
}

// return high-order d bits
HashVal bits'(d int, h HashVal)
{
    assert(d > 0 && d <= HashSize);
    return (h >> (HashSize-d));
}
```

... Flexible Hashing

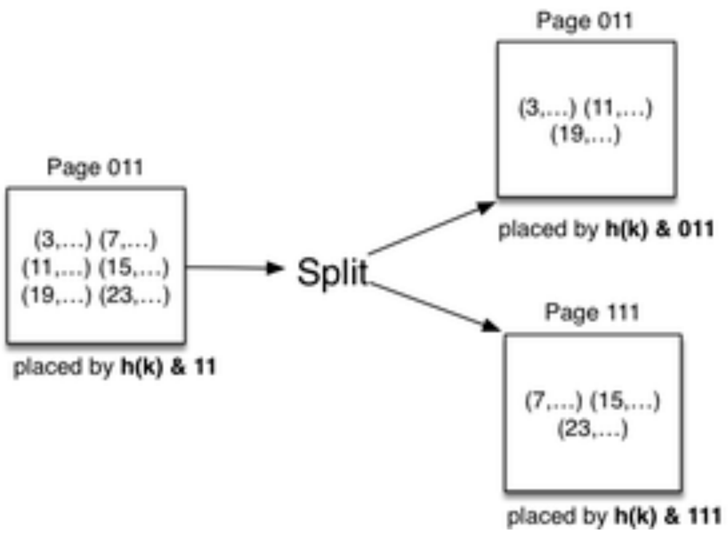
Important concept for flexible hashing: *splitting*

- 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`

Aim: expandable data file, requiring minimal large reorganisation

... Flexible Hashing

Example of splitting:



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

Extendible Hashing

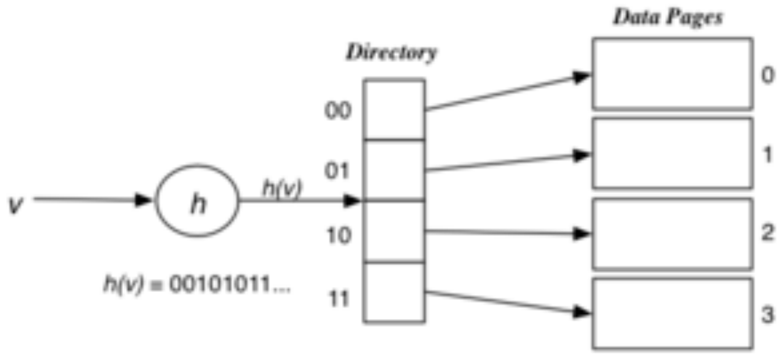
File organisation:

- 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.

... Extendible Hashing



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.

Insertion with Ext.Hashing

For insertion, use selection method to determine block.

If selected block *S* not full, simply insert tuple.

What to do if selected block full?

- add a new block *N* to the file
- partition tuples between blocks *S* and *N*

The partitioning process is called *splitting*.

... Insertion with Ext.Hashing

Basis for splitting?

All tuples `r` in this block have same `bits'(d,hash(r.k))`

So ... use *d+1* bits of hash (i.e. `bits'(d+1,hash(r.k))`

This gives twice as many possible hash values.

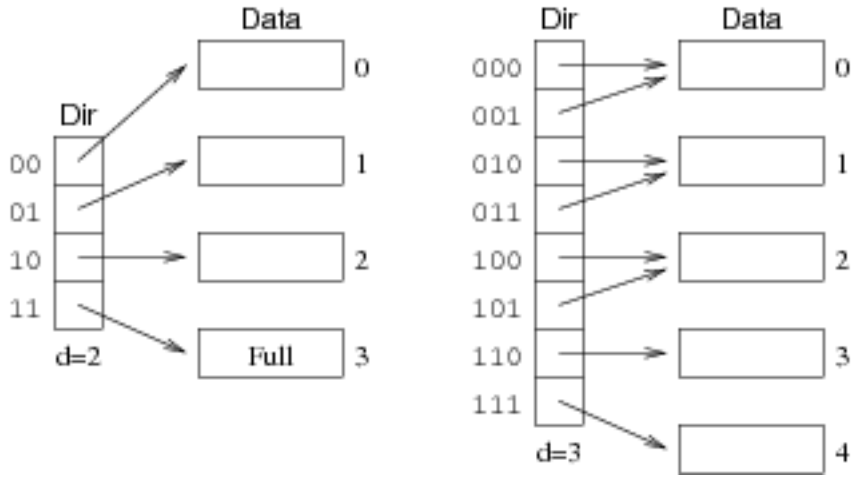
Since hash values index directory, directory size is doubled.

E.g. *d=4* gives indexes 0..15, *d=5* gives indexes 0..31

... Insertion with Ext.Hashing

Doubling directory size and adding one block creates many empty directory slots.

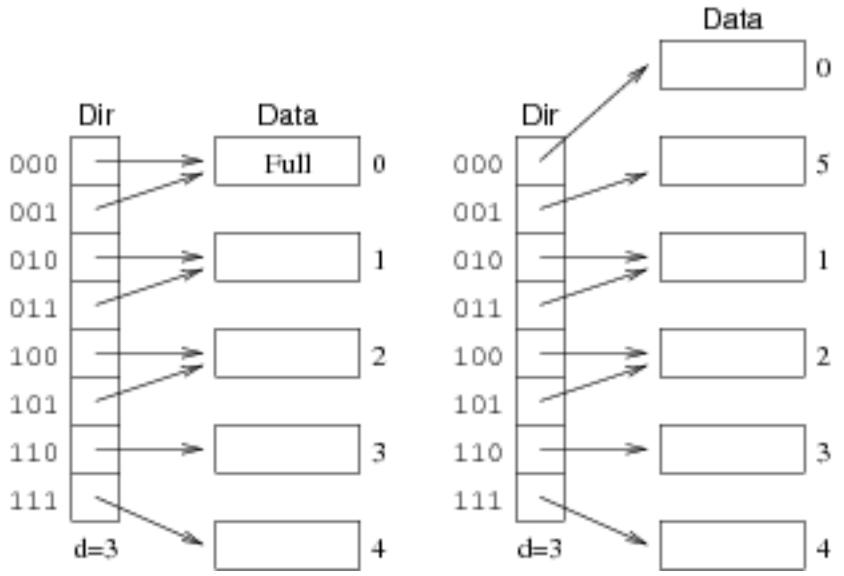
What to do with these slots? Make them point to existing (buddy) pages.



The idea: some parts are hashed effectively using *d* bits, others using *d+1* bits.

... Insertion with Ext.Hashing

If we split a block with two pointers to it, no need to make directory larger.



Ext.Hashing 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

... Ext.Hashing Example

79/164

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

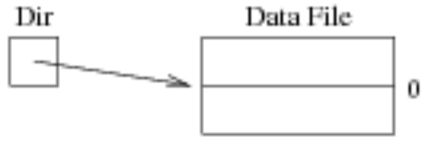
Branch	Hash Value
Brighton	`0010 1101 1111 1011`
Clearview	`1101 0101 1101 1110`
Downtown	`1010 0011 1010 0000`
Mianus	`1000 0111 1110 1101`
Perryridge	`1111 0001 0010 0100`
Redwood	`1011 0101 1010 0110`
Round Hill	`0101 1000 0011 1111`

... Ext.Hashing Example

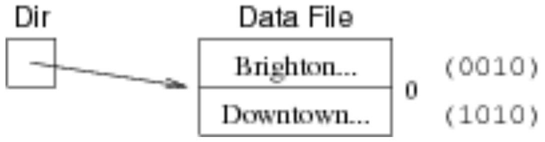
80/164

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

Start with an initially empty file:



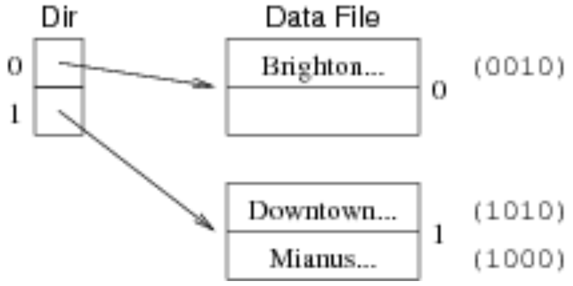
Add `Brighton...` tuple (hash=`0010...`).
Add `Downtown...` tuple (hash=`1010...`).



... Ext.Hashing Example

81/164

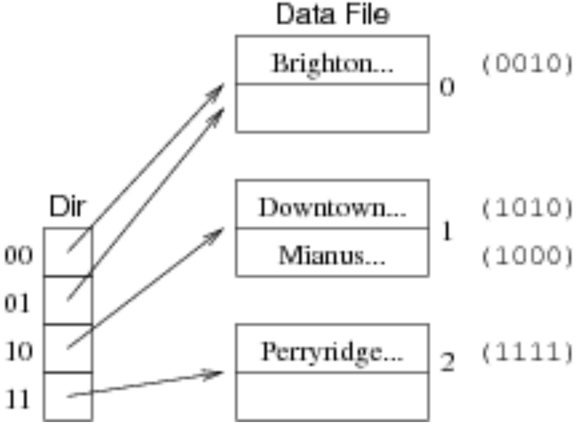
