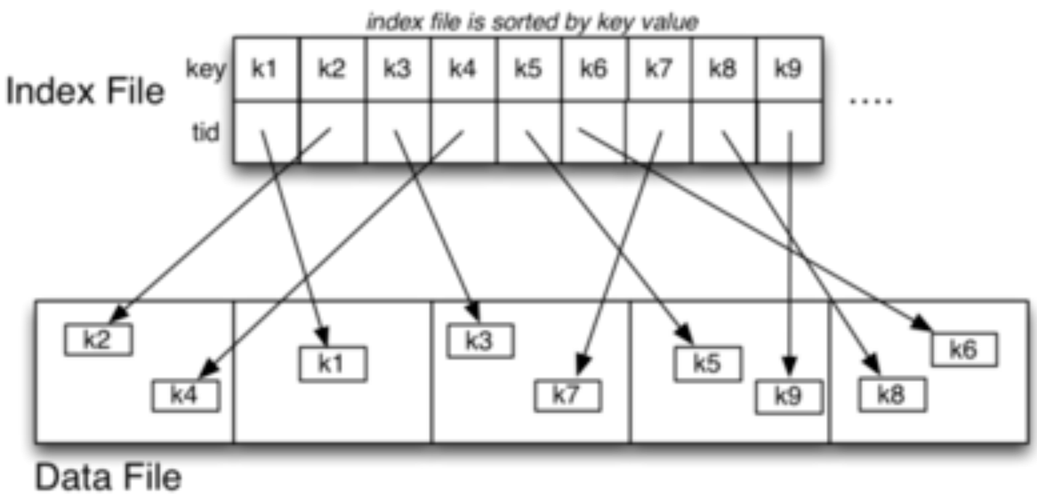
Week 07 Lecture

Indexing

An index is a table/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 data 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, may be sorted on *A*
clustering	index on non-unique field, file sorted on *A*
secondary	file *not* sorted on *A*

A given table may have indexes on several attributes.

... Indexes

Indexes themselves may be structured in several ways:

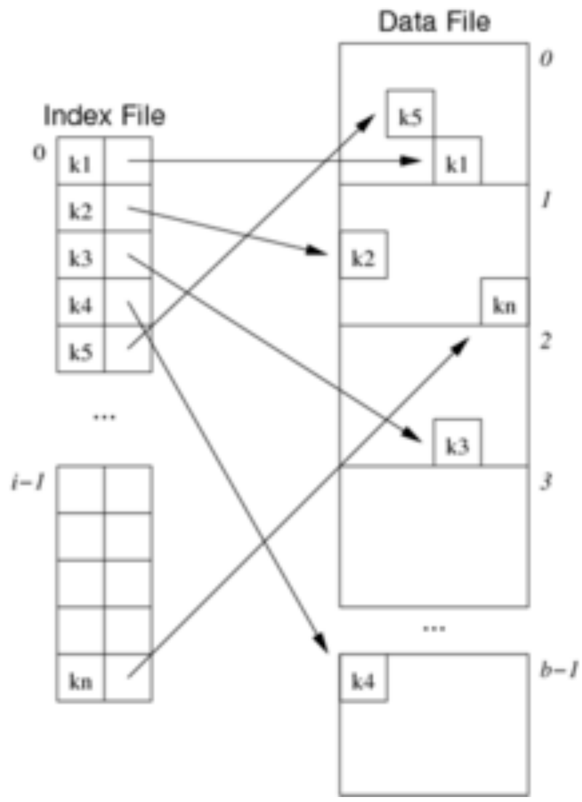
dense	every tuple is referenced by an entry in the index file
sparse	only some tuples are referenced by index file entries
single-level	tuples are accessed directly from the index file
multi-level	may need to access several index pages to reach tuple

Index file has total *i* pages (where typically $i \ll b$)