Add `Mianus...` tuple (hash=`1000...`).



... Ext.Hashing Example

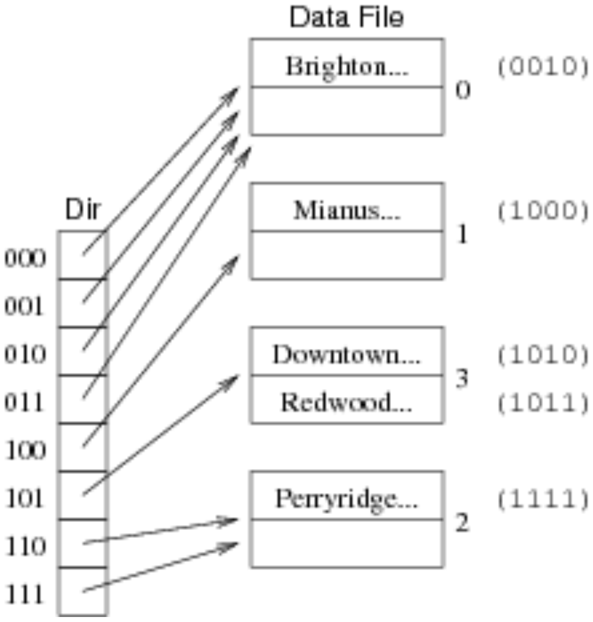
82/164

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



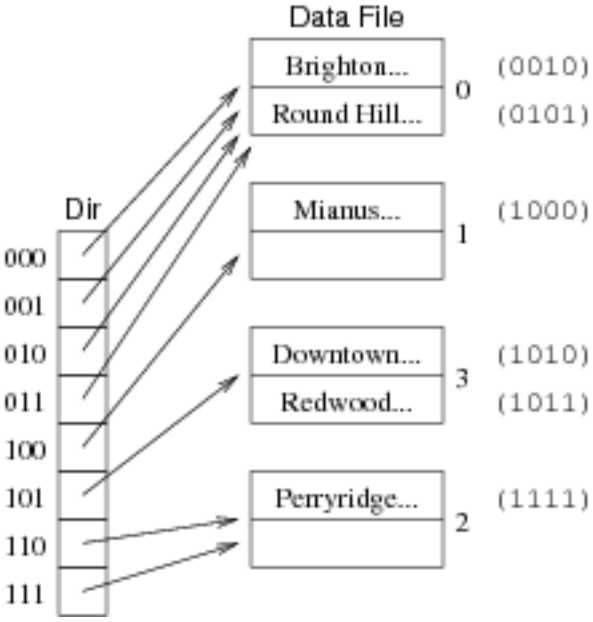
... Ext.Hashing Example

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



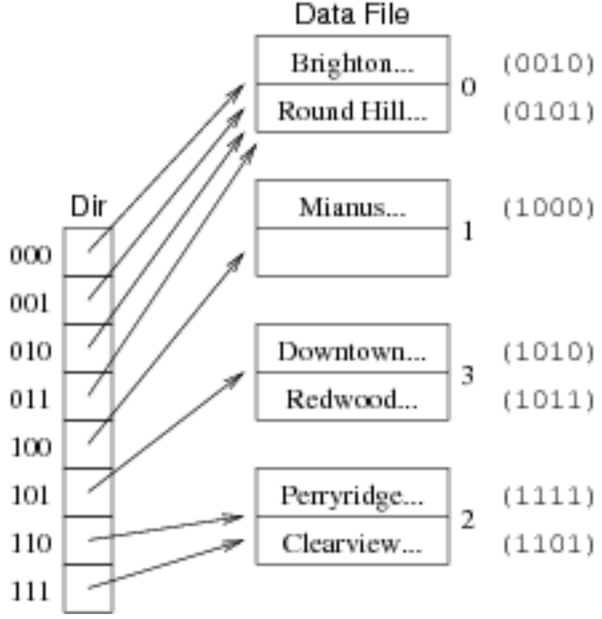
... Ext.Hashing Example

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



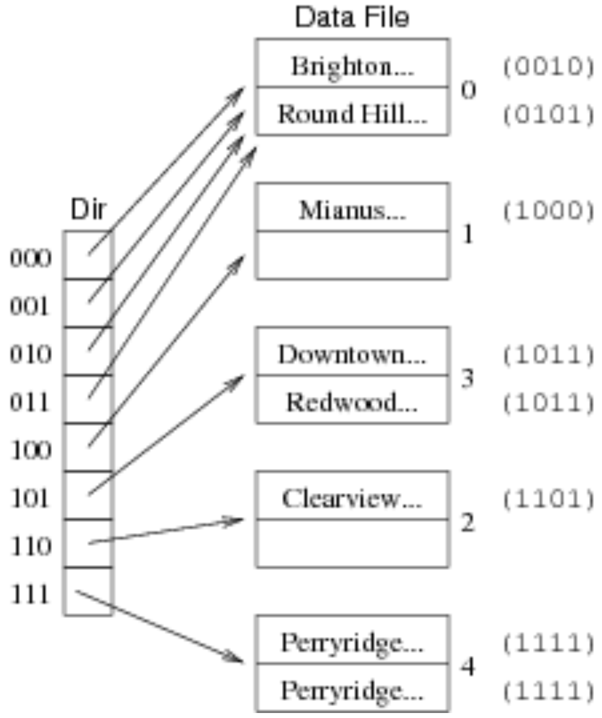
... Ext.Hashing Example

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



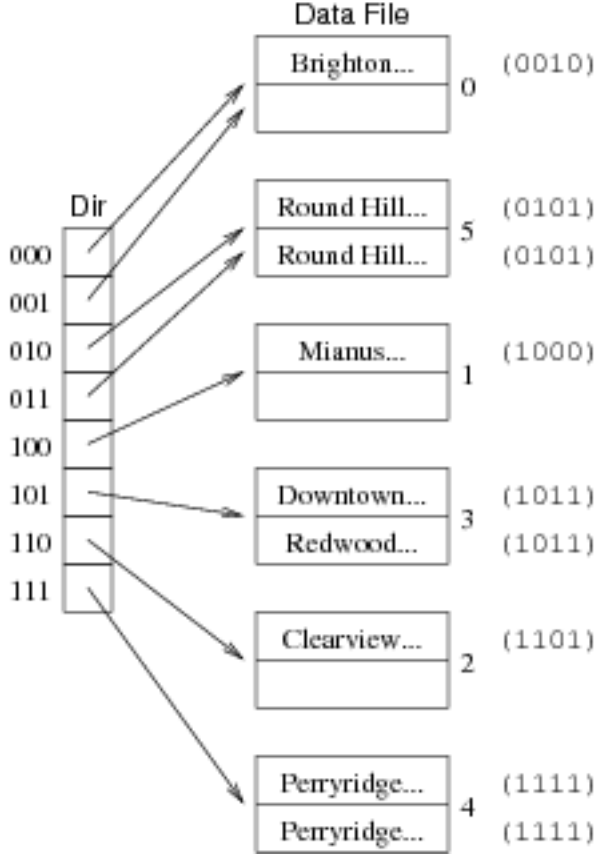
... Ext.Hashing Example

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



... Ext.Hashing Example

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



... Ext.Hashing Example

Asumption: directory is small enough to be stored in memory.

Most of the time we read and write exactly one block.

If we need to split, we write just one extra block.

Conditions to minimise splitting:

- hash function is good (distributes tuples uniformly)
- blocks contains many tuples ($b \gg 2$)

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

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

... Ext.Hashing Example

A potential problem:

- split might leave all tuples in same page (p)
- new tuple hashes to page p ⇒ no room
- need to split again (eventually tuples differ in $d+n$ bits)

A degenerate case:

- each block holds c tuples
- more than c tuples hash to exact same value
- splitting never disambiguates tuples ⇒ directory explodes

Deletion with Ext.Hashing

Similar to ordinary static hash file - mark tuple as removed.

However, might also wish to reclaim space:

- remove block when empty

- merge two "buddy" blocks if both half full

Both cases:

- remove a single block
- require simple adjustments in directory

... Deletion with Ext.Hashing

What to do with empty blocks from file perspective?

Empty block might create hole in middle of file.

Two methods to deal with this:

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

Linear Hashing

File organisation:

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

Advantages over other approaches:

- does *not* require auxiliary storage for a directory
- does *not* require periodic file reorganisation

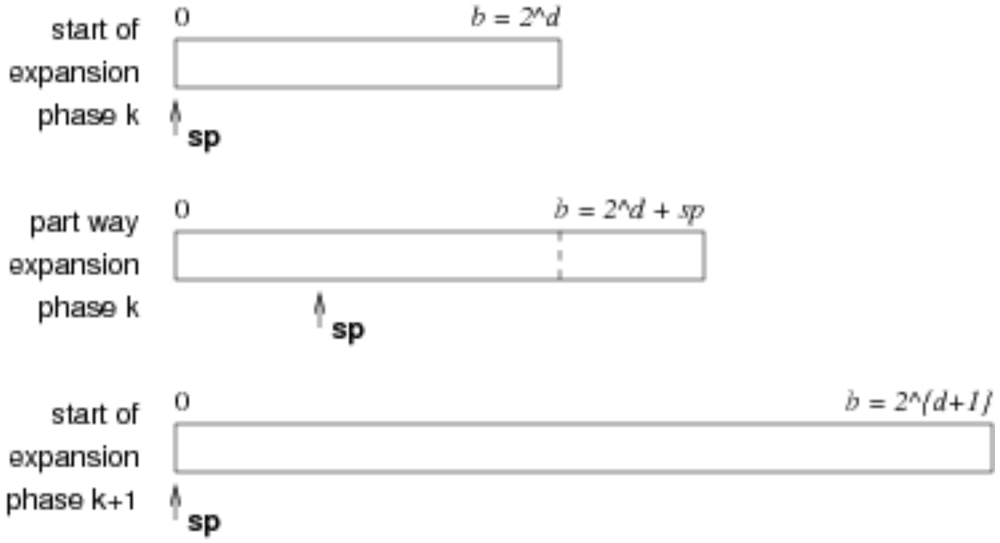
Uses systematic method of growing data file ...

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

... Linear Hashing

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

Selection with Lin.Hashing

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

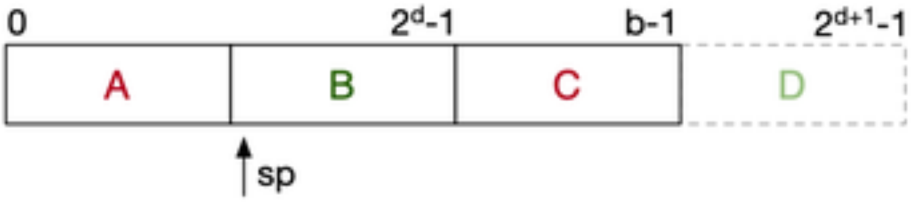
Use *d* bits of hash to compute block address.

```
// select * from R where k = val
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*

... Selection with Lin.Hashing

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 (*B* expands into it).

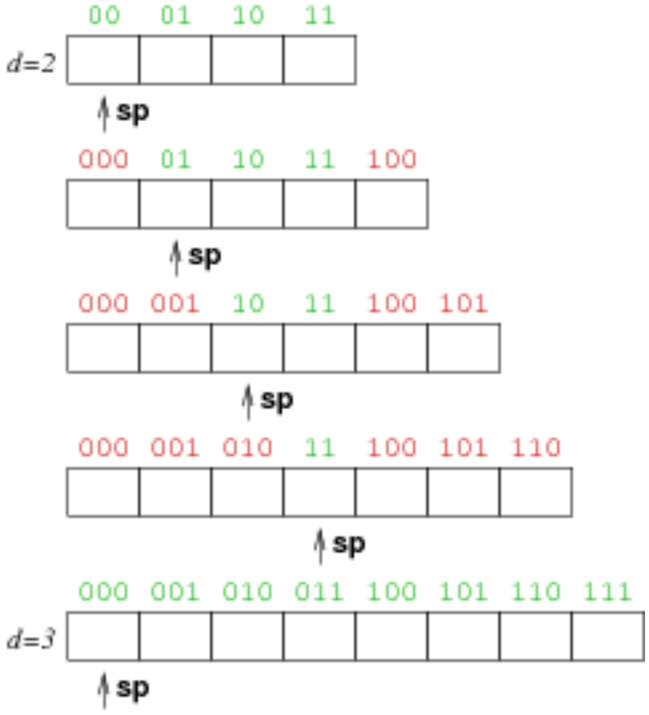
... Selection with Lin.Hashing

Modified algorithm:

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

Insertion with Lin.Hashing

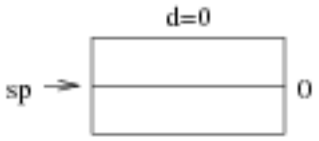
To see why selection works, need to look at how file expands.



Lin.Hashing Example

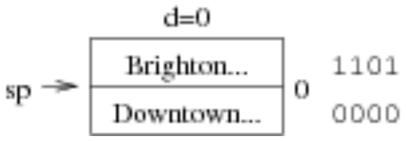
Assume we have a file with: *c=2*.

Start with an initially empty file:



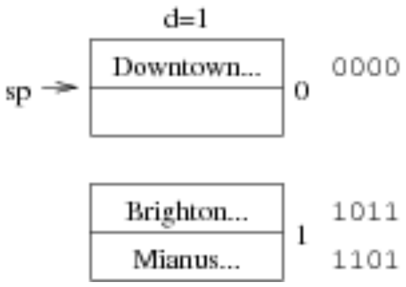
Add `Brighton...` tuple (hash=`...1011`).

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

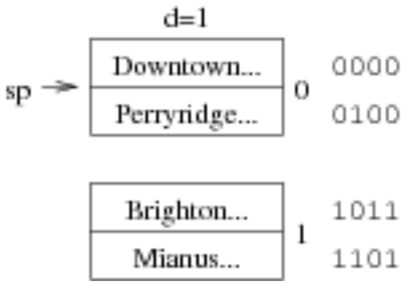


... Lin.Hashing Example

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

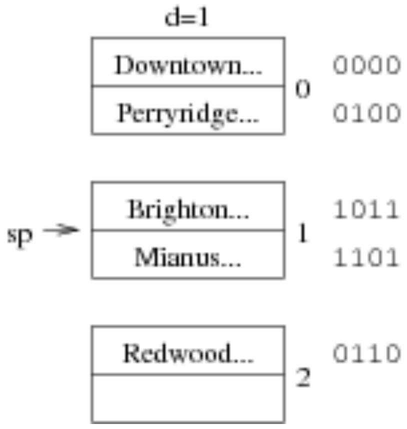


Add `Perryridge...` tuple (hash=`...0100`).