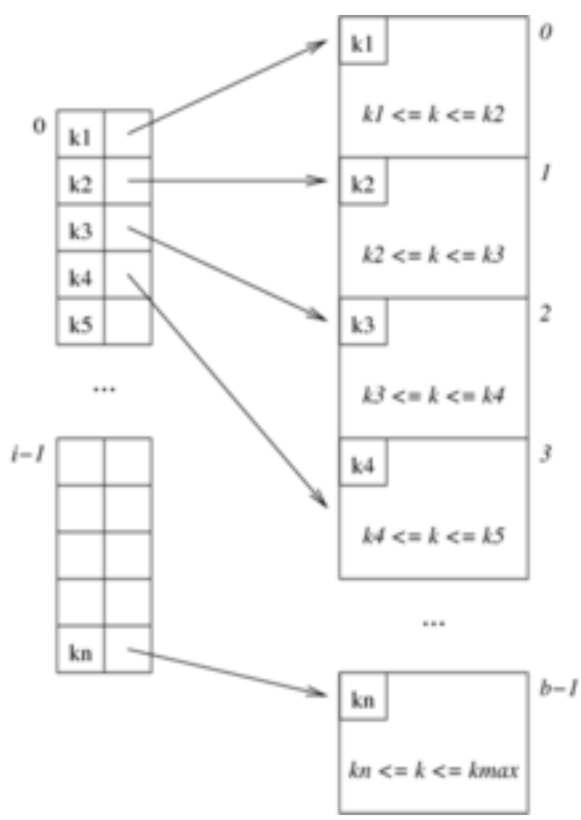
Index file has blocking factor c_i (where typically $c_i \gg c$)

Dense index: $i = ceil(r/c_i)$ Sparse index: $i = ceil(b/c_i)$

Dense Primary Index



Sparse Primary Index



Exercise 1: Index Storage Overheads

Consider a relation with the following storage parameters:

- *B = 8192, R = 128, r = 100000*
- header in data pages: 256 bytes
- key is integer, data file is sorted on key
- index entries (keyVal,tupleID): 8 bytes
- header in index pages: 32 bytes

How many pages are needed to hold a dense index?

How many pages are needed to hold a sparse index?

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

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

Assume: index pages are same size as data pages ⇒ same reading cost

---

... Selection with Prim.Index

For *range* queries on primary key:

- use index search to find lower bound
- read index sequentially until reach upper bound
- accumulate set of buckets to be examined
- examine each bucket in turn to check for matches

For *pmr* queries involving primary key:

- search as if performing *one* query.

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

Exercise 2: Selection with Prim.Index

Consider a range query like

```
select * from R where a between 10 and 30;
```

Give a detailed algorithm for solving such range queries

- assume table is indexed on attribute `a`
- assume file is *not* sorted on `a`
- assume existence of `Set` data type:
  `s=empty(); insert(s, n); foreach elems(s)`
- assume "the usual" operations on relations:
  `r = openRelation(name,mode); b=nPages(r); file(r)`
- assume "the usual" operations on pages:
  `buf=getPage(f,pid); foreach tuples(buf); pid = next(buf)`

Insertion with Prim.Index

Overview:

```
insert tuple into page P
find location for new entry in index file
   // could check whether it already exists
insert new index entry (k,P+i) into index file
   // P+i is tupleID = (PageID + offset within page)
```