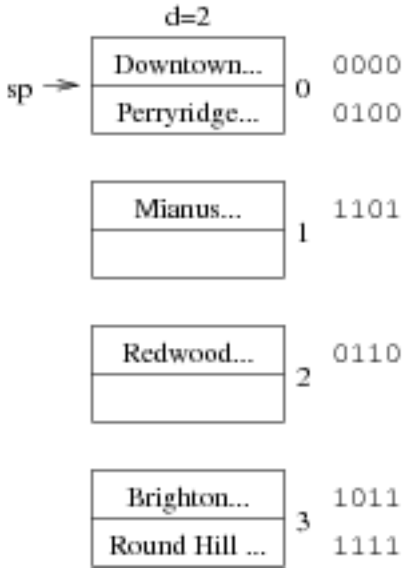
... Lin.Hashing Example

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



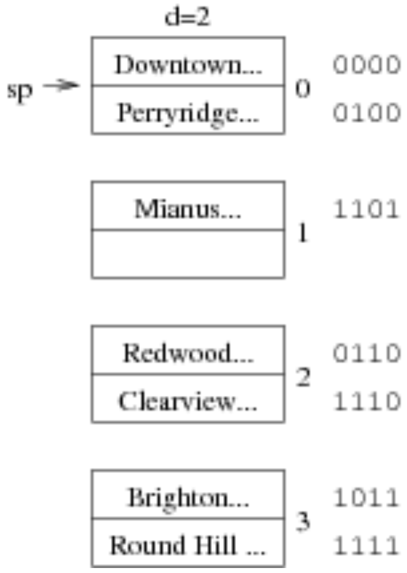
... Lin.Hashing Example

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



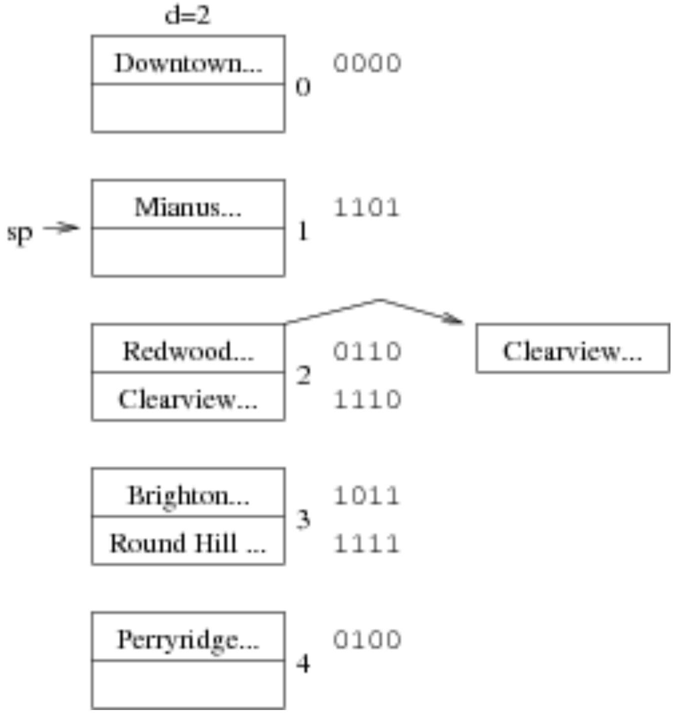
... Lin.Hashing Example

Add `Clearview...` tuple (hash=`...1110`).



... Lin.Hashing Example

Add another `Clearview...` tuple.



Linear Hashing Insertion

Abstract view:

```
p = bits(d,hash(tup.k));
if (p < sp) p = bits(d+1,hash(val));
// bucket p = page p + its ovflow 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; }
}
```

... Linear Hashing Insertion

Detailed algorithm:

```
h = hash(tup.k);
p = bits(d,h);
if (p < sp) p = bits(d+1,h);
//start insertion into p bucket
inserted = 0; inOvflow = 0;
buf = getPage(f,p);
if (isFull(buf))
    { p = ovFlow(buf); }
else {
    // have space in data page
    addTuple(buf,tup); // assume it fits
    writePage(f, p, buf);
    inserted = 1;
}
...
```

... Linear Hashing Insertion

```
...
// attempt insertion into ovflow pages
while (!inserted && p != NO_PAGE) {
    inOvflow = 1;
    buf = getPage(ovf,p);
    if (!isFull(buf)) {
        addTuple(buf,tup);
        writePage(ovf, p, buf);
        inserted = 1;
    }
    if (!inserted)
        { p = ovFlow(buf); }
}
// add a new ovflow page
if (!inserted) {
    ovflow(buf) = nPages(ovf);
    outf = inOvflow ? ovf : f;
    writePage(outf, p, buf);
    clear(buf);
    addTuple(buf, tup);
    writePage(ovf, nPages(ovf), buf);
}
// tuple inserted
```

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*.

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)

... Splitting

How do we accomplish partitioning?

All tuples in *sp* plus its overflow blocks have same hash (e.g. `01`).

New block is at address *sp+2^d^*.

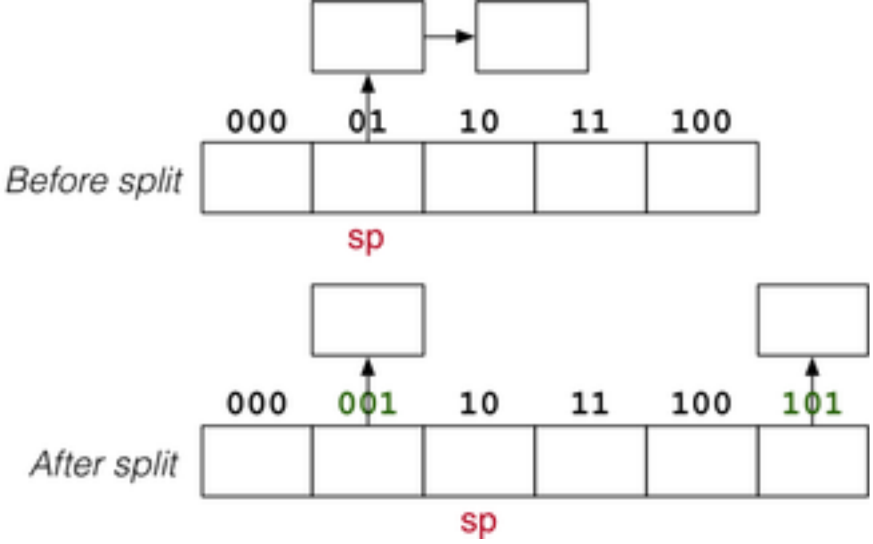
We consider *d+1* bits of hash (giving e.g. `001`, `101`).

Gives addresses for each tuple of *sp* or *sp+2^d^*.

Place tuples according to *d+1* hash bits.

... Splitting

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



... Splitting

Some observations on splitting:

- before split, page *sp* has e.g. *n* ovflow pages
- after split, page *sp* has e.g. *n/2* ovflow pages
- after split, page *sp+2^d* has e.g. *n/2* ovflow pages
- even if split is uneven, still have *n* total ovflow pages

Simplest if we maintain free list of overflow pages.

... 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

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

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

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

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

... Insertion Cost

If split occurs, incur $Cost_{insert}$ plus cost of partitioning.

Partitioning cost involves:

- reading block *sp* (plus all of its overflow blocks)
- writing block *sp* (and its new overflow blocks)
- writing block $sp+2^d$ (and its new overflow blocks)

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

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 block in 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

Note: "bucket" = data page + associated ovflow pages

... Hash Files in PostgreSQL

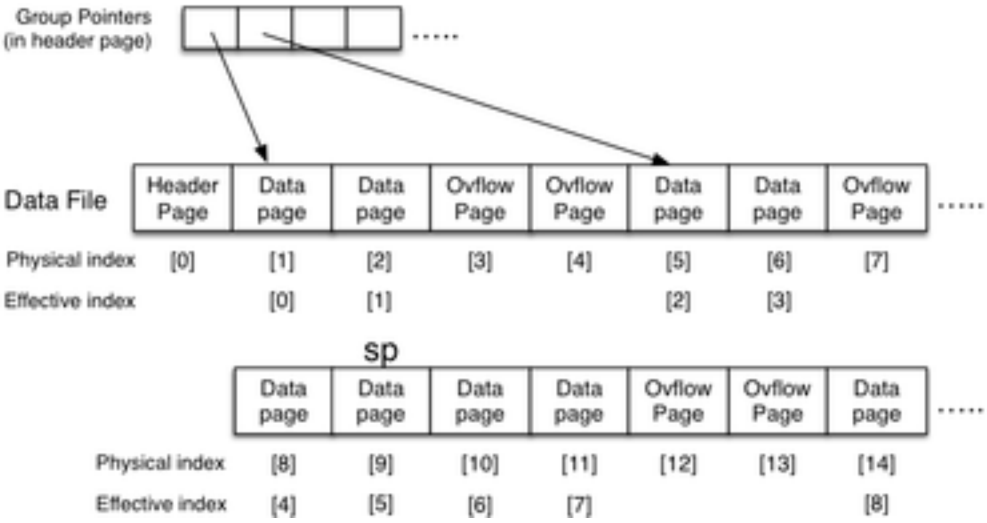
PostgreSQL uses slightly different file organisation ...