Problem: order of index entries must be maintained

- need to avoid overflow pages in index
- so we need to reorganise index file

On average, this requires us to 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
delete index entry for tuple
```

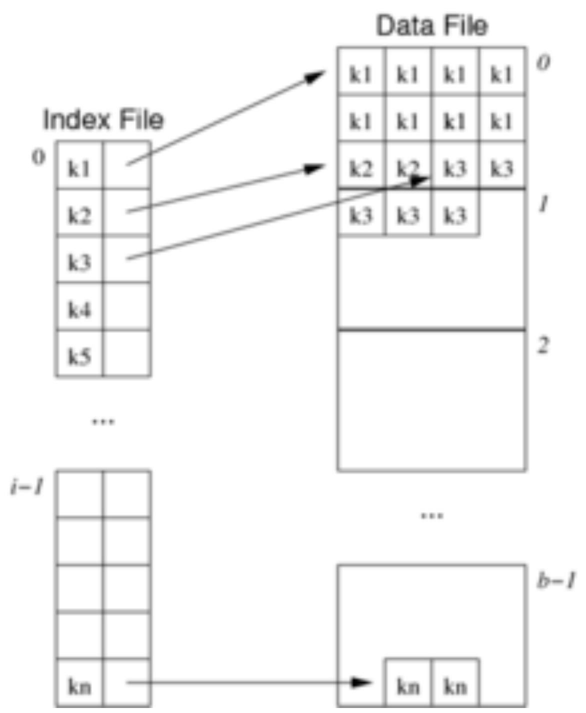
If we delete index entries by marking ...

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

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



... 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 data)
- *pmr* queries involving A_c (search index for specified value)

Insertions are expensive: rearrange index file and data file.

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

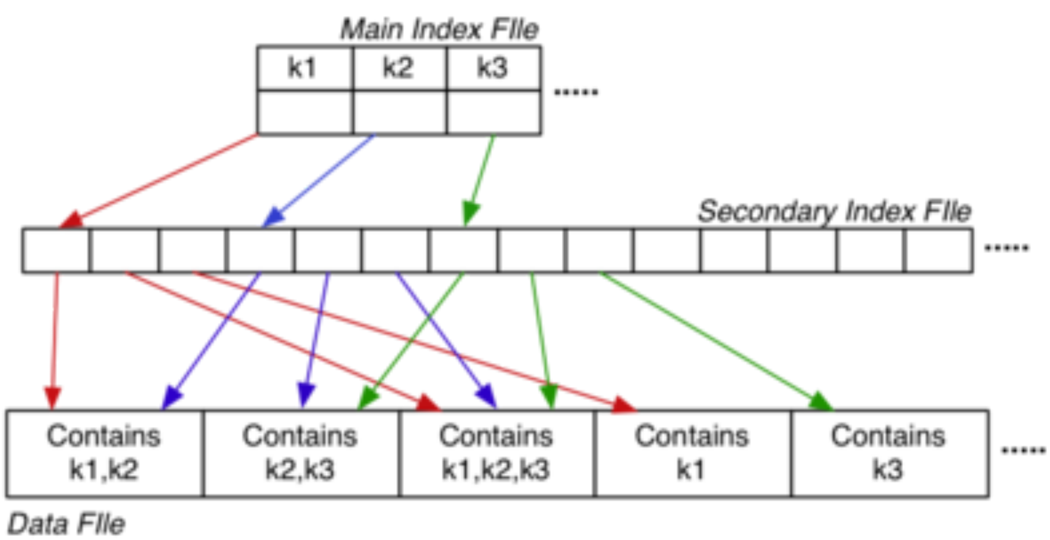
Problem: multiple tuples with same value for A_s.

A solution:

- dense index (`Ix2`) containing just `TupleId`'s
- sparse index (`Ix1`) on dense index containing *(key,offset)* pairs

Each *offset* references an entry in `Ix2`

... Secondary Index



$$Cost_{pmr} = Cost_{range} = (log_2 i + a_{q_2} + b_q.(1 + Ov))$$
$$Cost_{range} = (log_2 i + a_{q_1} + a_{q_2} + b_q.(1 + Ov))$$

Insertion/Deletion with Sec.Index

Insertion:

- each insert requires three files to be updated
- potentially costly rearrangement of index files

Deletion:

- use mark-style (tombstone) deletion for data tuples
- Ix2 entries: can always mark as "deleted"
- Ix1 entries: mark only after removing last instance for *k* in Ix2
- periodic "vacuum" to reduce storage overhead if many deletions

---

Multi-level Indexes

Above Sec.Index used two index files to speed up search

- by keeping the initial index search relatively quick
- Ix1 small (depends on number of unique key values)
- Ix2 larger (depends on amount of repetition of keys)

- typically, $b_{Ix1} \ll b_{Ix2}$

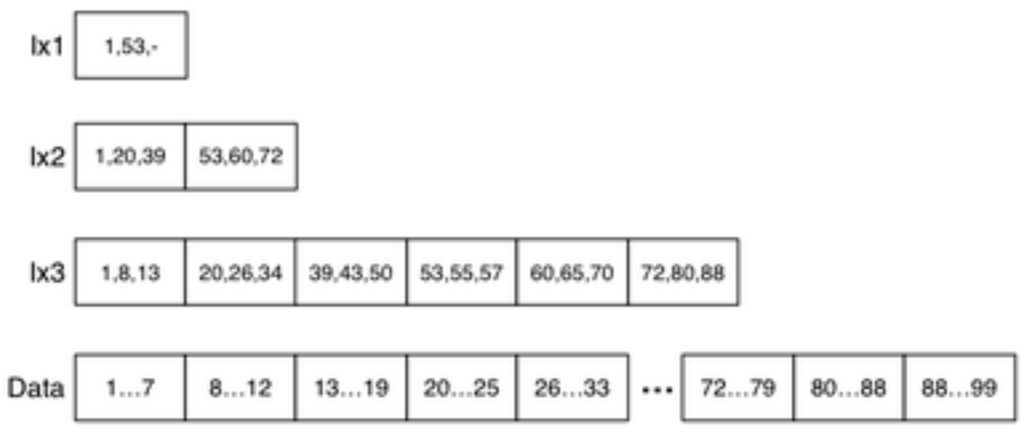
Could improve further by

- making Ix1 sparse, since Ix2 is guaranteed to be ordered
- in this case, $b_{Ix1} = ceil(b_{Ix2} / c_i)$
- if Ix1 becomes too large, add Ix3 and make Ix2 sparse
- if data file ordered on key, could make Ix3 sparse

Ultimately, reduce top-level of index hierarchy to one page.

... Multi-level Indexes

Example data file with three-levels of index:



Assume: not primary key, $c = 100$, $c_i = 3$

Select with ML.Index

For *one* query on indexed key field:

```
I = top level index page
for level = 1 to d {
    read index page I
    search index page for J'th entry
        where index[J].key <= K < index[J+1].key
    if J=0 { return NotFound }
    I = index[J].page
}
-- I is now address of data page
search page I and its overflow pages
```

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

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

(Note that $d = ceil(log_{c_i} r)$ and c_i is large because index entries are small)

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

21/65

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

(Note that nodes are pages, with potential for large branching factor, e.g. *n=500*)

B-Tree Depth

22/65

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

Simulation studies show typical B-tree nodes are 69% full.

Gives load $L_i = 0.69 \times c_i$ and depth of tree ~ $ceil(\log_{L_i} r)$.

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.

Insertion into B-Trees

23/65

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.

24/65

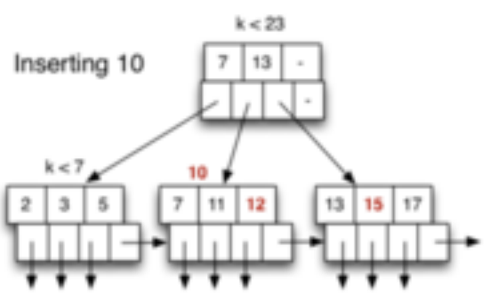
Example: B-tree Insertion

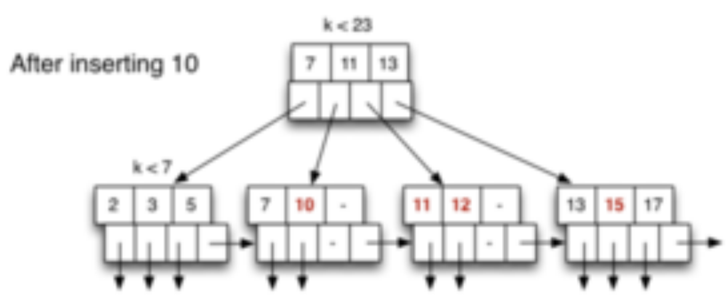
Starting from this tree:

insert the following keys in the given order `12 15 30 10`

... Example: B-tree Insertion

... Example: B-tree Insertion





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 sibling leaf page, hold in buffer
- read/write data page
- update/write leaf, parent and sibling

$Cost_{insert} = (D+1)_r + 3_w + 1_r + 1_w$

... B-Tree Insertion Cost

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

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

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

Selection with B-Trees

For *one* queries:

```
N = B-tree root node
while (N is not a leaf node)
   N = scanToFindChild(N,K)
TupleID = scanToFindEntry(N,K)
access tuple t using TupleID from N
```

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

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
}
```

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

B-trees in PostgreSQL

PostgreSQL implements Lehman/Yao-style B-trees.

A variant that works effectively in high-concurrency environments.

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

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

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)
```

N-dimensional Selection

N-dimensional Queries

Have looked at one-dimensional queries, e.g.

```
select * from R where a = K
select * from R where a between Lo and Hi
```

and *heaps*, *hashing*, *indexing* as ways of efficient implementation.

Now consider techniques for efficient *multi-dimensional* queries.

Compared to 1-d queries, multi-dimensional queries

- typically produce fewer results
- require us to consider more information
- require more effort to produce results

Operations for Nd Select

N-dimensional select queries = condition on ≥*1* attributes.

- *pmr* = partial-match retrieval (equality tests), e.g.

  ```
  select * from Employees
  where  job = 'Manager' and gender = 'M';
  ```

- *space* = tuple-space queries (range tests), e.g.

  ```
  select * from Employees
  where 20 ≤ age ≤ 50 and 40K ≤ salary ≤ 60K
  ```

N-d Selection via Heaps

Heap files can handle *pmr* or *space* using standard method:

```
// select * from R where C
r = openRelation("R",READ);
for (p = 0; p < nPages(r); p++) {
    buf = getPage(file(r), p);
    for (i = 0; i < nTuples(buf); i++) {
        t = getTuple(buf,i);
        if (matches(t,C))
            add t to result set
    }
}
```

$Cost_{pmr} = Cost_{space} = b$

N-d Selection via Multiple Indexes

DBMSs already support building multiple indexes on a table.

Which indexes to build depends on which queries are asked.

```
create table R (a int, b int, c int);
create index Rax on R (a);
create index Rbx on R (b);
create index Rcx on R (c);
create index Rabx on R (a,b);
create index Racx on R (a,c);
create index Rbcx on R (b,c);
create index Rallx on R (a,b,c);
```

But more indexes ⇒ space + update overheads.

N-d Queries and Indexes

Generalised view of *pmr* and *space* queries:

```
select * from R
where  a1 op1 C1 and ... and an opn Cn
```

pmr : all op_i are equality tests.   *space* : some op_i are range tests.

Possible approaches to handling such queries ...

1. use index on one a_i to reduce tuple tests
2. use indexes on all a_i, and intersect answer sets

... N-d Queries and Indexes

If using just *one* of several indexes, *which one* to use?

```
select * from R
where  a1 op1 C1 and ... and an opn Cn
```

The one with best *selectivity* for $a_i\ op_i\ C_i$ (i.e. fewest matching tuples/pages)

Factors determining selectivity of $a_i\ op_i\ C_i$

- tend to assume uniform distribution of values in $dom(a_i)$
- equality test on primary key gives at most one match
- equality test on larger $dom(a_i)$ ⇒ less matches
- range test over large part of $dom(a_i)$ is not helpful

... N-d Queries and Indexes

Implementing selection using *one of several* indices:

```
// Query: select * from R where a1op1C1 and ... and anopnCn
// choose ai with best selectivity
TupleIDs = IndexLookup(R,ai,opi,Ci)
// gives { tid1, tid2, ...} for tuples satisfying aiopiCi
PageIDs = { }
foreach tid in TupleIDs
   { PageIDs = PageIDs ∪ {pageOf(tid)} }

// PageIDs = a set of bqix page numbers
...
```

Cost = $Cost_{index} + b_{q_{ix}}$ (some pages do not contain answers, $b_{q_{ix}} > b_q$)

DBMSs typically maintain statistics to assist with determining selectivity

... N-d Queries and Indexes

Implementing selection using *multiple* indices:

```
// Query: select * from R where a1op1C1 and ... and anopnCn
// assumes an index on at least ai
TupleIDs = IndexLookup(R,a1,op1,C1)
foreach attribute ai with an index {
   tids = IndexLookup(R,ai,opi,Ci)
   TupleIDs = TupleIDs ∩ tids
}
PageIDs = { }
foreach tid in TupleIDs
   { PageIDs = PageIDs ∪ {pageOf(tid)} }
// PageIDs = a set of bq page numbers
...
```

Cost = $k.Cost_{index} + b_q$

Exercise 3: One vs Multiple Indices

Consider a relation with r = 100,000, B = 4K, defined as:

```
create table Students (
    id       integer primary key,
    name     char(10), -- simplified
    gender   char(1)   -- 'm' or 'f',
    birthday date      -- 1980 .. 2000
);
... and a query on this relation ...
select * from Students
where  gender='m' and birthday='YYYY-02-29'
```

which has a B-tree index on each attribute ...

- describe the selectivity of each attribute
- estimate the cost of answering using one index
- estimate the cost of answering using both indices

Bitmap Indexes

Alternative index structure, focussing on sets of tuples:

Data File

	Part#	Colour	Price
[0]	P7	red	$2.50
[1]	P1	green	$3.50
[2]	P9	blue	$4.10
[3]	P4	blue	$7.00
[4]	P5	red	$5.20
[5]	P5	red	$2.50

......

Colour Index

red	100011...
blue	001100...
green	010000...

Price Index

< $4.00	110001...
>= $4.00	001110...

Index contains bit-strings of r bits, one for each value/range

---

... Bitmap Indexes

Answering queries using bitmap index:

```
Matches = AllOnes(r)
foreach attribute A with index {
   // select ith bit-string for attribute A
   // based on value associated with A in WHERE
   Matches = Matches & Bitmaps[A][i]
}
// Matches contains 1-bit for each matching tuple
foreach i in 0..r {
   if (Matches[i] == 0) continue;
   t = fetchTuple(i)
   Results = Results ∪ {t}
}
```

... Bitmap Indexes

Storage costs for bitmap indexes:

- one bitmap for each value/range for each indexed attribute

- each bitmap has length *ceil(r/8)* bytes
- e.g. with 50K records and 8KB pages, bitmap fits in one page

Query execution costs for bitmap indexes:

- read one bitmap for each indexed attribute in query
- perform bitwise AND on bitmaps (in memory)
- read pages containing matching tuples

Note: bitmaps could index pages (shorter bitmaps, more comparisons)

Exercise 4: Bitmap Index

45/65

Using the following file structure:

Data File

	Part#	Colour	Price
[0]	P7	red	$2.50
[1]	P1	green	$3.50
[2]	P9	blue	$4.10
[3]	P4	blue	$7.00
[4]	P5	red	$5.20
[5]	P5	red	$2.50
		.....	

Colour Index

red	100011...
blue	001100...
green	010000...

Price Index

< $4.00	110001...
>= $4.00	001110...

Show how the following queries would be answered:

```
select * from Parts
where colour='red' and price < 4.00
```

```
select * from Parts
where colour='green' or colour ='blue'
```

Hashing for N-d Selection

Hashing and *pmr*

47/65

For a *pmr* query like

`select * from R where` a_1 `=` C_1 `and ... and` a_n `=` C_n

- if one a_i is the hash key, query is very efficient
- if no a_i is the hash key, need to use linear scan

Can be alleviated using *multi-attribute hashing* (*mah*)

- form a composite hash value involving all attributes
- at query time, some components of composite hash are known
  (allows us to limit the number of data pages which need to be checked)

MA.hashing works in conjunction with any dynamic hash scheme.

... Hashing and *pmr*

48/65

Multi-attribute hashing parameters:

- file size = $b = 2^d$ pages $\Rightarrow$ use *d*-bit hash values
- relation has *n* attributes: $a_1, a_2, \ldots a_n$
- attribute a_i has hash function h_i
- attribute a_i contributes d_i bits (to the combined hash value)
- total bits $d = \sum_{i=1}^{n} d_i$
- a *choice vector* (*cv*) specifies for all *k* ...
  bit *j* from $h_i(a_i)$ contributes bit *k* in combined hash value

---

MA.Hashing Example

49/65

Consider relation `Deposit(branch,acctNo,name,amount)`

Assume a small data file with 8 main data pages (plus overflows).

Hash parameters: $d=3$ $d_1=1$ $d_2=1$ $d_3=1$ $d_4=0$

Note that we ignore the `amount` attribute ($d_4=0$)

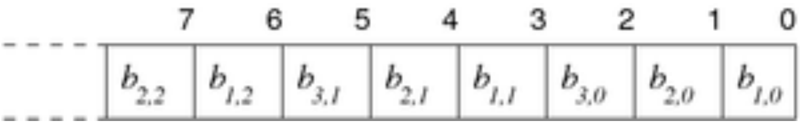
Assumes that nobody will want to ask queries like

```
select * from Deposit where amount=533
```

Choice vector is designed taking expected queries into account.

... MA.Hashing Example

Choice vector:



This choice vector tells us:

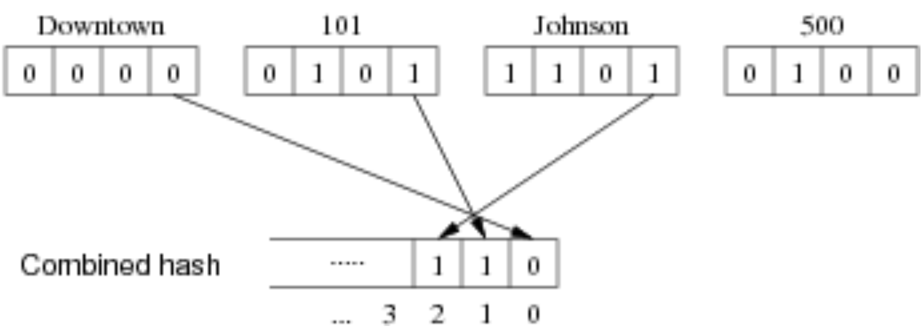
- bit 0 in hash comes from bit 0 of $hash_1(a_1)$ ($b_{1,0}$)
- bit 1 in hash comes from bit 0 of $hash_2(a_2)$ ($b_{2,0}$)
- bit 2 in hash comes from bit 0 of $hash_3(a_3)$ ($b_{3,0}$)
- bit 3 in hash comes from bit 1 of $hash_1(a_1)$ ($b_{1,1}$)
- etc. etc. etc. (up to as many bits of hashing as required, e.g. 32)

... MA.Hashing Example

Consider the tuple:

branch	acctNo	name	amount
Downtown	101	Johnston	512

Hash value (page address) is computed by:



MA.Hashing Hash Functions

Auxiliary definitions:

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

// extracts i'th bit from hash value
#define bit(i,h) (((h) & (1 << (i))) >> (i))

// choice vector elems
typedef struct { int attr, int bit } CVelem;
typedef CVelem ChoiceVec[MaxHashSize];

// hash function for individual attributes
HashVal hash1(Tuple t, int i) { ... }
```