- 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;
}
```

Indexes

Selection via Trees

Tree-based file access methods

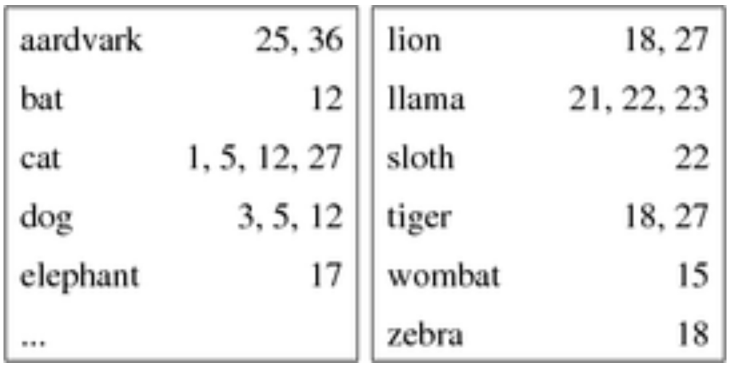
- developed from the notion of indexes

(calling file access methods "indexing schemes" probably derives from this)

- are the oldest kinds of access methods
- are the most widely available inside commercial RDBMSs
- have been developed for most query types

Indexing

The basic idea behind an *index* is as for books.



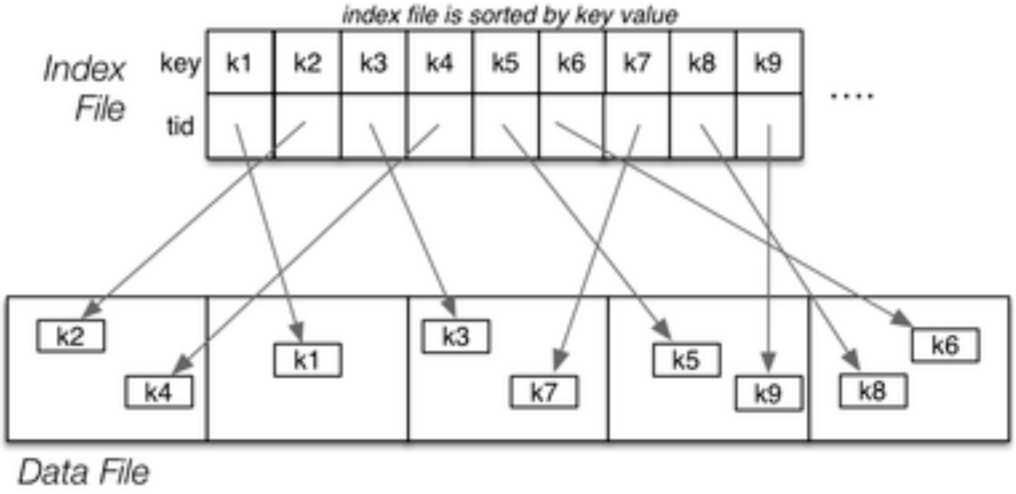
aardvark	25, 36	lion	18, 27
bat	12	llama	21, 22, 23
cat	1, 5, 12, 27	sloth	22
dog	3, 5, 12	tiger	18, 27
elephant	17	wombat	15
...		zebra	18

A table of (key,usage) pairs:

- order by key (for efficient lookup)
- usage is a list of places where the key occurs

Indexing File Structure

An index is a file of (keyVal,tupleID) pairs, e.g.



Indexes

A 1-d *index* is based on the value of a single attribute *A*.

Some possible properties of *A*:

- may be used to sort the file (or may be sorted on some other field)
- values may be unique (or there may be multiple instances)

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

primary	index on unique field, file sorted on this field
clustering	index on non-unique field, file sorted on this field
seconday	file *not* sorted on this field

... Indexes

Indexes themselves may be structured in several ways:

dense	every tuple is referenced by an entry in the index file
sparse	only some tuples are referenced by index file entries
single-level	tuples are accessed directly from the main index file
multi-level	may need to access several index pages to reach tuple

Index structures aim to minimise storage and search costs.

... Indexes

A relation/file may have:

- 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)

Indexes are typically stored in separate files to data (possibly cached in memory).

Primary Index

Primary index: ordered file whose "tuples" have two fields:

- key value (for primary key attribute)
- tuple id (for tuple containing key value)

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)

Index file has blocking factor c_i (where typically $c_i \gg c$)

Index has total i blocks: Dense: $i = \lceil r/c_i \rceil$ Sparse: $i = \lceil b/c_i \rceil$

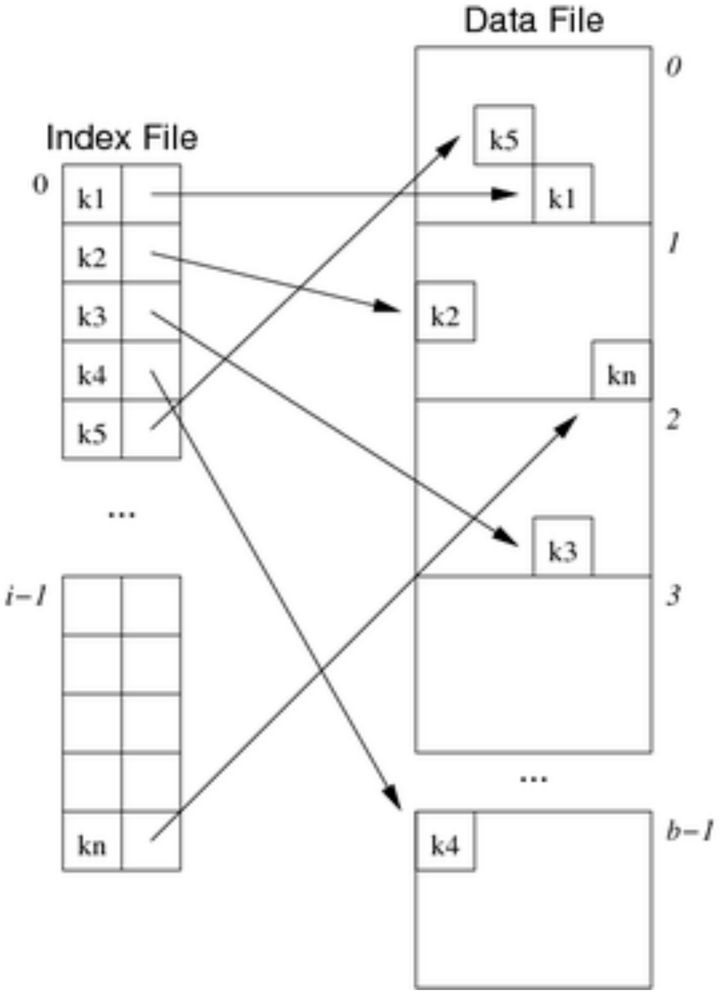
... Primary Index

One possible implementation for index blocks:

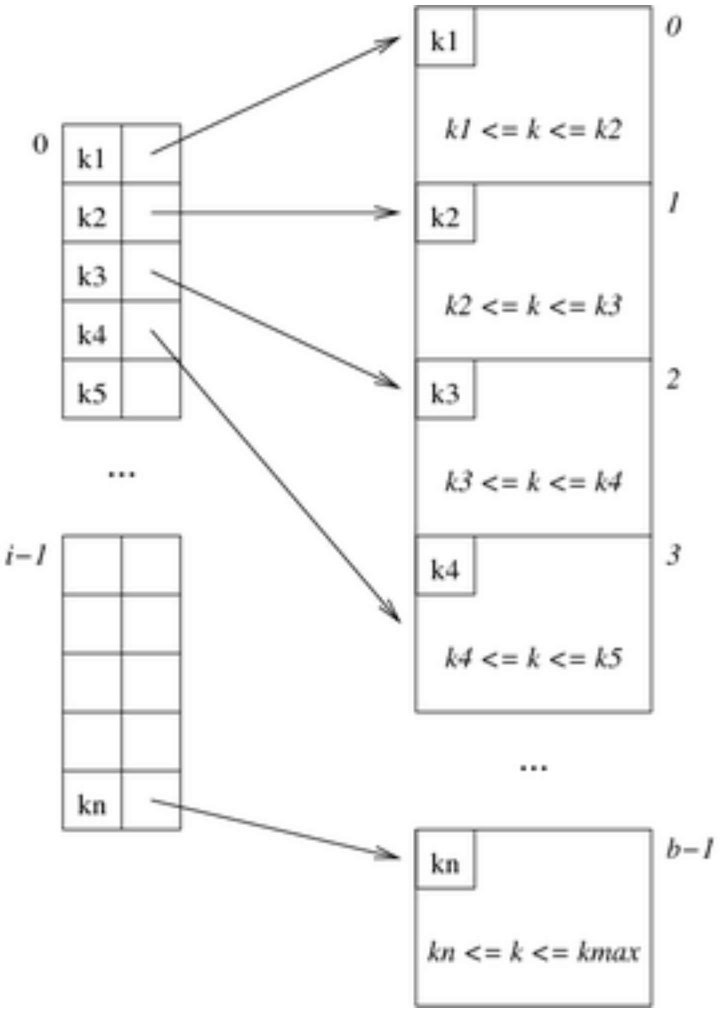
```
typedef struct {
    IndexHeader  info,
    IndexEntry[] entries
} IndexBlock;