... MA.Hashing Hash Functions

Produce combined *d*-bit hash value for tuple *t*:

```
HashVal hash(Tuple t, ChoiceVec cv, int d)
{
    HashVal h[nAttr(t)+1];  // hash for each attr
    HashVal res = 0, oneBit;
    int     i, a, b;

    for (i = 1; i <= nAttr(t); i++)
        h[i] = hash1(t,i);
    for (i = 0; i < d; i++) {
        a = cv[i].attr;
        b = cv[i].bit;
        oneBit = bit(b, h[a]);
        res = res | (oneBit << i);
    }
    return res;
}
```

Exercise 5: Multi-attribute Hashing

Compute the hash value for the tuple

```
('John Smith','BSc(CompSci)',1990,99.5)
```

where $d=6$, $d_1=3$, $d_2=2$, $d_3=1$, and

- cv = <(1,0), (1,1), (2,0), (3,0), (1,2), (2,1), (3,1), (1,3), ...>
- $hash_1$(`'John Smith'`) = `...0101010110110100`
- $hash_2$(`'BSc(CompSci)'`) = `...1011111101101111`
- $hash_3$(`1990`) = `...0001001011000000`

Queries with MA.Hashing

55/65

In a partial match query:

- values of some attributes are known
- values of other attributes are unknown

E.g.

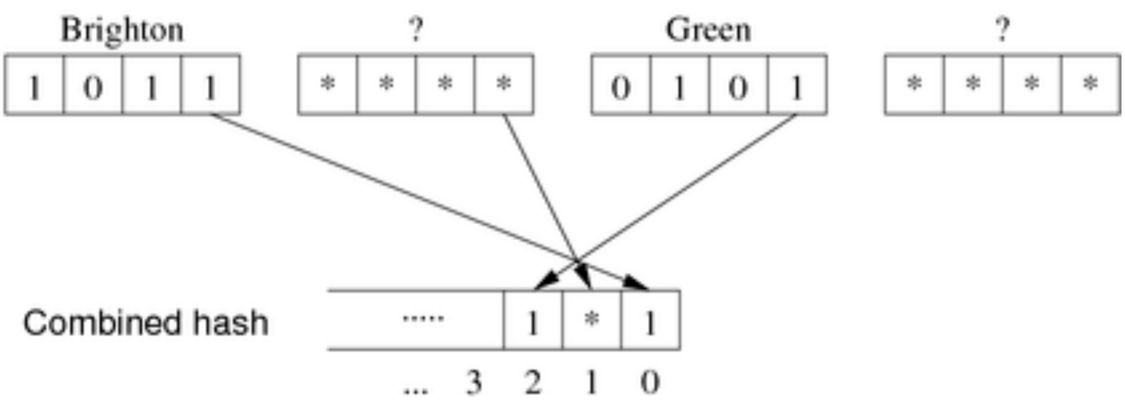
```
select amount
from   Deposit
where  branch = 'Brighton' and name = 'Green'
```

for which we use the shorthand `(Brighton, ?, Green, ?)`

... Queries with MA.Hashing

56/65

In composite hash for query, values for some bits are unknown:



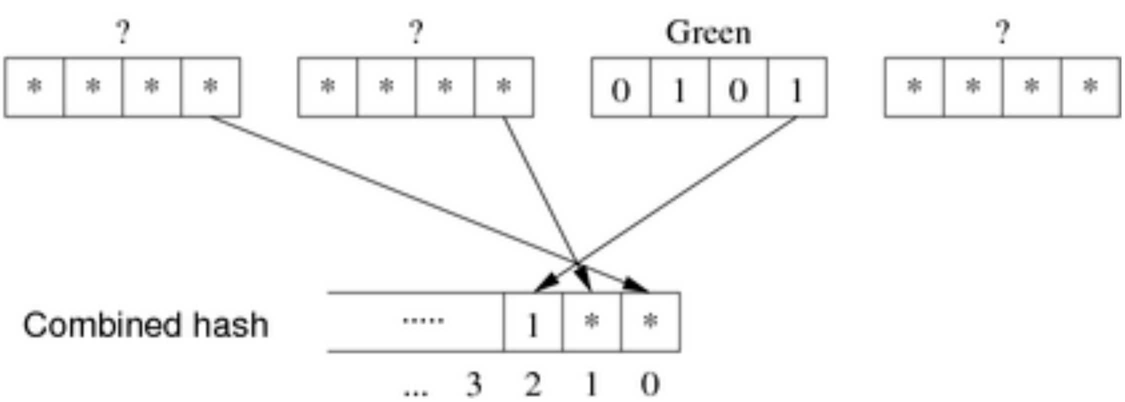
What this tells us: any matching tuples *must* be in pages `101`, `111`

... Queries with MA.Hashing

57/65

Consider the query:

```
select amount from Deposit where name = 'Green'
```



Need to check pages: `100`, `101`, `110`, `111`.

MA.Hashing Query Algorithm

58/65

```
// Builds the partial hash value (e.g. 10*0*1)
// Treats query like tuple with some attr values missing
nstars = 0;
for each attribute i in query Q {
    if (hasValue(Q,i)) {
        set d[i] bits in composite hash
            using choice vector and hash(Q,i)
    } else {
        set d[i] *'s in composite hash
            using choice vector
        nstars++;
    }
}
...
```

... MA.Hashing Query Algorithm


```
...
// Use the partial hash to find candidate pages

r = openRelation("R",READ);
for (i = 0; i < 2**nstars; i++) {
    P = composite hash
    replace *'s in P
        using i and choice vector
    Buf = readPage(file(r), P);
    for each tuple T in Buf {
        if (T satisfies pmr query)
            add T to results
    }
}
```

Exercise 6: Representing Stars

Our hash values are bit-strings (e.g. `0100101110101`)

MA.Hashing introduces a third value (`*` = unknown)

How could we represent "bit"-strings like `01011*1*0**010`?

Exercise 7: MA.Hashing Query Cost

Consider `R(x,y,z)` using multi-attribute hashing where

$d = 9 \quad d_x = 5 \quad d_y = 3 \quad d_z = 1$

How many buckets are accessed in answering each query?

1. `select * from R where x = 4 and y = 2 and z = 1`
2. `select * from R where x = 5 and y = 3`
3. `select * from R where y = 99`
4. `select * from R where z = 23`
5. `select * from R where x > 5`

Query Cost for MA.Hashing

Multi-attribute hashing handles a range of query types, e.g.

```
select * from R where a=1
select * from R where d=2
select * from R where b=3 and c=4
select * from R where a=5 and b=6 and c=7
```

A relation with *n* attributes has 2^n different query types.

Different query types have different costs   (different no. of `*`'s)

Query distribution gives probability p_Q of asking each query type *Q*.

... Query Cost for MA.Hashing

For a relation `R(a,b,c,d)` ...

```
select * from R where a=1
-- has 1 specified attribute (a)
-- has 3 unspecified attributes (b,c,d)

select * from R where b=5 and d=2
-- has 2 specified attributes (b,d)
-- has 2 unspecified attributes (a,c)

select * from R
where a=1 and b=5 and c=3 and d=2
-- has 4 specified attributes (a,b,c,d)
-- has 0 unspecified attributes
```

... Query Cost for MA.Hashing

Consider a query of type *Q* with *m* attributes unspecified.

Each unspecified A_i contributes d_i `*`'s.

Total number of `*`'s is $s = \sum_{i \notin Q} d_i$.

⇒ Number of pages to read is $2^s = \prod_{i \notin Q} 2^{d_i}$.

Ignoring overflows, $Cost(Q) = 2^s$ (where *s* is determined by *Q*)

Including overflows, $Cost(Q) = 2^s(1+Ov)$

... Query Cost for MA.Hashing

Min query cost occurs when all attributes are used in query

$Min\ Cost_{pmr} = 1$

Max query cost occurs when no attributes are specified

$Max\ Cost_{pmr} = 2^d = b$

Average cost is given by weighted sum over all query types:

$Avg\ Cost_{pmr} = \Sigma_Q\, p_Q\, \Pi_{i \notin Q}\, 2^{d_i}$

Aim to minimise the weighted average query cost over possible query types