typedef struct {
    Datum   key,
    int     pageNum,
    int     tupleNum
} IndexEntry;
```

Note that `pageNum+tupleNum` is equivalent to `TupleId`.

Dense Primary Index



Sparse Primary Index



Index Overheads

Consider a relation and index with the following parameters:

- data file tuples sorted on integer key A
- page size $B=4096$ bytes (for both data and index files)
- each data page has 256-byte header ⇒ 3840 bytes for tuples
- data tuple size $R=50$ bytes ⇒ tuples/page $c=76$
- total number of tuples $r=16,000$ ⇒ total data pages $b=211$
- key value is integer so 4 bytes, assume TupleID is 4 bytes
- since index entry is (value,TupleID) ⇒ index entry is 8-bytes
- assume that each index file page has a 64-byte header

(continued)

... Index Overheads

For each page in the index file ...

- B_i = space available for index entries = 4096-64 = 4032
- index entries per page $c_i = \lfloor B_i/8 \rfloor = 504$

Size of index file:

- dense index: one index entry for every tuple
  - index pages $i = \lceil r/c_i \rceil = \lceil 16,000/504 \rceil = 32$
- sparse index: one index entry for page (min key value)
  - index pages $i = \lceil b/c_i \rceil = \lceil 211/504 \rceil = 1$

Selection with Prim.Index

For *one* queries:

```
ix = binary search index for entry with key K
if nothing found { return NotFound }
b = getPage(ix.pageNum)
t = getTuple(b,ix.tupleNum)
   -- may require reading overflow pages
return t
```

Worst cost: read $log_2 i$ index pages + read $1+Ov$ data pages.

Thus, $Cost_{one,prim} = log_2 i + 1 + Ov$

... 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

For queries not involving primary key, index gives no help.

... Selection with Prim.Index

Method for range queries (when data file is not sorted)

```
// e.g. select * from R where a between lo and hi
pages = {}   results = {}
ixPage = findIndexPage(R.ixf,lo)
while (ixTup = getNextIndexTuple(R.ixf)) {
   if (ixTup.key > hi) break;
   pages += pageOf(ixTup.tid)
}
foreach pid in pages {
   // scan data page plus ovflow chain
   while (buf = getPage(R.datf,pid)) {
      foreach tuple T in buf {
         if (lo<=T.a && T.a<=hi) results += T
}  }  }
```

Insertion with Prim.Index

Overview:

```
tid = insert tuple into page P at position p
find location for new entry in index file
insert new index entry (k,tid) into index file
```

Problem: order of index entries must be maintained.

- avoid overflow pages; index search must be fast
- either reorganise index file or mark index entries

Reogranisation requires, on average, read/write half of index file.

$Cost_{insert,prim} \;=\; log_2 i_r + i/2.(1_r+1_w) + (1+Ov)_r + (1+\delta)_w$

Deletion with Prim.Index

Overview:

```
find tuple using index
mark tuple as deleted
remove index entry for tuple
```

If we delete index entries by marking ...

- $Cost_{delete,prim} \;=\; (log_2\, i + 1 + Ov)_r + 2_w$

If we delete index entry by index file reorganisation ...

- $Cost_{delete,prim} \;=\; (log_2\, i + 1 + Ov)_r + i/2.(1_r+1_w) + 1_w$

Clustering Index

Index on non-unique ordering attribute A_c.

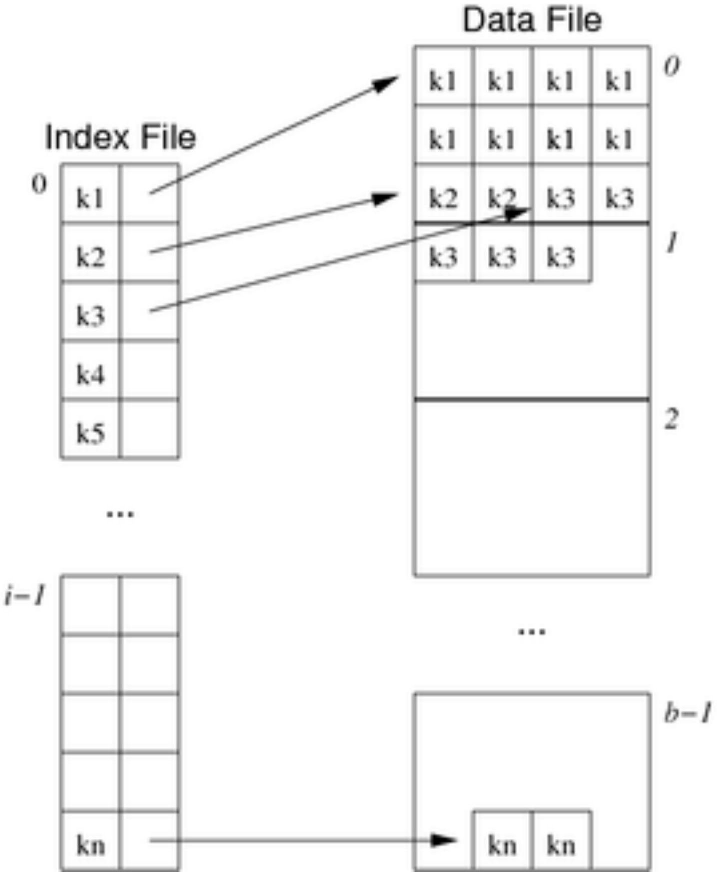
Usually a sparse index; one pointer to first tuple containing value.

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

... Clustering Index



... Clustering Index

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)

Secondary Index

Generally, dense index on non-unique attribute A_s

- data file is not ordered on attribute A_s
- index file *is* ordered on attribute A_s

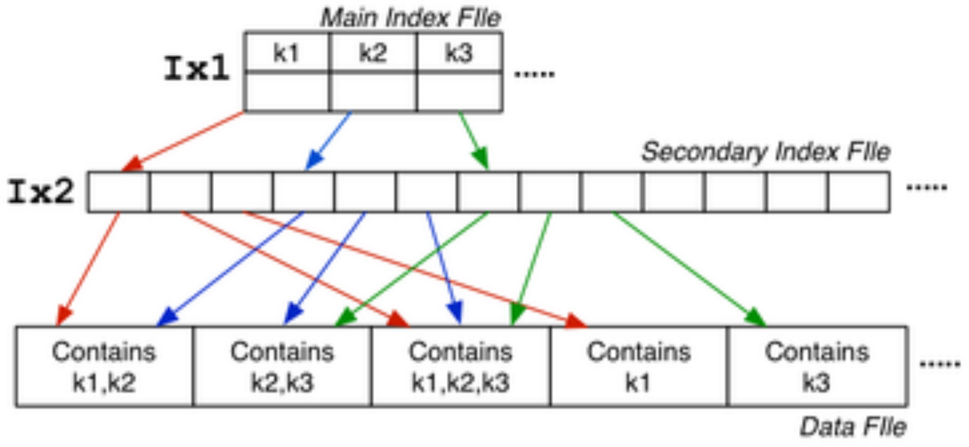
Problem: multiple tuples with same value for A_s.

Three possible solutions:

- have several index entries for value
- have variable length index entries (key value + multiple tupleId's)
- have blocks of tuple addresses, referenced by ``primary index''

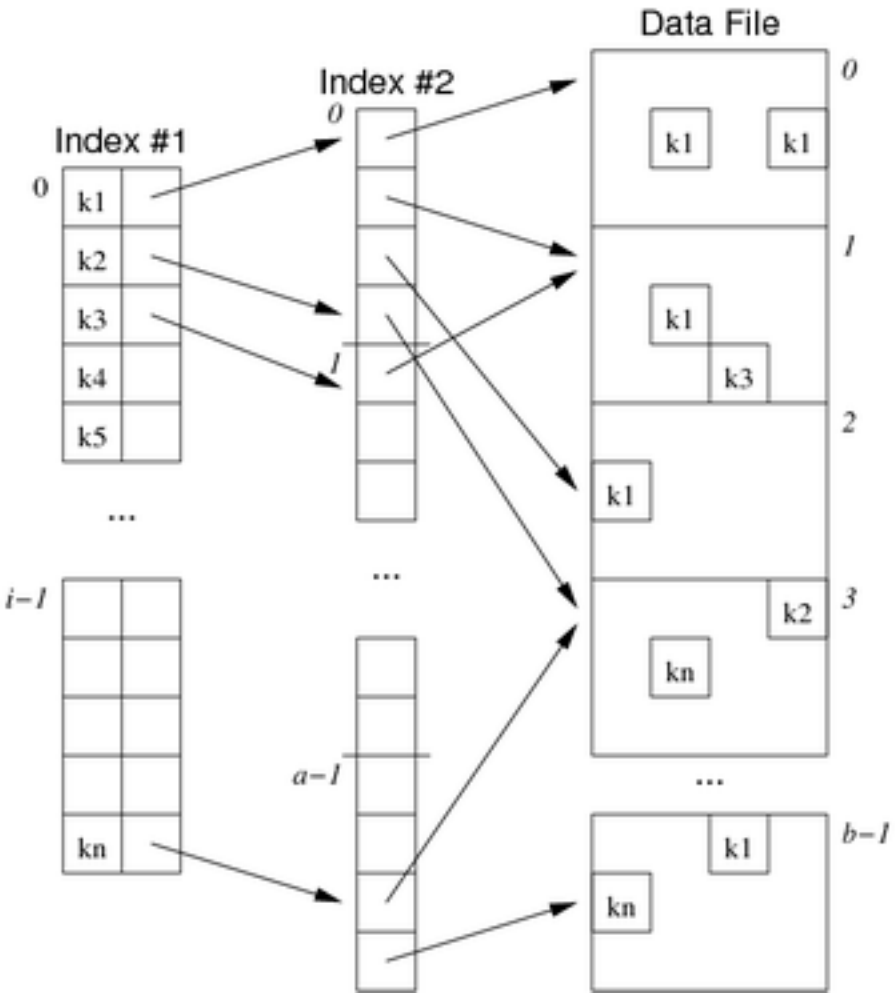
... Secondary Index

First style of secondary index:



... Secondary Index

Third (favoured) style of secondary index:



Selection with Sec.Index

Since non-unique attribute, there are no (explicit) *one* queries.

For *pmr* queries involving A_s:

```
binary search for key K in index file 1
for each entry for K in index file 2 {
    b = getPage(ix.pageNum)
    search for tuple within buffer b
}
```

Assume that answering the query *q* requires:

- binary search of $log_2 i$ index file 1 pages
- read a_q pages in index file 2
- read b_q data blocks + overflow blocks

$Cost_{pmr} = (log_2 i + a_q + b_q \cdot (1 + Ov))_r$

... Selection with Sec.Index

For *range* queries e.g. $\texttt{Lo} \le A_s \le \texttt{Hi}$:

```
binary search for key Lo in index file 1
FOR each entry in index file 2 until Hi DO
    access tuple R via TupleID from index
END
```

Analysis almost same as for *pmr* ...

$Cost_{range} = (log_2 i + a_q + \sum_{k=\texttt{Lo}}^{\texttt{Hi}} (b_k \cdot (1 + Ov)))_r$

Note: may read some blocks multiple times (alleviate with buffer pool)

Insertion/Deletion with Sec.Index

Insertion:

As for primary index:

- may need overflow blocks in data file
- may need to rearrange index file

Deletion:

Can use mark-style (tombstone) deletion for tuples.

Caution: can only mark index-entry for *k* when no more entries for *k* in block-address blocks.

Multi-level Indexes

In indexes described so far, search begins with binary search of index.

Requires $log_2 i$ index block accesses.

We can improve this by putting a second index on the first index.

Second index is sparse; (key,pointer) to first entry in first index block.

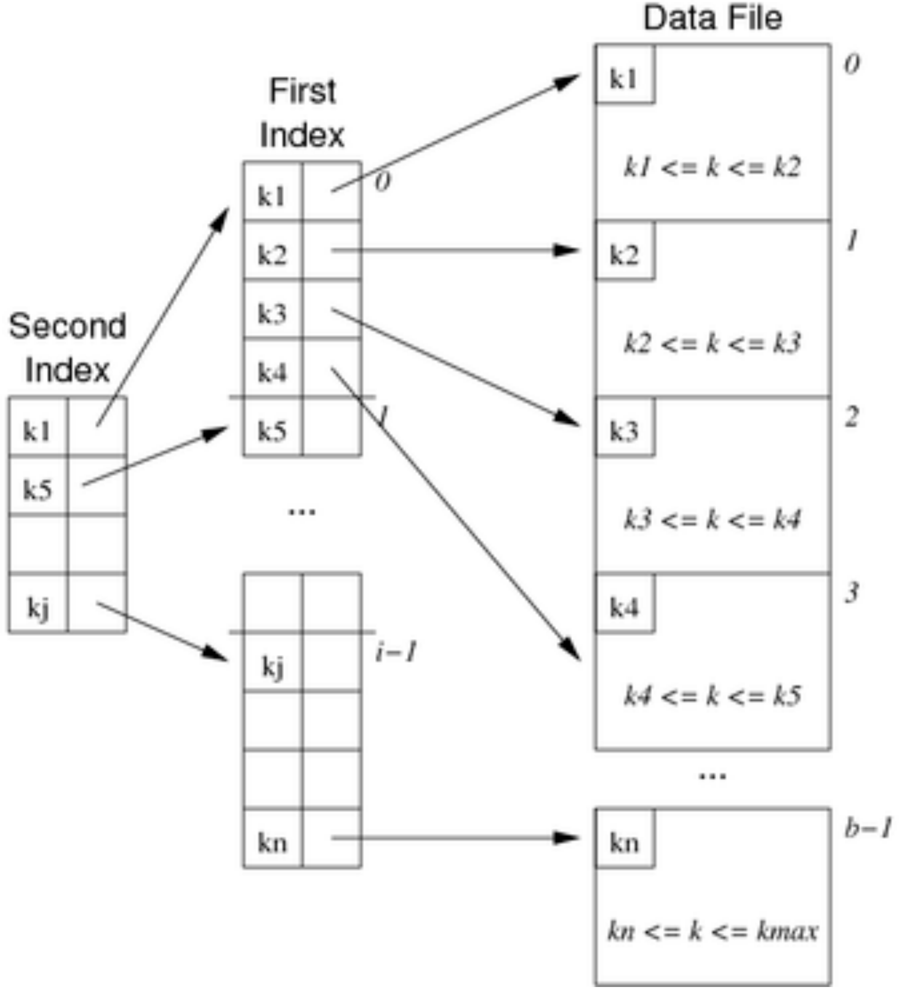
If there are *i* first-level index blocks, there will be $i_2 = \lceil i/c_i \rceil$ second-level index blocks.

If second-level index too big, add third level ...

Maximum levels: $t = \lceil log_{c_i} r \rceil$ (dense index)

Top-level of index is always just a single block.

... Multi-level Indexes



Select with ML.Index

For *one* query on indexed key field:

```
xpid = top level index page
for level = 1 to d {
    read index entriy xpid
    search index page for J'th entry
        where index[J].key <= K < index[J+1].key
    if (J == -1) { return NotFound }
    pid = xpid = index[J].page
}
pid = xpid  // pid is data page index
search page pid and its overflow pages
```

Read *d* index entries and *1+Ov* data blocks.

Thus, $Cost_{one,mli} = (t + 1 + Ov)_r$

(Recall that $t = \lceil log_{c_i} r \rceil$ and single-level index needs $log_2 i$)

B-Trees

B-trees are MSTs with the properties:

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

B-tree insertion and deletion methods

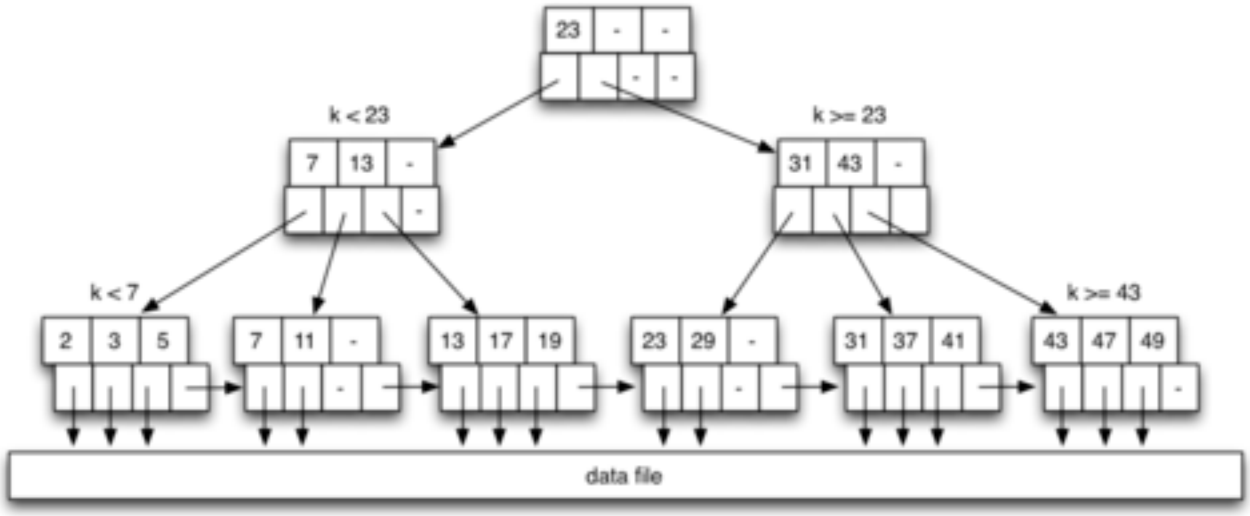
- are moderately complicated to describe
- can be implemented very efficiently

Advantages of B-trees over general MSTs

- better storage utilisation (around 2/3 full)
- better worst case performance (shallower)

... B-Trees

Example B-tree (depth=3, n=3):



B-Tree Depth

Depth depends on effective branching factor (i.e. how full nodes are).

Simulation studies (random insertions and deletions) show typical B-tree nodes are 69% full.

Gives load $L_i = 0.69 \times c_i$ and depth of tree ~ $\lceil \log_{L_i} r \rceil$.

Example: c_i=128, L_i=88

Level	#nodes	#keys
root	1	87
1	88	7656
2	7744	673728
3	681472	59288064

Note: c_i is generally larger than 128 for a real B-tree.

B+Trees

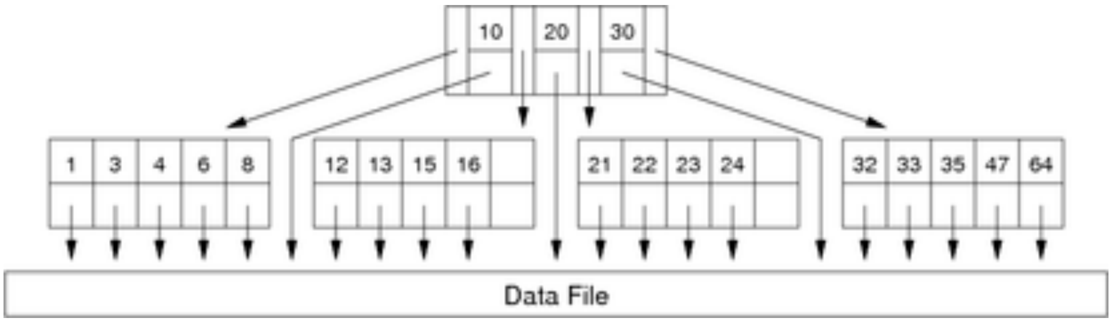
In database context, nodes are index blocks.

Two possible ways to structure them:

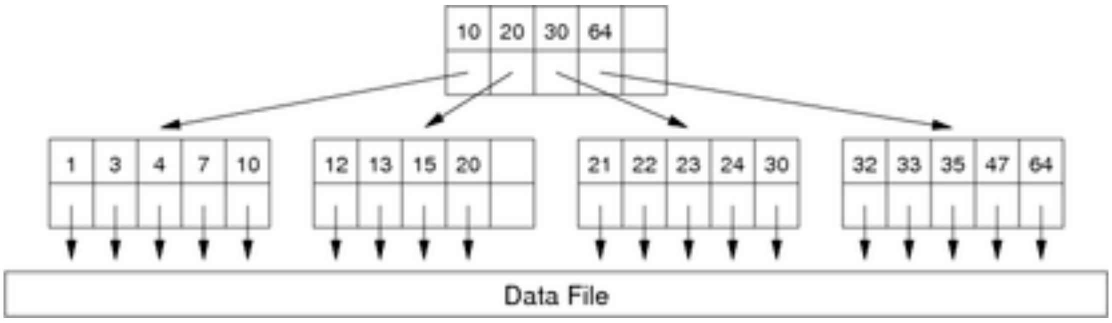
- B trees:
  - some entries in each node have (key, tid) pair
  - other entries (key,ixBlock) pairs (pointers within tree)
- B+ trees:
  - high-level entries contain (key, index block) pairs
  - first-level entries contain (key, data block) pairs

Higher levels of B+ tree have greater c_i ⇒ B+ tree may have less levels.

B-Tree Index



B+Tree Index



Insertion into B-Trees

Overview of the method:

1. find leaf node and position in node where entry would be stored
2. if node is not full, insert entry into appropriate spot
3. if node is full, split node into two half-full nodes and
4. if parent full, split and promote

Note: if duplicates not allowed and key is found, may stop after step 1.

... Insertion into B-Trees

When splitting a node and promoting the middle element, we may find that the parent node is full.

In this case we split the parent in the same way as the child and promote its middle element.

This splitting may propagate all the way to root.

In this case we create a new root containing a single entry.

This is the only exception to the half-full rule.

B-Tree Insertion Cost

Insertion cost = $Cost_{treeSearch} + Cost_{treeInsert} + Cost_{dataInsert}$

Best case: write one page (most of time)

- traverse from root to leaf
- read/write data page, write updated leaf

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

Common case: *3* node writes (rearrange 2 leaves + parent)

- traverse from root to leaf, holding nodes in buffer
- read/write data page
- update/write leaf, parent and new sibling

$Cost_{insert} \;=\; D_r + 1_r + 1_w + 3_w$

... B-Tree Insertion Cost 158/164

Worst case: *2D-1* node writes (propagate to root)

- traverse from root to leaf, holding nodes in buffer
- read/write data page
- update/write leaf, parent and sibling
- repeat previous step *D-1* times

$Cost_{insert} \;=\; D_r + (2D\text{-}1)_w + 1_r + 1_w$

Deletion from B-Trees 159/164

Overview of the method:

1. find entry in node *nd*
2. if *nd* is a non-leaf node:
    1. remove entry
    2. promote its immediate successor from child node to take its place
    3. continue as if operation were deletion of promoted child
3. if *nd* is a leaf node
    1. remove entry
    2. if still more than half full, compact
    3. if < half full, rearrange entries between parent and siblings to make half full

Selection with B+Trees 160/164

For *one* queries:

```
N = B-tree root node
while (N is not a leaf node) {
   scan N to find reference to next node
   N = next node
}
scan leaf node N to find entry for K
access tuple t using TupleId from N
```

$Cost_{one} \;=\; (D + 1)_r$

Generally, *D* is around 2 or 3, even for very large data files.

A further optimisation is to buffer B-tree root page (⇒ total *D* page reads)

... Selection with B+Trees 161/164

For *range* queries (assume sorted on index attribute):

```
search index to find leaf node for Lo
for each leaf node entry until Hi found {
        access tuple t using TupleId from entry
}
```

$Cost_{range} \;=\; (D + b_i + b_q)_r$

(If hold root block in memory ⇒ read *D-1* index blocks)

... Selection with B+Trees

For *pmr*, need index on ≥ *1* query attribute.

Could have indexes on several attributes:

```
Pages = {}
for each Ai = k in query {
    find pages P containing Ai = k
    Pages = Pages ∩ P
}
for each P in Pages {
    scan P for matching tuples
}
```

If *q* mentions a_i, a_j, a_k, then

$Cost_{pmr} \;=\; (D_i + D_j + D_k + |b_{qi} \cap b_{qj} \cap b_{qk}|)_r$

For *space* queries, treat as a combination of *pmr* and *range*.

B-trees in PostgreSQL

PostgreSQL implements ≅ Lehman/Yao-style B-trees

- variant that works effectively in high-concurrency environments.

B-tree implementation: **`backend/access/nbtree`**

- **`README`** ... comprehensive description of methods
- **`nbtree.c`** ... interface functions (for iterators)
- **`nbtsearch.c`** ... traverse index to find key value
- **`nbtinsert.c`** ... add new entry to B-tree index

Notes:

- stores all instances of equal keys
- avoids splitting by scanning right if key = max(key) in page
- common insert case: new key is max(key) overall; handled efficiently

... B-trees in PostgreSQL

Interface functions for B-trees

```
// build Btree index on relation
Datum btbuild(rel,index,...)
// insert index entry into Btree
Datum btinsert(rel,key,tupleid,index,...)
// start scan on Btree index
Datum btbeginscan(rel,key,scandesc,...)
// get next tuple in a scan
Datum btgettuple(scandesc,scandir,...)
// close down a scan
Datum btendscan(scandesc)
```

Produced: 9 Jul 2019